Spatio-Temporal Multi Data Stream Analysis with Applications in Team Sports

Inauguraldissertation

zur

Erlangung der Würde eines Doktors der Philosophie

vorgelegt der

Philosophisch-Naturwissenschaftlichen Fakultät

der Universität Basel

von

Lukas Probst

aus Deutschland

Basel, 2020

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Prof. Dr. Heiko Schuldt, Fakultätsverantwortlicher und Dissertationsleiter
Prof. Dr. Peter Michael Fischer, Korreferent

Basel, den 23.06.2020

Prof. Dr. Martin Spiess, Dekan

This thesis is dedicated to my beloved girlfriend Lea.

Zusammenfassung

Die Menge der Live-Daten, die über Individuen gesammelt werden können, steigt stetig. Heutzutage können Menschen mit physischen Geräten ausgestattet und mit Kameras überwacht werden, um Information, wie beispielsweise ihre Position, ihren Gesundheitszustand und den Zustand ihrer Umgebung, zu erfassen. Fitnesstracker und Gesundheitsanwendungen, die den Zustand und das Verhalten eines Individuums anhand der Daten analysieren, die über dieses Individuum gesammelt werden, sind bereits weit verbreitet.

Allerdings handeln Menschen selten allein. Stattdessen tendieren sie dazu, in Teams zusammenzuarbeiten, um ein gemeinsames Ziel zu erreichen. So arbeiten zum Beispiel Fussballspieler zusammen, um ein Spiel zu gewinnen, und Feuerwehrleute arbeiten zusammen, um Waldbrände zu löschen. Die Analyse des Teamverhaltens auf der Basis der Daten über die Individuen, die das Team bilden, ist nicht nur sehr interessant, sondern stellt auch verschiedene Herausforderungen an das System, welches die Analysen durchführt. Der Schwerpunkt dieser Dissertation liegt in der Bewältigung dieser Herausforderungen.

Wir definieren ein Datenmodell und ein Systemmodell, um eine theoretische Basis für die Entwicklung eines Systems zu schaffen, welches dazu geeignet ist, als Grundlage für die Implementierung einer Teamverhaltensanalyseanwendung genutzt zu werden. Beide Modelle sind neuartig im Bezug auf die Tatsache, dass sie die Besonderheiten der Teamverhaltensanalyseanwendung, wie zum Beispiel die Semantik der Ein- und Ausgabedaten, berücksichtigen. Ausserdem etablieren wir ein starkes Fundament für die Verwendung der räumlichen und zeitlichen Informationen, welche eine zentrale Rolle in der Teamverhaltensanalyse spielen. Genauer gesagt definieren wir grundlegende räumliche Funktionen und Beziehungen. Zudem präsentieren wir ein ausführliches datenstrombezogenes Zeitmodell, das weit über die bisherige Literatur über Zeitbegriffe in Datenstromanalysesystemen hinausgeht und überdies ein neuartiges Gleichzeitigkeitskonzept beinhaltet.

Nachdem die theoretische Basis geschaffen ist, präsentieren wir StreamTeam, unsere generische Infrastruktur zur Echtzeitdatenstromanalyse, welche dafür entworfen wurde, als Grundlage für die Implementierung von Teamverhaltensanalyseanwendungen genutzt zu werden. Das Datenstromanalysesystem im Herzen von StreamTeam ist eine Prototyp-Implementierung unserer Modelle, welche zusätzlich neuartige Ansätze enthält, um Fachexperten ohne fundierte

Softwareentwicklungskenntnisse bei der Entwicklung eigener Analysen zu unterstützen. Ausserdem präsentieren wir StreamTeam-Football, eine Anwendung zur Echtzeitfussballanalyse, die mit Hilfe von StreamTeam implementiert wurde. StreamTeam-Football ist die erste Analyseanwendung, welche das Teamverhalten in einem Fussballspiel in Echtzeit analysieren und die Analyseresultate sowohl live in einer Benutzeroberfläche darstellen als auch persistent für spätere Aktivitäten speichern kann.

Abstract

The amount of live data about individuals which can be collected is steadily growing. These days, humans can be equipped with physical devices or observed with cameras in order to capture information such as their positions, their health state, and the state of their environment. Fitness trackers and health applications which analyze the state and the behavior of an individual on the basis of the data that are captured for this individual are already widely used.

However, humans rarely act alone but rather collaborate in teams in order to achieve a common objective. For instance, football players collaborate to win a match and firefighters collaborate to extinguish a forest fire. Analyzing the collaborative team behavior on the basis of data about the individuals which form the team is not only interesting but further poses several challenges on the system that performs the analyses. The focus of this thesis is to address these challenges.

We define a data model and a system model in order to provide a theoretical basis for implementing a system that is suited to serve as a foundation for developing team collaboration analysis applications. Both models are novel with respect to the fact that they take the particularities of team collaboration analysis applications, such as the semantics of their input and output data, into account. Moreover, we establish a strong foundation for using the spatial and temporal information which play a central role in analyzing the collaborative behavior of a team. More precisely, we define basic spatial functions and relations and present an extensive stream time model which goes far beyond existing literature on stream time notions and comprises a novel simultaneousness concept.

After establishing the theoretical basis, we present StreamTeam, our generic real-time data stream analysis infrastructure which is designed to be used as a foundation for developing team collaboration analysis applications. The data stream analysis system at the heart of StreamTeam is a prototype implementation of our models which further introduces novel approaches to assist domain experts without a profound software engineering background in developing their own analyses. Moreover, we present StreamTeam-Football, a real-time football analysis application which is implemented on top of StreamTeam. StreamTeam-Football is the first analysis application which performs complex team behavior analyses in a football match in real-time, visualizes the live analysis results in a user interface, and stores them persistently for offline activities.

Acknowledgements

First, I would like to thank my advisor, Prof. Dr. Heiko Schuldt. I am very grateful that Prof. Schuldt provided me the opportunity to contribute to interesting research projects in his research group, not only during my time as a Ph. D. student but also during my Bachelor's and Master's studies. I very much appreciate the feedback and support I got from him over the last years.

Second, I would like to thank Prof. Dr. Peter Michael Fischer from the University of Augsburg for reviewing my thesis.

In addition, I would like to thank my colleagues from the StreamTeam project, Martin Rumo and Philipp Seidenschwarz, especially for their valuable input from a sports science perspective.

My thanks also go to my former and current colleagues at the Department of Mathematics and Computer Science with whom I enjoyed working and sharing my lunch breaks. In particular, I would like to thank Alexander Stiemer not only for being a great office mate but also for being always willing to discuss the latest research problems.

Moreover, I would like to take this opportunity to thank my parents Christina and Andreas for their unconditional support during my academic education.

Last but certainly not least, my biggest "Thank you" goes to my beloved girlfriend Lea to whom I dedicate this thesis.

This work has been partly supported by the Hasler Foundation in the context of the project StreamTeam, contract no. 16074, which is also thankfully acknowledged.

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

List of Algorithms

List of Examples

List of Acronyms

CDF	Cummulative Distribution Function
CEP	Complex Event Processing
DBMS	Database Management System
DotA2	Defense of the Ancients 2
DSMS	Data Stream Management System
FIFO	First In First Out
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KNN	K-Nearest Neighbors
NTP	Network Time Protocol
OSI	Open Systems Interconnection
PCA	Principal Component Analysis
PTP	Precision Time Protocol
REST	Representational State Transfer
RNN	Recurrent Neural Network
SFISM	Swiss Federal Institute of Sport Magglingen
SVM	Support Vector Machine
XML	Extensible Markup Language

List of Symbols

$\prec$	Ordering
$\top, \bot$	True, False
Δts	Time bound
Δd	Distance threshold
λ	Null element
ξ	Sequence number
ρ	Three dimensional position
τ	Processing timestamp
φ	Event phase
T	Ingestion timestamp
$\mathfrak{T}$	Time space
ato	Atomicity flag
co	Code
cat	Data stream category
$contained(\rho, pos)$	$\top$ if $planar(\rho)$ is contained in the polygon spec. by *pos*
$d(\rho_1, \rho_2)$	Distance between ρ_1 and ρ_2
Dom	Domain
ds	Data stream
DS	Set of data streams
dsas	Data stream analysis system
DSAS	Set of data stream analysis systems
dse	Data stream element
DSE	Set of data stream elements
dsp	Data stream partition
DSP	Set of data stream partitions
e	Edge of a graph
E	Set of edges of a graph
ec	Entry component
EC	Set of entry components
ecid	Entry component identifier
eid	Event identifier
gid	Group identifier
gids	Group identifiers tuple

igd	Raw input stream generating device
IGD	Set of raw input stream generating devices
$igdid$	Raw input stream generating device identifier
k	Partitioning key
m	Module
M	Set of modules
$mbr(P)$	Minimum bounding rectangle of P
$name$	Name of a data stream
oe	Opta event
oid	Object identifier
$oids$	Object identifiers tuple
P	Set of positions
$pch(P)$	Planar convex hull of P
pd	Payload
$planar(\rho)$	Planar projection of ρ
pos	Positions tuple
pr	Processor
PR	Set of processors
q	Opta event qualifier
Q	Set of Opta event qualifiers
$\langle dse_1, dse_2\rangle \in SEQ(\Delta ts)$	dse_1 and dse_2 are sequential w.r.t. Δts
sch	Payload schema
$\langle dse_1, dse_2\rangle \in SIM(\Delta ts)$	dse_1 and dse_2 are simultaneous w.r.t. Δts
$\langle ds_1, ds_2\rangle \in EXSEQ(\Delta ts)$	ds_1 and ds_2 are exclusively sequential w.r.t. Δts
st	State
$surface(pos)$	Surface of the polygon defined by pos
T	Timer period
ts	Generation timestamp
ts_{emis}	Emission timestamp
ts_{prod}	Production timestamp
$UID(X)$	A unique tuple identifier for the named tuples in X
V	Set of vertices of a graph
w	Worker
W	Set of workers
wid	Worker identifier
wf	Workflow
WF	Set of Workflows

Part I

Introduction and Motivation

1 Introduction

In the last decade, the capabilities to collect information about people and their environment improved remarkably. Smartphones have emerged from unloved tools for businessmen to esteemed companions of the general population. In addition, new devices such as smartwatches and fitness bracelets have been introduced. As shown in a study conducted by Bitkom Research [Ame19], 81 % of the German population (excluding children below 14 years) stated in 2018 that they are regularly using a smartphone and 87 % of the smartphone users stated that smartphones simplify their daily life. Moreover, 42 % of the smartphone users state that they have connected their smartphone at least once to a smart watch or a fitness bracelet [Ame19].

While smartphones already enable tracking the position of a person, smartwatches and fitness bracelets enable even collecting information about the human body, such as the heart rate of their wearer. Information, such as the air temperature and quality, which cannot be captured with smartphones, smartwatches, and fitness bracelets can be measured with special-purpose sensor devices. While the size of these sensor devices has decreased over the last years, their precision has improved. Moreover, many of these special-purpose sensor devices can be connected to a smartphone or even directly to the internet.

The almost unlimited capabilities to measure data with physical devices and the fact that the mobile network has become faster and cheaper results already in a huge amount of live data. However, physical devices are not the only sources for live data about individuals. In addition, there are solutions to extract information, such as the location or the body temperature of a person, from live videos.

The captured data about a single individual are used, for instance, by a plethora of established fitness trackers and health applications, such as Google Fit [Goo20], Health [App20], and Fitbit [Fit20]. These applications analyze the state (e.g., the average heart rate) and the behavior (e.g., the movement pattern) of an individual on the basis of the data that are measured for this individual (such as its position, heart rate, etc.).

We admit that analyzing the state and behavior of individuals can yield interesting results. However, humans only rarely act alone. Instead, humans tend to collaborate in teams in order to achieve a common objective. This behavior pattern is not a symptom of modern times but observable since humans exist. For instance, humans in the Stone Age formed teams to hunt large animals which an individual would never dare to hunt alone. Nowadays, this behavior pattern is observable for instance in team sports or in disaster management. Members of a football team collaborate in order to win a match against another team. Similarly, firefighters collaborate in order to extinguish a large forest fire.

In consequence, we argue that analyzing the collaborative behavior of the teams by means of analyzing the data about the individuals which form the teams not separately but jointly yield even more interesting results. For instance, although the performance of each football player is important, every football enthusiast knows that a team formed by mid-range players which collaborate very well can win against a team consisting of world-class individualists. Therefore, analyzing the collaborative team behavior in a football match is at least as important as analyzing the performance of the individual players. Similarly, monitoring the health state of each firefighter is definitely important. However, since the firefighters have to collaborate well in order to prevent the fire from spreading, we argue that also analyzing their collaborative team behavior can improve the safety of firefighters and residents.

In the remainder of this chapter, we will describe the challenges which analyzing collaborative team behavior poses on the system that performs the analyses, list the contributions which we make in this thesis to address these challenges, and outline the content of this thesis.

1.1 Challenges

The objective to analyze collaborative team behavior on the basis of data about the individuals which form the team poses several challenges on the system that performs the analyses:

Real-Time The immediate availability of the captured data enables analyzing the behavior of individuals and the collaborative behavior of teams in real-time.

We argue that performing the analysis in real-time is beneficial since having live analysis results while the collaboration is in action enables providing the members of the team with live feedback about how they can improve their individual performance and their interaction with the other members of the team. However, analyzing the data in real-time does not only introduce benefits but also technical challenges. Namely, the data cannot be accumulated to one huge dataset which is analyzed using static data analysis methods. Instead, the data have to be processed in small packets which we denote as data stream elements using a data stream analysis approach.

Multi Data Stream As indicated above, the data about the individuals which are used as input data for analyzing the collaborative behavior of a team are generated by multiple devices which are attached to the individuals (e.g., a smartwatch) or monitor a certain area (e.g., a tracking camera). More precisely, multiple devices perform diverse measurements and emit the results in elements of different data streams. A system for analyzing collaborative team behavior has to support analyzing these elements not only separately for each data stream and/or for each device but jointly.

Modular Analyzing the behavior of individuals and the collaborative behavior of the teams which are formed by these individuals in the same system is meaningful as all analyses are performed on the basis of the same data. However, we argue that it makes sense to implement different analysis subtasks separately and to split a complex analysis subtask into multiple simpler analysis subtasks which perform the complex analysis stepwise. Doing so does not only simplify the implementation of the analyses – especially for domain experts without a profound software engineering background – but further facilitates sharing intermediate results and even using the final results of one analysis as the input for another analysis. In addition, changing user demands should be easily solvable by modifying analysis subtasks or adding new analysis subtasks without changing the rest of the system. In consequence, a system for analyzing collaborative team behavior has to provide support for splitting the overall analysis task into smaller subtasks which are implemented separately.

Scalable The amount of input data for each individual is not constant but depends on the the number of measurements which are performed for the individual. Moreover, the number of individuals which form a team depends on the objective they aim to achieve. Depending on the scenario, the volume and frequency of input data stream elements can range from moderate to overwhelming. In addition, we argue that it is reasonable to analyze the behavior of multiple teams which aim to achieve the same or a very similar objective in parallel in the same system. Hence, a system for analyzing collaborative team behavior has not only to be able to scale with respect to the number of analyses it performs but also with respect to the number of input data stream elements for which these analyses are performed. For this purpose, the system should support parallelism and a distributed deployment of its components.

Spatio-Temporal Especially when analyzing the collaborative behavior of a team, the spatial and temporal information of each data stream element which ships information about an individual is very important. Among others, temporal and spatial information is indispensable to assess the simultaneousness and the proximity of two measurements or actions. In order to use this information properly it is important to have a strong theoretical foundation about the different stream time notions and the basic spatial functions and relations.

1.2 Contributions

In order to address these challenges, we make the following contributions in this thesis:

– We present a data stream model which formally defines data streams, data stream elements, and data stream partitions. Based on investigations of the information which is shipped in data stream elements that are consumed or produced by team collaboration analysis applications, we introduce a novel schema for encoding common information in a consistent way in generic data stream independent attributes and data stream specific information in a data stream specific payload attribute. Moreover, we introduce a distinction between atomic and non-atomic data stream elements as well as a separation of the data streams into four categories which reflect their semantics in collaborative team behavior analysis.

– We define our system model of a data stream analysis system. More precisely, we define the conceptual and physical components of a data stream analysis system, discuss and describe how parallelism is supported, describe the processing procedure at the physical components, and address machine and network related aspects of our system model. A novelty of our system model is that we consider the different data stream categories and differentiate between the input and output streams of a data stream analysis system. Moreover, we introduce sophisticated well-formation constraints. In addition, we do not limit our system model to the components of a single data stream analysis system. Instead, we also model the devices which produce the input data for the data stream analysis and state which additional constraints have to be regarded when deploying multiple coexisting data stream analysis systems.

– We present an extensive stream time model which establishes the theoretical foundation for using temporal information in data stream analysis. More precisely, we define, compare, and discuss different time notions as well as the orderings introduced by the sequence numbers contained in and by the timestamps which can be assigned to the data stream elements. In doing so we go far beyond existing literature on time notions in data stream analysis. Moreover, we present a novel simultaneousness concept which covers if two data stream elements refer to approximately the same moment in time or not and if it is even possible in a certain team collaboration analysis scenario that two elements refer to approximately the same moment in time.

– We discuss why it is impossible to define generic spatial functions and relations on a data stream element level without introducing ambiguities or violating application demands. To nevertheless provide a theoretical foundation for performing real-time analyses of spatial data stream elements we define basic spatial functions and relations for arbitrary positions which can be used as building blocks to develop the logic for detecting, calculating, and generating collaborative team events, states, and statistics.

– We present StreamTeam, our generic real-time data stream analysis infrastructure, which contains our prototype implementation of a data stream analysis system which expects data to be structured as defined in our data stream model, whose architecture is designed according to our system model, and which supports all time notions that we define in our stream time model. StreamTeam introduces novel approaches to modularize the code and to facilitate separating the analysis by application-specific keys in order to assist domain experts without a profound software engineering background in developing their own analyses.

- We present StreamTeam-Football, the real-time football analysis application which we have implemented on top of StreamTeam. To the best of our knowledge, StreamTeam-Football is the first analysis application which performs complex team behavior analyses in a football match in real-time, visualizes the live analysis results in a user interface, and stores them persistently for offline activities such as video scene retrieval.
- We present the results of our qualitative and quantitative evaluations which show that StreamTeam-Football is a non-trivial real-time team collaboration analysis application which fulfills the analysis demands of football coaches, match analysts, and sports scientists, that StreamTeam-Football is able to analyze multiple football matches in parallel in real-time, that StreamTeam's data stream analysis systems scales with respect to the number of processed and emitted data stream elements, and that the theoretical statements on the (un)ambiguity of the diverse timestamps which we pose in our stream time model are correct.

1.3 Thesis Outline

This thesis is separated into four parts. In the remainder of Part I, we will present the real-time football analysis scenario which we use as the running example in this thesis (see Chapter 2).

In Part II, we will cover the model and thus the theoretical part of this thesis. For doing so, we will first describe the fundamentals of data stream analysis and define the mathematical notations which we use in our model (see Chapter 3). Subsequently, we will present our data stream model (see Chapter 4), our system model (see Chapter 5), our stream time model (Chapter 6), and our definitions of the basic spatial functions and relations (Chapter 7).

Subsequently, we will cover our implementations and evaluations and thus the technical part of this thesis in Part III. More precisely, Chapter 8 presents StreamTeam, our generic real-time data stream analysis infrastructure, Chapter 9 presents StreamTeam-Football, our real-time football analysis application, and Chapter 10 presents the qualitative and quantitative evaluations.

Finally, we will conclude the thesis in Part IV by means of discussing related work (see Chapter 11), summarizing the content of the thesis (see Chapter 12), and proposing ideas for future work (see Chapter 13).

2

Example Scenarios

In this chapter, we will describe the real-time football analysis scenario that we will use as the running example throughout this thesis and whose implementation we will present in Chapter 9. In doing so, we will show that developing a real-time football analysis application requires addressing all challenges listed in Section 1.1. Subsequently, we will list some other scenarios which can also profit from the contributions we make with this thesis as they demand solving the same challenges.

2.1 Real-Time Football Analysis

During a football match, the players of a team collaborate in order to shoot goals and to prevent the opposing team from shooting goals and thus to win the match. The most obvious collaboration is that the players of a team pass the ball to each other. However, also those players which are not in possession of the ball collaborate. For instance, multiple players approach the opposing player who is in possession of the ball at the same time in order to generate a pressing situation. Moreover, the spatial formation of the whole team has a huge impact on the match. For instance, the attacking team can spread to generate more passing options and the defense players can form a straight line in order to set offside traps. Already these simple examples which are understandable for a layman – experts, such as football coaches, match analysts, and sports scientists, can discuss for hours about diverse tactics how to collaborate with and without the ball – show the huge potential of analyzing the collaborative team behavior in football matches.

Knowing the position of each player and the ball is mandatory for analyzing a football match. There are many sensor-based and video-based tracking systems which can be used to capture the positions of all players and the ball with a sufficient frequency.[1] Some of these systems even support emitting the positions in real-time in data stream elements. In addition, the players can be optionally equipped with devices which capture information about the state of the players. For instance, each player can be equipped with a fitness bracelet that periodically measures the heart rate of the player and ships this information in a heart rate stream element.

Different stakeholders could benefit from getting the results of the collaborative team behavior analysis not only after the match but live during the match. For instance, coaches could benefit from having a real-time user interface which shows statistics, visualizes events, and highlights the spatial arrangement of the players. Such a tool would not replace coaches but assist them in their decision making process. They could use the visualized analysis results to give live feedback to individual players, to identify necessary substitutions, and to modify the tactics of the whole team. Moreover, broadcasters could follow a similar approach by providing a second screen application which provides fancy graphs and visualizations for the customers. Such an application can personalize the television experience, as each customer can decide on his/her own which statistics he/she wants to see, and thus improve the customer experience remarkably. In consequence, we argue that there is a demand for performing the collaborative team behavior analysis in a way that the results are available within seconds and thus for addressing the real-time challenge listed in Section 1.1.[2] Although, it is possible to perform the analysis manually while still meeting or at least only slightly violating these real-time demands, doing so is quite labor intensive and thus expensive. Therefore, we argue that it is the better option to perform the collaborative team behavior analysis fully automatically in an application that is implemented on top of a data stream analysis system. This data stream analysis system has to be able to jointly analyze the data which multiple devices emit in elements of multiple input streams and thus to address the multi data stream challenge listed in Section 1.1.

As indicated above, the multitude of analyses which can be conducted to assess the performance of individual players as well as the collaborative team behavior is almost unlimited. Since all these analyses are performed on the basis

[1] An overview about these tracking systems will be given in Section 11.2.3.1.

[2] Although, a real-time user interface, of course, benefits from receiving new data with subsecond latencies, we argue that in football analysis also latencies in the low second range are tolerable.

of the same input data, it is reasonable to combine all of them to a single football analysis application which is implemented on top of a single data stream analysis system. However, we argue that it is also reasonable to separate the implementation of the different analysis subtasks. For instance, it makes sense to separate the code for generating player and team heatmaps from the code for detecting passes. Moreover, it is reasonable to split complex analyses into multiple analysis subtasks. For instance, we admit that it is possible to detect pass sequences directly on the basis of raw position data, but we argue that it is the better option to first detect ball possession changes, then use this information to detect single passes, and then combine the passes to pass sequences. Doing so does not only simplify the implementation of each analysis subtask and thus assists match analysts without a profound software engineering background in developing their own analysis subtasks but enables further sharing intermediate results. For instance, the ball possession changes which are detected to detect passes are also a helpful input for analyzing the pressing behavior of the attacking team. In addition, separating the analysis task also facilitates reacting to changing user demands which are likely to occur if the coach of the team changes. If the new coach demands, for instance, that the pressing analysis is modified since he/she has a different definition of pressing in mind, this can be done by changing only the code of a single analysis subtask. Moreover, if the new coach requests a completely new analysis, this new analysis can be added to the application by implementing the new analysis subtask without changing the code of the existing analysis subtasks. Because of these reasons, we argue that the data stream analysis system on top of which the football analysis application is implemented has to provide support for splitting the overall analysis task into cleanly separated analysis subtasks and thus to address the modular challenge listed in Section 1.1.

Analyzing the collaborative team behavior in a football match requires processing frequent position updates. If the position of every player and the ball is measured and emitted in a new position stream element only ten times per seconds, this results already in 230 position stream elements per second. However, many tracking systems generate data with a higher frequency.[3] Moreover, the number of position stream elements scales linearly with the number of matches which have to be analyzed in parallel. If the real-time football analysis application is not deployed by a club to analyze the matches of its team but by a sports analysis company to analyze all matches of multiple leagues, the num-

[3] For instance the TRACAB Optical Tracking dataset [Chy20c] which we use in our evaluation (see Section 10.1.2) contains a new position for each player and the ball every 40 milliseconds.

ber of concurrent matches and thus the number of position stream elements per second can become huge. In addition, as mentioned above, the analysis application can comprise a plethora of different analyses. In consequence, the data stream analysis system which is used as a foundation for implementing the real-time football analysis application has to support parallelism and a distributed deployment and thus to address the scalable challenge listed in Section 1.1 in order to be able to scale with respect to the number of analyses and with respect to the number of input stream elements for which these analyses have to be performed.

Almost all collaborative team analyses which a football analysis application might conduct can be boiled down to performing calculations on the basis of spatio-temporal information. For instance, calculating the surface of the area which the players of a team span requires calculations on the basis of the positions where all players of the team were located at the same moment in time. Moreover, detecting a ball possession change requires observing the velocity and the moving direction of the ball – both information can be calculated using the position history of the ball – and identifying the closest player. Because of this, we argue that it is important to address the spatio-temporal challenge listed in Section 1.1 in order to be able to implement the analyses on the basis of a strong theoretical foundation about the different stream time notions and the basic spatial functions and relations.

To sum up, developing a real-time football analysis application which analyzes the collaborative team behavior in football matches requires addressing all challenges listed in Section 1.1.

2.2 Other Scenarios

Although we use the football analysis scenario as the running example throughout this thesis, there are also other scenarios in which it is beneficial to analyze the collaborative team behavior in real-time.

First, football is not the only team sports which could profit from analyzing the collaborative behavior of the team in real-time. As presented in a Wintergreen Research study [Win17], the general sports analytics markets is already big ($ 764.3 million in 2016) and expected to continue growing in the next years (up to $ 15.5 billion in 2023). Other famous team sports are, for instance, American football, ice hockey, and basketball. Despite of their differences, all these team sports share that players collaborate in order to score and prevent the opposing team from scoring. Analyzing these team sports in real-time poses the same challenges as the real-time football analysis scenario. In consequence, our thesis does not only make contributions towards analyzing football matches but arbitrary team sports matches in real-time. In fact, we even argue that the real-time football analysis application which we will present in Chapter 9 can be modified to analyze other team sports.

Second, individuals collaborate in teams to manage disasters. For instance, firefighters collaborate in teams in order to extinguish large forest fires. An application which analyzes the health state of the individuals as well as the collaborative behavior of the teams could not only help extinguishing the fire faster but even improve the safety of the firefighters. Although such disaster management scenarios pose some additional challenges which we do not cover in this thesis (e.g., observing firefighters requires considering failure tolerance and long-lasting network partitions which is out of the scope of this thesis), they share the challenges of the team sports scenarios. Therefore, we argue that also disaster management scenarios can profit from the contributions we make in this thesis.

Part II

Model

3 Fundamentals

In Part II, we will present our model which underlies the implementation of our generic data stream analysis infrastructure and our real-time football analysis application. More precisely, we will formally define our data stream model (Chapter 4) and our system model (Chapter 5). Subsequently, we will establish the theoretical foundation for the temporal and spatial focus of our analyses. That is, for one thing, we will discuss different stream time notions, the orderings they introduce, and our novel simultaneousness concept (Chapter 6), and for another thing, we will present basic spatial functions which serve as a foundation for analyzing spatial data stream elements (Chapter 7).

However, before we dig into the details of our model, we will present the necessary fundamentals in this chapter. First, we will provide the fundamentals of data stream analysis. Subsequently, we will define the mathematical notations which we will use in our definitions, examples, theorems, and proofs.

3.1 Data Stream Analysis

As its name already implies, the *data stream analysis* research field deals with the analysis of data streams. The main difference between data stream analysis and static data analysis is the nature of the data that have to be analyzed.

In static data analysis as performed by systems such as MapReduce [DG04] and Spark[1] [ZCD+12], there is a static dataset which serves as an input for the analysis. This input dataset is already fully available before the analysis starts and does not change during the analysis. In consequence, the size of the input

[1] Note that we distinguish in this thesis between the original "Spark" published in [ZCD+12] and its streaming extension published in [ZDL+13] to which we refer with the term "Spark Streaming".

dataset is well known or can at least be determined before the analysis starts. Furthermore, it is possible to access information from the input dataset multiple times by iterating multiple times over the dataset or even by reaccessing specific data items if supported by the static data analysis system.

In contrast, in data stream analysis there is no input dataset which is available in its entirety from begin on. Instead, the input data for the analysis are materialized as a continuous and potentially unlimited flow of information. That is, new input data arrive over time in packets called data stream elements. Moreover, the data which arrive disappear again if they are not stored explicitly in state. Since typically components which consume data streams do not store the data stream elements completely in their state but (if at all) keep only some selected, potentially aggregated information, the input data in data stream analysis are volatile. This volatility implies that it is not possible to simply access information from past data stream elements. Therefore, it has to be carefully considered which information has to be stored in state for future usage.

Moreover, data stream analysis usually entails the implicit requirement that the analysis has to be performed in real-time. That is, incoming data stream elements should be analyzed and the analysis results should be made available as fast as possible demanding the analysis procedure to introduce as little latency as possible. For instance, if the positions of football players and the ball are packed into data stream elements and transferred immediately while a match is in progress, this is done to get real-time insights about the match.

In contrast, static data analysis typically does not imply such strong temporal requirements. The reason for this is that, since the input dataset has already been gathered over a longer period of time, it is usually negligible if the analysis takes a little longer. For instance, if a dataset contains the position of a whole football match, the match has already ended and thus waiting a few more minutes for the analysis results does not change much.

The data stream analysis systems which we regard in this thesis consume data streams as its sole input, perform analyses on the basis of the contained data (and the information in the state of the analysis system components), and emit data streams shipping the analysis results as its sole output (see Figure 3.1). This implies that we assume that components of a data stream analysis system neither store analysis results in nor read additional information from files, remote databases, or any other data source. If information stored in external sources is required as input data by a data stream analysis system, an external device has to emit this information as elements of a data stream. Moreover,

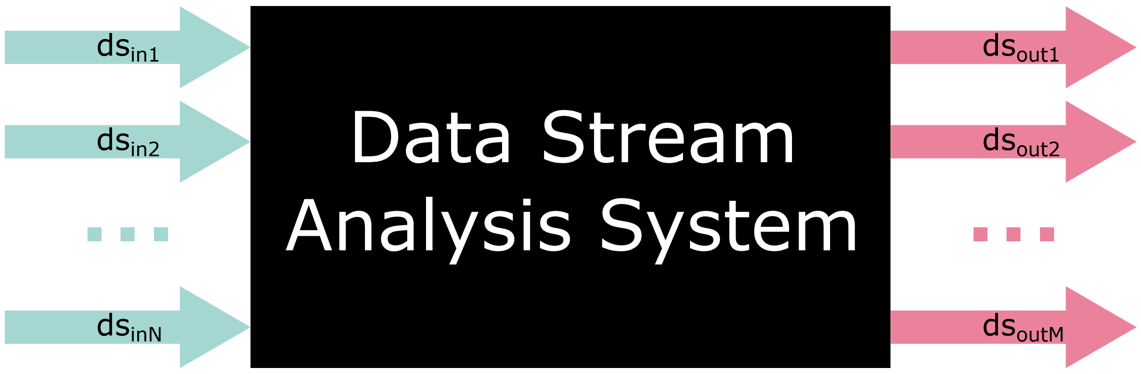


Figure 3.1 Data Stream Analysis System regarded as a Black Box. The black box represents the data stream analysis system. The mint and red arrows visualize n input streams (ds_{in1} to ds_{inN}) and m output streams (ds_{out1} to ds_{outM}).

if analysis results should be stored in files or remote databases, an external consumer which consumes the data streams whose elements ship the analysis results and which takes care of the storing procedure is required.

Figure 3.1 illustrates the data stream analysis system as a black box. The reason for this is that there are many diverse data stream analysis systems. In the following, we will categorize these systems with respect to their analysis specification approach into two super-categories and four sub-categories (see Figure 3.2).

There are *language-based* data stream analysis systems in which the analyses are specified in a dedicated system-specific language. These systems can be further separated into Data Stream Management Systems (DSMSs) and Complex Event Processing (CEP) systems. DSMSs, such as Aurora [ACÇ+03], STREAM/CQL [ABB+03; ABW06], and TelegraphCQ [CCD+03], have been developed by the database research community and can be seen as a derivative of classical Database Management Systems (DBMSs) [CM12b]. Instead of processing non-static queries on more or less static data, these systems process static so-called continuous queries which each define a part of the overall analysis task for non-static data which arrive in data stream elements [CM12b]. CEP systems, such as Amit [AE04], Cayuga [DGP+07], PB-CED [AÇT08], RACED [CM09], and T-REX/TESLA [CM10; CM12a; CM13], have been developed by a different research community and "can be seen as an extension to traditional publish-subscribe" [CM12b]. Instead of subscribing messages that are published under a certain topic or with a certain content, these systems enable specifying complex event patterns which consider also the past [CM12b]. For a more extensive description and categorization of language-based data stream analysis systems we refer the reader to [CM12b] and to [AMU+17].

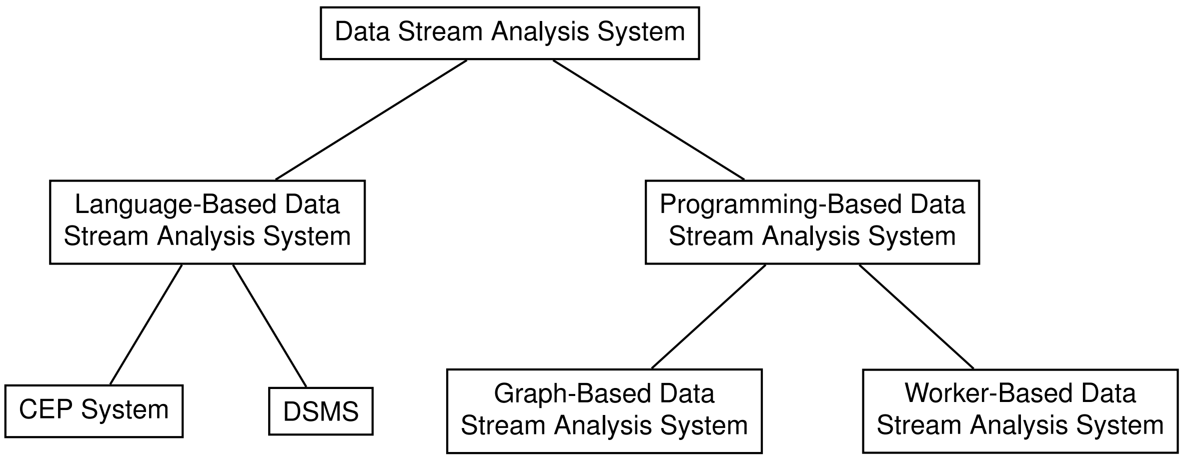

Data Stream Analysis System
Language-Based Data Stream Analysis System
Programming-Based Data Stream Analysis System
CEP System
DSMS
Graph-Based Data Stream Analysis System
Worker-Based Data Stream Analysis System

Figure 3.2 Data Stream Analysis System Class Hierarchy. The tree illustrates the class hierarchy which results from the different analysis specification approaches of the data stream analysis systems.

Moreover, there are *programming-based* data stream analysis systems in which the overall analysis task is specified in a normal state-of-the-art programming language such as Java. These programming-based data stream analysis systems can be further separated with respect to how the overall analysis task is implemented. On the one hand, here are *graph-based* data stream analysis systems, such as Apache Spark Streaming [ZDL$^+$13][2] and Apache Flink [CKE$^+$15], in which the overall analysis task is implemented by concatenating higher-order functions to form a graph. The distribution onto system components which perform the analysis workload is performed automatically by the system. On the other hand, there are *worker-based* data stream analysis systems, such as Apache Storm [TTS$^+$14], Apache Samza [NPP$^+$17][3], and MillWheel [ABB$^+$13], in which the developers themselves split the overall analysis task to multiple components, denoted as workers in this thesis (see Section 5.2), which (or more precisely whose processors, see Section 5.3) each perform a subtasks of the analysis. The code of each worker and thus the logic for each analysis subtask is implemented separately. In consequence, the main difference between worker-based and graph-based data stream analysis systems is that worker-based data stream analysis systems enforce that the code of the overall analysis task is separated cleanly while graph-based data stream analysis systems only support

[2] Note that Spark Streaming [ZDL$^+$13] does not process each data stream element individually but constructs micro-batches which it processes in the same way as regular Spark [ZCD$^+$12]. As the batch size can be set very small to achieve real-time performance, Spark Streaming is nevertheless typically categorized as a data stream analysis system.

[3] Note that even Samza's new high-level API (see Literature Discussion 8.1) does not convert Samza into a graph-based data stream analysis system as workers are still implemented separately.

this.[4]

We argue that language-based data stream analysis systems share, despite their undeniable differences, the problem that it is very complicated or even impossible (depending on the expressiveness of the language) to specify very complex analyses such as those of the football analysis application envisioned in Section 2.1 in their system-specific languages. This assessment is backed by Röger and Mayer as they state that "[i]mperative programming increases expressiveness as the definition of operations is not limited by a declarative language" [RM19]. Moreover, we argue that enforcing a clean code separation is better than only supporting it since we identified splitting the overall collaborative team behavior analysis into analysis subtasks as beneficial for multiple purposes (such as sharing intermediate results, modifying existing analyses, and adding new analyses) and since we aim to assist domain experts without a profound software engineering background in developing analyses (see Chapter 1). Therefore, we have decided to follow the worker-based data stream analysis system approach in our work. In the following chapters we will give more details on how our model defines data streams and fills the details of the black box depicted in Figure 3.1.

3.2 Notations

The notations and semantics of our model are based on sets, tuples and predicate logic. Minuscules (e.g., x) denote elements, capitals (e.g., X) and curly brackets (e.g., $\{y, z\}$) denote sets, and angle brackets (e.g., $\langle y, z\rangle$) denote tuples.[5] Capitals with a hat (e.g., $\widehat{X}$) denote global sets that comprise all elements of a certain type. Dom_x denotes a domain and λ is used as a dedicated null element. The symbols $\top$ and $\bot$ are used to mark a flag as true and false, respectively.

Referring the cardinality of a set $|\{\ldots\}|$ (e.g., $\left|\{y,z\}\right| = 2$), we define $|\langle\ldots\rangle|$ to be the number of elements of a tuple (e.g., $\left|\langle y,z\rangle\right| = 2$) and $|\lambda|$ to be zero (i.e., $|\lambda| = 0$). In contrast, $|x|$ with x being a numerical element is the normal absolute value function (e.g., $|-5.3| = |5.3| = 5.3$).

[4] Graph-based data stream analysis systems also support implementing the overall analysis task directly in the higher-order function graph and thus in a single code block. Outsourcing logic from the higher-order function graph into separate files (or at least clearly separated code blocks) is only good practice but not enforced.

[5] Note that we deviate from this rule by using the capital T to denote the period in which the timer of a worker triggers the code execution (see Section 5.2.1) in order to be consistent with the period symbol that is used in physics.

Moreover, we use the dot-operator with numbers to access elements of tuples (e.g., $x.2 = 37$ if $x = \langle 25, 37 \rangle$). If the tuple has named attributes the dot-operator can also be used with symbols (e.g., $x.z = 37$ if the tuple is named with $\langle y, z \rangle$).

The sole exception from using angle brackets to denote tuples are the parameter tuples of functions which we denote with parentheses. That is, the expression $foo\,(x, y)$ denotes that the function foo expects the ordered list of parameters x and y, and the expression $foo\,(\langle 25, 37 \rangle, 12.7)$ denotes that function foo is evaluated with $x = \langle 25, 37 \rangle$ and $y = 1.27$ and thus with $\langle \langle 25, 37 \rangle, 12.7 \rangle$ as the parameter tuple.

If a set X contains only tuples with the same attributes, an interesting question is how a tuple contained in X can be uniquely identified, or more precisely, which attributes are required to uniquely identify a tuple contained in X. We denote such a set of attributes as a unique tuple identifier $UID(X)$ which we formally define as follows:

Definition 3.1 Unique Tuple Identifier

The non-empty attribute set $UID(X) = \{uid_1, uid_2, \ldots, uid_n\}$ is a unique tuple identifier for the named tuples contained in set X.
That is, each tuple $x \in X$ is uniquely identified by its values for the attributes contained in $UID(X)$.

$$\forall\, v_1 \in Dom_{uid_1}, v_2 \in Dom_{uid_2}, \ldots, v_n \in Dom_{uid_n} : \left(\exists_{\leq 1}\, x \in X : \bigwedge_{uid_i \in UID(X)} x.uid_i = v_i \right)$$

Moreover, $UID(X)$ is minimal. That is, there is no real subset Y of $UID(X)$ which qualifies as a unique tuple identifier since each tuple $x \in X$ can be uniquely identified by its values for the attributes contained in Y.

$$\nexists\, Y \subset UID(X) : \left(Y \neq \emptyset \wedge \forall\, v_1 \in Dom_{uid_1}, v_2 \in Dom_{uid_2}, \ldots, v_n \in Dom_{uid_n} : \left(\exists_{\leq 1}\, x \in X : \bigwedge_{uid_i \in Y} x.uid_i = v_i \right) \right)$$

Our unique tuple identifier concept is very similar to the (primary) key concept of relational databases [Cod70]. In fact, in the same manner in which a relation in a relational database can have multiple key candidates, a set of tuples might also have multiple potential unique tuple identifiers. We could have modeled that by defining *UID*(X) as a set of potential unique tuple identifiers and thus as a set of attribute sets. However, in this thesis, we will never discuss multiple potential unique tuple identifiers. Instead, we will always define only a single unique tuple identifier to state which attributes can be used for a unique identification. For instance, we will define which attributes of a data stream element uniquely identify it in the global set of all data stream elements. In consequence, we have decided to define *UID*(X) as done above in order to improve the simplicity and conciseness of the following definitions. Note that this restriction is also covered in our formal unique tuple identifier definition as Definition 3.1 states that *UID*(X) is only "a" but not "the" unique tuple identifier.

4

Data Stream Model

During team collaboration analysis, various kinds of raw input, event, state, and statistics data are consumed and produced. If the analysis is performed in a data stream analysis system, these data are transferred in data streams.

In this chapter, we will formally define our data stream model which is inspired by Brettlecker's model [Bre08] and which adopts some fundamental concepts of Apache Samza [Sam17d]. More precisely, we will first formally define data streams, data stream elements, and data stream partitions. Subsequently, we will take a closer look at the atomicity of data streams and at the different stream categories. Finally, we will give detailed examples to illustrate our model.

4.1 Data Streams

A data stream (*ds*) is a continuous and potentially unlimited flow of information in a (distributed) computer system. The information transferred in a data stream is contained in data stream elements which we will model in Section 4.2.

Let $\widehat{DS}$ be the global set of all data streams and Dom_{name} be the domain for the data stream names. We formally define a data stream as follows:

Definition 4.1 Data Stream

A data stream *ds* is a 4-tuple $\langle name, cat, ato, sch \rangle$ where $name \in Dom_{name}$ is a name which uniquely identifies the data stream (i.e., $UID(\widehat{DS}) = \{name\}$), $cat \in \{\text{"rawInput"}, \text{"event"}, \text{"state"}, \text{"statistics"}\}$ is the category of the data stream, $ato \in \{\top, \bot\}$ is a flag that specifies if the data stream is atomic ($\top$) or not ($\bot$), and *sch* is the payload schema of the data stream elements belonging to this data stream.

Depending on the data stream, elements contain data with various semantics. A data stream element might contain new sensor measurements such as the current position and heart rate of a football player or other raw input. Alternatively, a data stream element may contain all information of a detected event such as a successful pass or updates of a prolonged event such as a duel between two players. Moreover, a data stream element can contain a calculated state such as the current surfaces of the areas spanned by the players of a team. Finally, a data stream element might transport the latest statistics such as the average velocity and heart rate of a player in the last two minutes. The category (*cat*) of each data stream classifies it into one of four main categories, namely raw input streams, event streams, state streams, and statistics streams. More details about the different categories will be given in Section 4.5.

As we will show in Section 4.2, many pieces of information which are shipped in data stream elements can be encoded in a generic way. However, this is not the case for all pieces of information as the elements of different data streams can contain very different information. The data stream specific pieces of information are encoded in the flexible payload of a data stream element. In order to nevertheless guarantee a consistent information encoding at least on a data stream level, each data stream defines a schema (*sch*) for the payload of its elements. This schema can be chosen arbitrarily. It can be as sophisticated as a JavaScript Object Notation (JSON) schema, an Extensible Markup Language (XML) schema, or a Google Protocol Buffer [Goo19] message type (which might even comprise flexible parts such as arrays), or as simple as a well-defined value sequence for a semicolon-separated String. While our implementation leverages Google Protocol Buffer message types (see Section 8.2.4), we will use semicolon-separated Strings in all examples we will give throughout this thesis for the sake of brevity.

As we will present in more detail in Chapter 5, a data stream is emitted by an arbitrary number of producers and consumed by an arbitrary number (incl. zero) of consumers. In order to enable producers to advertise and consumers to subscribe elements of a data stream, every data stream has a name (*name*) that uniquely identifies it. This approach is followed also in literature such as [ABB+13], [KNR11], and [KK15]. The domain of this name can be chosen arbitrarily. In our work, we use Strings to identify data streams. Moreover, the name is not only a unique data stream identifier but can also carry semantics that can be leveraged during the analysis. For instance, the name "playerSensorInput" enables inferring that elements of this data stream contain sensor measurements concerning a player while the name “successfulPassEvent” implies that elements of this data stream represent successful pass events.

As mentioned above, data streams and thus their elements are used to ship data with various semantics. The atomicity flag of a data stream (*ato*) enables distinguishing atomic and non-atomic data streams. We denote a data stream as *atomic* if one of its elements contains the data of a raw input, an event, a state, or a statistic in its entirety, i.e., if the data are shipped as a whole in a single data stream element. A data stream whose elements contain only updates of an event is denoted as *non-atomic*. Elements of atomic data streams are denoted as atomic data stream elements and elements of non-atomic data streams are denoted as non-atomic data stream elements. More details regarding the atomicity of data streams and their elements will be given in Section 4.4.

4.2 Data Stream Elements

As mentioned in Section 4.1, the information transferred in a data stream is shipped in data stream elements. In our model, we consider data stream elements to be the smallest data transfer unit. That is, we do not consider network transmission details but ignore the lower layers of the Open Systems Interconnection (OSI) model [DZ83]. Moreover, we define each data stream element (*dse*) to belong to exactly one data stream (*ds*).

Since we have given only some first rough insights regarding the content of a data stream element yet, we will take a closer look at the content of the elements of five sample data streams from the football analysis scenario before we define and explain our formal data stream element model.

First, assume that the latest position and heart rate of a player measured by a sensor attached to this player are shipped periodically in an element of a dedicated player sensor input stream. In this case, each player sensor input stream element contains the measured values, i.e., the latest position and the heart rate of the player. Moreover, each element contains two identifiers to identify the player and his/her team as well as a timestamp which specifies the moment when the sensor conducted the contained measurements.

Second, assume that information about a detected successful pass event is transferred in an element of the successful pass event stream. Each successful pass event stream element contains the position where the ball was kicked, the position where the ball was received, as well as the length, velocity, and angle of the pass. Moreover, each element contains a timestamp which specifies when the successful pass took place and three identifiers to identify the player who kicked the ball, the player who received the ball, and the team to which both players belong.

Third, assume that updates of a prolonged duel event are shipped in elements of the duel event stream. Every duel event stream element contains the latest positions of and the distance between the two players which fight for the ball, a player and team identifier for both players, and a timestamp which specifies the moment in time to which the event update refers.

Fourth, assume that information about the areas which are spanned by the players of a team is published periodically in elements of the team area state stream. Each team area state stream contains the identifier of the team, the surface of the minimum bounding rectangle and the planar convex hull spanned by the players of the team (see Section 7.2), and a timestamp which specifies the moment in time to which the state refers.

Finally, assume that there is a fitness statistics stream whose elements contain the latest fitness statistics of a player or a team for different time intervals. Each fitness statistics stream element contains a timestamp which specifies the end of the time interval, the length of the time interval, the average heart rate, and the average velocity. If the element contains statistics of a player, it further contains a player identifier and a team identifier. If the element contains statistics of a team, it contains only a team but no player identifier.

When considered together, these examples lead to the following conclusion: Each data stream element can but does not have to contain one or multiple positions, object (e.g., player) identifiers, and group (e.g., team) identifiers. Moreover, each data stream element contains a timestamp. In addition, each data stream element can contain some arbitrary data stream specific pieces of information.

In consequence, we model the timestamp, the positions, and the affiliation information (i.e., the object and group identifiers) in a generic, data stream independent way by means of defining dedicated attributes for these information. The data stream specific pieces of information which do not fit into these attributes are covered in our model by a data stream specific payload attribute which enables encoding arbitrary information.

Let $\widehat{DSE}$ be the global set of all data stream elements of all data streams, Dom_k be the key domain, Dom_{eid} be the event identifier domain, Dom_{oid} be the object identifier domain, and Dom_{gid} be the group identifier domain. We formally define a data stream element as follows:

Definition 4.2 Data Stream Element

A data stream element dse is a 10-tuple $\langle ds, k, eid, \varphi, \xi, ts, pos, oids, gids, pd \rangle$ where $ds \in \widehat{DS}$ is the data stream the element belongs to, $k \in Dom_k$ is the key, $eid \in Dom_{eid} \cup \lambda$ is the event identifier that groups data stream elements representing a non-atomic event (or null), $\varphi \in \{\text{"start"}, \text{"active"}, \text{"end"}\} \cup \lambda$ is the phase (or null), $\xi \in \mathbb{N}_0$ is the sequence number that orders the data stream elements belonging to the same data stream and having the same key, $ts \in \mathbb{N}_0$ is the explicit timestamp, $pos \in \{\langle \rho_1 \rangle, \langle \rho_1, \rho_2 \rangle, \langle \rho_1, \rho_2, \rho_3 \rangle, \ldots\} \cup \lambda$ with $\rho_i = \langle x, y, z \rangle$ and $x, y, z \in \mathbb{R}$ is a tuple containing positions (or null), $oids \in \{\langle oid_1 \rangle, \langle oid_1, oid_2 \rangle, \langle oid_1, oid_2, oid_3 \rangle, \ldots\} \cup \lambda$ with $oid_i \in Dom_{oid}$ is a tuple containing object identifiers (or null), $gids \in \{\langle gid_1 \rangle, \langle gid_1, gid_2 \rangle, \langle gid_1, gid_2, gid_3 \rangle, \ldots\} \cup \lambda$ with $gid_i \in Dom_{gid}$ is a tuple containing group identifiers (or null), and pd is the payload of the data stream element.
Each data stream element is uniquely identifiable by the combination of its key k, its sequence number ξ, and the data stream ds it belongs to.

$$UID(\widehat{DSE}) = \{ds, k, \xi\}$$

The payload pd of a data stream element is restricted to contain neither any position nor any affiliation information (i.e., object or group identifiers) and to be structured according to the schema ($dse.ds.sch$) of the data stream it belongs to.

Raw input values, such as the heart rate in case of a player sensor measurement, event-specific information, such as the velocity in case of a successful pass event, state data, such as the surface of the minimum bounding rectangle in case of a team area state, and statistical values, such as the average heart rate in case of a fitness statistic, are data stream specific pieces of information. As indicated above, these data stream specific pieces of information are encoded in the payload (pd) of the data stream element which is structured according to the schema of the data stream the element belongs to ($dse.ds.sch$).

In contrast, the time of generation, i.e., the moment when a value was measured by a sensor, when an atomic event occurred, when a non-atomic event was updated, when a state was calculated, or when a statistic was generated, is encoded as a natural number in form of an explicit timestamp (ts). As we will present in more detail in Section 6.1.1, the timestamps of all data stream elements specify the moment of generation with respect to one globally consistent time space. Depending on the application the unit of the timestamp may be seconds, milliseconds, nanoseconds, or even an application-specific unit. In our real-time football analysis application we use milliseconds (see Chapter 9).

Moreover, positions are given only in the dedicated positions tuple (*pos*). Each element of the positions tuple is a three dimensional position. The number of positions in the tuple can be arbitrarily chosen (incl. zero). For instance, elements of the player sensor input stream comprise the location of the player as the only position while elements of a successful pass event stream contain two positions – the start and the end of the pass. Due to the unambiguous ordering of the elements of a tuple, the positions in the positions tuple can be assigned semantics. For instance, one can define that in a successful pass event stream element the first element of the positions tuple is the start position and the second element of the positions tuple is the end position of the pass. If no positions are available, the positions tuple is set to null (i.e., $pos = \lambda$).

Similarly, affiliation information is given only in the dedicated object identifiers tuple (*oids*) and in the dedicated group identifiers tuple (*gids*) which comprise an arbitrary number (incl. zero) of object and group identifiers, respectively. The domain for the object identifiers and group identifiers can be chosen arbitrarily. Every object identifier uniquely identifies an object and every group identifier uniquely identifies a group. Hence, a data stream element can be assigned to an arbitrary number of objects and groups. In consequence, an event stream element can represent an event which is either anonymous, i.e., neither assigned to an object nor to a group, individual, i.e., assigned to a single object, or collaborative, i.e., assigned to multiple objects or even a single or multiple groups. Similarly, a raw input stream element, a state stream element, and a statistics stream element can contain anonymous, individual, or collaborative input, state, and statistics data, respectively. The semantics of the term object and group is highly scenario-specific. For instance, in the real-time football analysis scenario an object is a moving object on the field, i.e., a player or the ball, and the group is a team (see Section 2.1). In contrast, in a disaster management scenario an object might be a firefighter and the group might be a squad. As for the positions tuple, the positions of elements in the object identifiers and group identifiers tuple can be assigned semantics such as that in a successful pass event stream element the first object identifier identifies the player who shot the ball and the second object identifier identifies the player who received the ball. If no object and group is available, the object identifiers and group identifiers tuple is set to null (i.e., $oids = \lambda$ and $gids = \lambda$), respectively.

Elements of the same data stream typically contain the same number of positions, object identifiers, and group identifiers. For instance, successful pass event stream elements always contain two positions (the start and the end of the pass), two object identifiers (identifying the player who shot and the player who received the ball), and a single group identifier (identifying the team both players belong to). However, this is not required. For instance, fitness statistics stream elements which contain statistics of a player contain an object identifier while fitness statistics stream elements which contain statistics of a whole team do not. Moreover, elements of the offside line state stream comprise the positions and identifiers of all players who are currently in offside position. As a matter of course, the number of players in offside position is not constant but changes during the match. In consequence, the number of positions and object identifiers in the offside line state stream elements varies.

There are four more attributes contained in each data stream element which we have not taken a look at yet. Namely, each data stream element further contains a key (k), a sequence number (ξ), an event identifier (eid), and a phase (φ). The key is used to split data streams into partitions for parallelization purposes and the sequence number is used to order the elements belonging to each partition. More details about data stream partitions will be given in Section 4.3. The purpose of the event identifier and the phase is to group those elements of a non-atomic data stream which belong to the same non-atomic event. More details on how these attributes have to be set to achieve this goal and how they are set in atomic data stream elements will be given in Section 4.4.

Finally, we define data stream elements to be immutable. That is, once a data stream element is generated, its attributes, such as the timestamp, the key, the phase, and the positions tuple, cannot be modified. This is in line with the fundamental concepts of Samza [Sam17d]. The sole exception to this is that a data stream element might be assigned its sequence number not by the component which has generated it but by a proxy (see Section 6.2.1.1 for more details). However, this is done immediately after generating the data stream element, i.e., before any other component receives the element, and once the element is assigned a sequence number, also this sequence number cannot by modified anymore. Consequently, if any component of the data stream analysis system intends to modify an attribute of a data stream element (e.g., to adapt the identifier of a group), it cannot actually modify the attribute of the existing element but instead has to create a new element. This also includes modifications of the key for repartitioning purposes (see Section 5.3.1).

4.3 Data Stream Partitions

As already indicated in Section 4.2, every data stream element contains a key (k). This key is used to split data streams into multiple data stream partitions [RM19]. Each data stream partition (dsp) comprises all data stream elements belonging to a certain data stream and having a certain key. That is, the partition of data stream ds for key k comprises all data stream elements belonging to ds and having key k.

Let $\widehat{DSP}$ be the global set of all partitions of all data streams. We formally define a data stream partition as follows:

Definition 4.3 Data Stream Partition

A data stream partition dsp is a pair $\langle ds, k\rangle$ where $ds \in \widehat{DS}$ is the data stream of which dsp is a partition and $k \in Dom_k$ is the key of the partition. The combination of both attributes uniquely identifies a data stream partition.

$$UID(\widehat{DSP}) = \{ds, k\}$$

$DSE(ds, k, \xi)$ contains all elements of dsp, i.e., all data stream elements belonging to ds and having key k, up to a given sequence number $\xi \in \mathbb{N}_0$.

$$DSE(ds, k, \xi) = \left\{dse \;\middle|\; dse \in \widehat{DSE} \wedge dse.ds = ds \wedge dse.k = k \wedge dse.\xi \leq \xi\right\}$$

Partitioning data streams by means of the key enables analyzing data stream elements belonging to different partitions in parallel and thus data-parallel analysis [HSS+14]. For instance, our real-time football analysis application supports analyzing multiple matches separately in parallel as we set the key of every data stream element to be the identifier of the match it belongs to (see Chapter 9).

Data stream elements which belong to the same partition are ordered by means of the sequence number (ξ) contained in every data stream element. That is, at every moment in time the set of data stream elements belonging to the same partition (i.e., $DSE(ds, k, \xi)$ where the combination of ds and k identifies the partition and ξ implicitly defines a moment in time) is totally ordered. This results in the fact that every data stream element is uniquely identifiable by the combination of the data stream partition dsp it is part of, i.e., by the data stream ds it belongs to and by the key k it has, and its sequence number ξ, as defined in Definition 4.2. However, note that elements of different data stream partitions cannot be ordered by means of the sequence number.

Literature Discussion 4.1 Key-Based Partitioning

We want to highlight at this point, that this part of our model, i.e., the key-based partitioning and the sequence number ordering, has been designed along the fundamental concepts of Samza described in [Sam17d]. More details on key-based data-parallel data stream analysis and on the ordering which is introduced by the sequence numbers will be given in Section 5.3 and Section 6.2.1.

We want to emphasize that the purpose of the sequence numbers leads to the fact that a sequence number can never by copied from an original data stream element, even if a one-to-one copy of the original element should be created and emitted in another data stream. Instead a new sequence number has to be created and assigned. This is indispensable to guarantee the total ordering property since other components might also emit elements in this data stream.

4.4 Atomicity

As we will describe in more detail in Section 4.5, raw input data, state data, statistics data, and the data of atomic events are assignable to one moment in time and shipped as a whole in a single data stream element of an atomic data stream. For instance, a player sensor measurement is taken and a team area state or a fitness statistic is calculated at a specific moment in time and shipped in a single data stream element. Moreover, a penalty box entry event is detected and assigned to the moment when a player enters the penalty box and the corresponding penalty box entry event stream element contains all data to represent this event. However, there are also non-atomic events, i.e., events which span a longer duration and for which information updates can be generated while they take place. As there are consumers that require the changing information in real-time (e.g., for real-time visualization purposes), it is not sufficient to ship all data in a single element of an atomic data stream when the event has finished. Instead, for these events updates are shipped as elements of a non-atomic data stream. For instance, a duel event, i.e., a situation during which two players are fighting for the ball, spans a certain amount of time – from the start to the end of the duel. To provide consumers with real-time updates during a duel, a duel event is represented by multiple duel event stream elements which do not contain all data of a duel event but only the latest positions of the players participating in the duel event and the latest distance between these players.

In order to enable grouping all data stream elements of a non-atomic event (e.g., of a specific duel) we define all elements of a non-atomic data stream which transport updates of the same non-atomic event to be emitted in the same data stream partition and to contain the same event identifier (*eid*) that identifies the non-atomic event. Moreover, we define each of these data stream elements to have a phase (φ) that explicitly marks if a data stream element is the first, the last, or any other data stream element of a non-atomic event. If a data stream element belongs to an atomic data stream, i.e., to a data stream that transports raw input data, state data, statistics data, or atomic event data, the identifier and the phase are set to null. Formally, we define these additional atomicity constraints as follows:

Definition 4.4 Atomicty Constraints

The event identifier *eid* and the phase φ of every data stream element *dse* have to fulfill the following constraints:

1. If the data stream element is atomic, i.e., $dse.ds.ato = \top$, the phase and the event identifier of the element have to be set to null.

$$dse.eid = \lambda \wedge dse.\varphi = \lambda$$

2. If the data stream element is non-atomic, i.e., $dse.ds.ato = \perp$, the event identifier of the data stream element has to be specified, all other data stream elements with the same event identifier have to belong to the same data stream partition, and the phase of the data stream element has to be set as follows:

$$dse.eid \neq \lambda \wedge$$
$$\left(\nexists dse_2 \in \widehat{DSE} : dse.eid = dse_2.eid \wedge (dse.ds \neq dse_2.ds \vee dse.k \neq dse_2.k)\right) \wedge$$
$$dse.\varphi = \begin{cases} \text{“start”} & \text{if } dse \text{ is the first element for the event identified} \\ & \text{by } dse.eid\text{, i.e., } \nexists dse_2 \in DSE_{dse.eid}(dse.ds, dse.k, dse.\xi) \\ & \text{with } dse_2 \neq dse \\ \text{“end”} & \text{if } dse \text{ is the last element for the event identified} \\ & \text{by } dse.eid\text{, i.e., } \nexists dse_2 \in DSE_{dse.eid}(dse.ds, dse.k, \xi) \\ & \text{with } dse_2.\xi > dse.\xi \text{ and for any } \xi \in \mathbb{N}_0 \\ \text{“active”} & \text{otherwise} \end{cases}$$

where $DSE_{eid}(ds, k, \xi) = \{dse \mid dse \in DSE(ds, k, \xi) \wedge dse.eid = eid\}$

Note that the phase value constraint given in Definition 4.4 further induces that the data of every non-atomic event are split at least into two data stream elements – a start and an end element – as a data stream element cannot be the first and the last data stream element for a non-atomic event since this would require its phase to be set to “start” and “end” at the same time. In the remainder of this thesis, we assume that all atomicity constraints are fulfilled by every data stream element, unless otherwise specified.

Finally, it is important to note that some events that seem to be non-atomic at the first glance are actually atomic. This is due to the fact that for an event to be non-atomic it has to be detectable from an early stage on. For instance, a human perceives a successful pass to start when a player kicks the ball and to end when another player receives the ball. Hence, from a human perspective the pass may span several seconds and continuous updates about the pass may seem beneficial. However, it is impossible to emit updates of an ongoing successful pass event (e.g., the positions of both players and the ball) in elements of a non-atomic data stream since it cannot be known for sure that the current ball movement is a successful pass until the second player receives the ball and thus the pass has finished. If we did so and a pass which seemed to be promising in the beginning is intercepted by a player of the other team, the whole event would be an interception event instead of a successful pass event and thus elements of a non-atomic successful pass event stream containing successful pass event updates would be faulty. As our model does not support probabilistic events [ASA+17], the only option is to handle such events as atomic events and thus to ship all information about such an event in a single element of an atomic data stream when it is reliably detected. Hence, a successful pass event is atomic and all data to represent a successful pass event are emitted as a whole in a single successful pass event stream element when the successful pass is detected, i.e., when the second player has received the ball.

4.5 Categories

As mentioned in Section 4.1, the category attribute (*cat*) of the data stream is used to classify data streams into four main categories: Raw input streams, event streams, state streams, and statistics streams. In this section, we will define these categories in more detail.

Elements of *raw input streams*, i.e., data streams whose category is “rawInput”, embody different forms of raw input data for data stream analysis systems. That is, in contrast to the elements of event, state, and statistics streams, raw input stream elements are not generated by components of a data stream analysis system but originate from external devices (see Section 5.1). Typically, these external devices are physical or virtual sensors. One example for a physical sensor is a heart rate sensor that periodically measures the current heart rate of a person. Another example for a physical sensor is a position sensor (e.g., a GPS sensor[1]) that periodically generates position updates. Both physical sensors can also be combined to a player sensor device (e.g., a fitness bracelet) that periodically measures the heart rate and position of a football player and emits these values in a joint data stream element as raw input for a football analysis system. Examples for raw input data emitted by virtual sensors are tweets [Twi20] or stock price updates but also position updates which are generated by a video-based tracking system. Although a raw input stream element typically refers to a single object (as in the examples listed above) or to no object at all (e.g., if it contains the current rainfall quantity in a football stadium), a raw input stream element can also refer to a set of objects (e.g., if it contains the identifiers of the two players participating in a substitution) or even a set of groups. In theory, the forms of and sources for raw input data are unlimited. In our model, we shrink the potential raw input stream space only by adding a single restriction. That is, we require raw input data to be shipped as a whole in a single raw input stream element and thus every raw input stream to be atomic.

Elements of *event streams*, i.e., data streams whose category is "event", represent a certain event, such as a successful pass or a duel, which was detected by a data stream analysis system. An element of an atomic event stream ($ds.ato = \top$) contains all information of an atomic event (e.g., a successful pass event). In contrast, an element of a non-atomic event stream ($ds.ato = \bot$) comprises updates of a non-atomic event (e.g., a duel event).

Elements of *state streams*, i.e., data streams whose category is "state", contain a state, such as the current surfaces of the areas spanned by the players of a team or the current pressing intensity on the player in ball possession, which was calculated by a data stream analysis system. A state stream element can contain multiple state values if they correspond to the same spatial and temporal information and if they are assigned to the same objects and groups. For instance, a team area state stream element can contain the surface of the minimum bounding rectangle and the planar convex hull spanned by the players of a team. Since all states are calculated with respect to a specific point in time

[1] Note that geo-coordinates $\langle long, lat \rangle$ can be projected to three dimensional positions if necessary.

and can thus be emitted as a whole in a single state stream element, we demand every state stream to be atomic.

Elements of *statistics streams*, i.e., data streams whose category is “statistics”, comprise statistical values, such as the average heart rate measured by a player sensor or the pass success rate of a player or team, which were generated by a data stream analysis system. As a state stream element, a statistics stream element can contain multiple statistical values if they correspond to the same spatial and temporal information and if they are assigned to the same objects and groups. Moreover, also statistics are calculated with respect to a specific point in time and can thus be emitted as a whole in a single statistics stream element. Hence, we demand every statistics stream to be atomic.

Although the state and the statistics category might seem exchangeable at the first glance, there is a subtle but very important semantic difference between state stream elements and statistics stream elements. A state stream element contains information which describes the current situation or more precisely the situation at the moment the timestamp of the state stream element specifies. In contrast, a statistics stream element contains statistical values which are aggregates (sum, average, maximum, etc.) over a certain time interval. This time interval ends at the moment the timestamp of the statistics stream element specifies. The start of the interval can be defined implicitly (e.g., a pass statistics stream element is simply assumed to contain the statistic for the full match up to its timestamp) or explicitly (e.g., a fitness statistics stream element contains the interval length in its payload which in combination with the timestamp specifies the start of the interval).

Based on the restrictions we posed above for the different data stream categories, we formally define a *well-formed data stream* as follows:

Definition 4.5 Well-Formed Data Stream

Raw input streams, state streams, and statistics streams are well-formed if they are atomic. Event streams are always well-formed as they can be atomic or non-atomic. Consequently, a data stream *ds* is well-formed if the following formula is true:

$$ds.cat = \text{“event”} \lor \left(ds.cat \in \left\{\text{“rawInput”}, \text{“state”}, \text{“statistics”}\right\} \land ds.ato = \top\right)$$

Data streams which are not well-formed are called ill-formed.

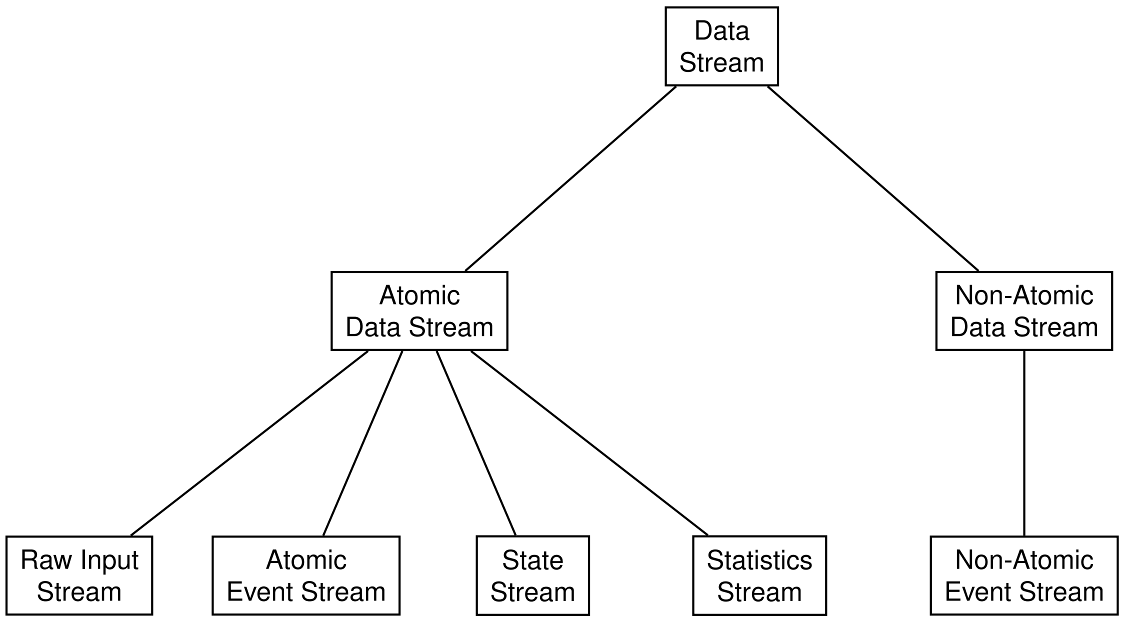


Figure 4.1 Data Stream Class Hierarchy. The tree illustrates the data stream class hierarchy which results from our well-formed data stream definition.

In our work, we assume that every data stream that is consumed or produced by a data stream analysis system fulfills the constraint given in Definition 4.5 and is thus well-formed. In consequence, we assume that all raw input streams, state streams, and statistics streams are atomic. Hence, elements of these streams contain the whole data of a raw input, a state, or a statistic. In contrast, event streams can be atomic or non-atomic depending on the fact if the events their elements are supposed to represent are represented by a single data stream element and thus atomic or represented by multiple data stream elements and thus non-atomic. Figure 4.1 depicts this data stream class hierarchy as a tree.

4.6 Examples

We have used some running examples in the previous sections to motivate and explain our data model. In this section we will go even one step further and present five detailed examples with explicit sample tuples in order to illustrate our data model in general and the atomicity and category semantics in particular. More precisely, we will present a sample raw input stream, a sample event stream whose elements represent atomic events, a sample event stream whose elements are updates of non-atomic events, a sample state stream, and a sample statistics stream from the football analysis scenario (see Section 2.1).

Example 4.1 Player Sensor Input Stream

As an example for a raw input, assume that each player of a football match is equipped with a sensor device that periodically (every 100 milliseconds) measures the current position and heart rate of a player as raw input data for a real-time football analysis system. Further, assume *ds* to be the data stream containing these input data.

$$ds = \left\langle \text{“playerSensorInput”}, \text{“rawInput”}, \top, \text{“heartRate”} \right\rangle$$

The name of the data stream is “playerSensorInput”. Thus, we denote the data stream as player sensor input stream and elements of this data stream as player sensor input stream elements. The data stream defines that every player sensor input stream element contains a heart rate value encoded as a String. Note that the data stream is further defined to be atomic and thus enforces that player sensor input stream elements contain neither an event identifier nor a phase. Assume dse_1 to dse_8 to be sample player sensor input stream elements.

$$dse_1 = \left\langle ds, \text{“X”}, \lambda, \lambda, 1, 7, \langle\langle 17.381, -2.402, 0\rangle\rangle, \langle\text{“A4”}\rangle, \langle\text{“A”}\rangle, \text{“100.72”} \right\rangle$$
$$dse_2 = \left\langle ds, \text{“X”}, \lambda, \lambda, 2, 32, \langle\langle 1.854, -15.178, 0\rangle\rangle, \langle\text{“B7”}\rangle, \langle\text{“B”}\rangle, \text{“93.45”} \right\rangle$$
$$dse_3 = \left\langle ds, \text{“X”}, \lambda, \lambda, 3, 32, \langle\langle 12.462, 5.975, 0\rangle\rangle, \langle\text{“A1”}\rangle, \langle\text{“A”}\rangle, \text{“112.83”} \right\rangle$$
$$dse_4 = \left\langle ds, \text{“Y”}, \lambda, \lambda, 1, 59, \langle\langle -3.821, 9.482, 0\rangle\rangle, \langle\text{“A2”}\rangle, \langle\text{“A”}\rangle, \text{“105.23”} \right\rangle$$
$$dse_5 = \left\langle ds, \text{“X”}, \lambda, \lambda, 4, 107, \langle\langle 17.213, -2.456, 0\rangle\rangle, \langle\text{“A4”}\rangle, \langle\text{“A”}\rangle, \text{“100.37”} \right\rangle$$
$$dse_6 = \left\langle ds, \text{“X”}, \lambda, \lambda, 5, 132, \langle\langle 1.789, -15.203, 0\rangle\rangle, \langle\text{“B7”}\rangle, \langle\text{“B”}\rangle, \text{“92.23”} \right\rangle$$
$$dse_7 = \left\langle ds, \text{“X”}, \lambda, \lambda, 6, 132, \langle\langle 12.638, 5.821, 0\rangle\rangle, \langle\text{“A1”}\rangle, \langle\text{“A”}\rangle, \text{“106.14”} \right\rangle$$
$$dse_8 = \left\langle ds, \text{“Y”}, \lambda, \lambda, 2, 159, \langle\langle -3.898, 9.521, 0\rangle\rangle, \langle\text{“A2”}\rangle, \langle\text{“A”}\rangle, \text{“113.74”} \right\rangle$$

The timestamps are the measurement time of the sensor and given in milliseconds. In order to enable analyzing multiple matches in parallel, the key of every data stream element is set to be the match identifier. The partitions for both matches are totally ordered by means of the sequence numbers. The keys, object identifiers and group identifiers show that dse_1 and dse_5 are measurements in match X for player A4 belonging to team A, dse_2 and dse_6 are measurements in match X for player B7 belonging to team B, dse_3 and dse_7 are measurements in match X for player A1 belonging to team A, and dse_4 and dse_8 are measurements in match Y for player A2 belonging to team A. As the constraint given in Definition 4.5 is fulfilled ($ato = \top$), *ds* is a well-formed raw input stream.

Example 4.2 Successful Pass Event Stream

As an example for an atomic event, assume that there is a real-time football analysis system which detects successful pass events and that *ds* is the data stream for shipping the successful pass event data.

$$ds = \left\langle \text{“successfulPassEvent”}, \text{“event”}, \top, \text{“length;velocity;angle”} \right\rangle$$

Since the name of the data stream is “successfulPassEvent”, we denote the data stream as successful pass event stream and elements of this data stream as successful pass event stream elements. The data stream defines every successful pass event stream element to contain the length, velocity and angle of the pass encoded as a semicolon-separated String. Note that the data stream is further defined to be atomic and thus enforces that successful pass event stream elements contain neither an event identifier nor a phase. Assume dse_1, dse_2 and dse_3 to be sample successful pass event stream elements.

$$\begin{aligned}
dse_1 = \Big\langle & ds, \text{“X”}, \lambda, \lambda, 1, 34782, \langle\langle -9.27, -17.614, 0\rangle, \langle -17.346, -14.95, 0\rangle\rangle, \\
& \langle \text{“A6”}, \text{“A2”}\rangle, \langle \text{“A”}\rangle, \text{“8.504;10.172;341.744”} \Big\rangle \\
dse_2 = \Big\langle & ds, \text{“Y”}, \lambda, \lambda, 1, 42195, \langle\langle 44.089, -31.989, 0\rangle, \langle 21.61, -29.83, 0\rangle\rangle, \\
& \langle \text{“A7”}, \text{“A6”}\rangle, \langle \text{“A”}\rangle, \text{“22.582;20.127;354.514”} \Big\rangle \\
dse_3 = \Big\langle & ds, \text{“X”}, \lambda, \lambda, 2, 42873, \langle\langle 21.582, -28.411, 0\rangle, \langle 12.981, -10.653, 0\rangle\rangle, \\
& \langle \text{“B4”}, \text{“B3”}\rangle, \langle \text{“B”}\rangle, \text{“19.731;2.865;115.843”} \Big\rangle
\end{aligned}$$

The timestamp of each successful pass event stream element is the largest timestamp of all data stream elements which have contributed to the successful pass event and thus the generation time of the event. Again timestamps are given in milliseconds, the key of every data stream element is set to be the match identifier, and the elements of all resulting partitions are totally ordered by means of the sequence numbers. dse_1 represents a pass in match X from player A6 at $\langle -9.27, -17.614, 0\rangle$ to player A2 at $\langle -17.346, -14.95, 0\rangle$, both belonging to team A. dse_2 represents a pass in match Y from player A7 at $\langle 44.089, -31.989, 0\rangle$ to player A6 at $\langle 21.61, -29.83, 0\rangle$, both belonging to team A. dse_3 represents a pass in match X from player B4 at $\langle 21.582, -28.411, 0\rangle$ to player B3 at $\langle 12.981, -10.653, 0\rangle$, both belonging to team B. Since *ds* is an event stream, it is inherently well-formed.

Example 4.3 Duel Event Stream

As an example for a non-atomic event, assume that there is a real-time football analysis system which detects duel events and that *ds* is the data stream for shipping the duel event data.

$$ds = \langle\text{“duelEvent”}, \text{“event”}, \perp, \text{“distance”}\rangle$$

Since the name of the data stream is “duelEvent”, we denote the data stream as duel event stream and elements of this data stream as duel event stream elements. The data stream defines every duel event stream element to contain the distance between the players encoded as a String. Moreover, as the data stream is non-atomic, every duel event stream element has to contain an event identifier (unique identifier of the duel event) and a phase. Assume dse_1 to dse_9 to be sample duel event stream elements.

$$\begin{aligned}
dse_1 = \Big\langle & ds, \text{“X”}, 1, \text{“start”}, 1, 23862, \langle\langle 10.35, -5.3, 0\rangle, \langle 10.55, -5.5, 0\rangle\rangle, \\
& \langle\text{“A1”}, \text{“B3”}\rangle, \langle\text{“A”}, \text{“B”}\rangle, \text{“0.283”}\Big\rangle \\
dse_2 = \Big\langle & ds, \text{“X”}, 1, \text{“active”}, 2, 23962, \langle\langle 10.65, -5.74, 0\rangle, \langle 10.87, -5.54, 0\rangle\rangle, \\
& \langle\text{“A1”}, \text{“B3”}\rangle, \langle\text{“A”}, \text{“B”}\rangle, \text{“0.297”}\Big\rangle \\
dse_3 = \Big\langle & ds, \text{“X”}, 1, \text{“active”}, 3, 24062, \langle\langle 10.87, -5.92, 0\rangle, \langle 10.69, -6.08, 0\rangle\rangle, \\
& \langle\text{“A1”}, \text{“B3”}\rangle, \langle\text{“A”}, \text{“B”}\rangle, \text{“0.241”}\Big\rangle \\
dse_4 = \Big\langle & ds, \text{“X”}, 1, \text{“active”}, 4, 24162, \langle\langle 11.32, -5.84, 0\rangle, \langle 11.18, -5.97, 0\rangle\rangle, \\
& \langle\text{“A1”}, \text{“B3”}\rangle, \langle\text{“A”}, \text{“B”}\rangle, \text{“0.191”}\Big\rangle \\
dse_5 = \Big\langle & ds, \text{“X”}, 1, \text{“end”}, 5, 24262, \langle\langle 11.38, -5.89, 0\rangle, \langle 11.16, -5.95, 0\rangle\rangle, \\
& \langle\text{“A1”}, \text{“B3”}\rangle, \langle\text{“A”}, \text{“B”}\rangle, \text{“0.228”}\Big\rangle \\
dse_6 = \Big\langle & ds, \text{“X”}, 2, \text{“start”}, 6, 78396, \langle\langle -20.32, 10.78, 0\rangle, \langle -20.58, 10.98, 0\rangle\rangle, \\
& \langle\text{“B5”}, \text{“A7”}\rangle, \langle\text{“B”}, \text{“A”}\rangle, \text{“0.328”}\Big\rangle \\
dse_7 = \Big\langle & ds, \text{“X”}, 2, \text{“active”}, 7, 78496, \langle\langle -19.89, 10.58, 0\rangle, \langle -20.17, 10.67, 0\rangle\rangle, \\
& \langle\text{“B5”}, \text{“A7”}\rangle, \langle\text{“B”}, \text{“A”}\rangle, \text{“0.294”}\Big\rangle \\
dse_8 = \Big\langle & ds, \text{“X”}, 2, \text{“active”}, 8, 78596, \langle\langle -20.03, 10.33, 0\rangle, \langle -19.86, 10.54, 0\rangle\rangle, \\
& \langle\text{“B5”}, \text{“A7”}\rangle, \langle\text{“B”}, \text{“A”}\rangle, \text{“0.270”}\Big\rangle
\end{aligned}$$

$$dse_9 = \Big\langle ds, \text{“X”}, 2, \text{“end”}, 9, 78696, \langle\langle -20.18, 10.07, 0\rangle, \langle -20.02, 10.23, 0\rangle\rangle, \\ \langle \text{“B5”}, \text{“A7”}\rangle, \langle \text{“B”}, \text{“A”}\rangle, \text{“0.226”}\Big\rangle$$

The timestamp of each duel event stream element is the largest timestamp of all data stream elements which have contributed to the duel event update and thus the generation time of the event update. Again timestamps are given in milliseconds, the key of every data stream element is set to be the match identifier, and the elements of all resulting partitions are totally ordered by means of the sequence numbers. dse_1 to dse_5 represent a duel event in match X between the defending player A1 and the attacking player B3, and dse_6 to dse_9 represent a duel event in match X between the defending player B5 and the attacking player A7. Every data stream element updates the position of both players participating in the duel as well as the distance between these players. Since ds is an event stream, it is inherently well-formed.

Example 4.4 Team Area State Stream

As an example for a state, assume that there is a real-time football analysis system which periodically (every 100 milliseconds) calculates the surface of the minimum bounding rectangle and the planar convex hull (see Section 7.2) spanned by the players of the two teams of a football match. Further, assume ds to be the data stream containing these input data.

$$ds = \big\langle \text{“teamAreaState”}, \text{“state”}, \top, \text{“mbrSurface;pchSurface”}\big\rangle$$

The name of the data stream is “teamAreaState”. Thus, we denote the data stream as team area state stream and elements of this data stream as team area state stream elements. The data stream defines that every team area state stream element contains the surface of the minimum bounding rectangle and the surface of the planar convex hull encoded as a semicolon-separated String. Note that the data stream is further defined to be atomic and thus enforces that team area state stream elements contain neither an event identifier nor a phase. Assume dse_1 to dse_6 to be sample team area state stream elements.

$$dse_1 = \Big\langle ds, \text{“X”}, \lambda, \lambda, 1, 32, \lambda, \lambda, \langle \text{“A”}\rangle, \text{“2837.28;1668.35”}\Big\rangle$$

$$dse_2 = \Big\langle ds, \text{“X”}, \lambda, \lambda, 2, 47, \lambda, \lambda, \langle \text{“B”}\rangle, \text{“1925.69;1354.32”}\Big\rangle$$

$$dse_3 = \Big\langle ds, \text{“Y”}, \lambda, \lambda, 1, 59, \lambda, \lambda, \langle \text{“A”}\rangle, \text{“2167.93;1445.46”}\Big\rangle$$

$$dse_4 = \Big\langle ds, \text{“X”}, \lambda, \lambda, 3, 132, \lambda, \lambda, \langle\text{“A”}\rangle, \text{“2839.76;1668.48”}\Big\rangle$$
$$dse_5 = \Big\langle ds, \text{“X”}, \lambda, \lambda, 4, 147, \lambda, \lambda, \langle\text{“B”}\rangle, \text{“1922.14;1352.79”}\Big\rangle$$
$$dse_6 = \Big\langle ds, \text{“Y”}, \lambda, \lambda, 2, 159, \lambda, \lambda, \langle\text{“A”}\rangle, \text{“2166.34;1445.98”}\Big\rangle$$

The timestamp of each team area state stream element is the largest timestamp of all data stream elements which have contributed to the team area state and thus the generation time of the state. In order to enable analyzing multiple matches in parallel, the key of every data stream element is set to be the match identifier. The partitions for both matches are totally ordered by means of the sequence numbers. dse_1 and dse_4 contain the surface of the minimum bounding rectangle (2837.28 m^2 and 2839.76 m^2) and the planar convex hull (1668.35 m^2 and 1668.48 m^2) around the players of team A in match X 32 and 132 milliseconds after match X has started. dse_2 and dse_5 contain the same for team B 47 and 147 milliseconds after match X has started. dse_3 and dse_6 contain the same for team A in match Y 59 and 159 after match Y has started. As the constraint given in Definition 4.5 is fulfilled ($ato = \top$), ds is a well-formed state stream.

Example 4.5 Fitness Statistics Stream

As a statistics example, assume that there is a real-time football analysis system which calculates the average heart rate measured by each player sensor and the average velocity for each player (derived by means of the positions measured by the player sensors) for two and five minute intervals. Moreover, assume that the real-time football analysis system calculates aggregated statistics for the teams. Further, assume ds to be the data stream for shipping these statistics.

$$ds = \big\langle \text{“fitnessStatistics”}, \text{“statistics”}, \top, \text{“intervalInS;avgHeartRate;avgVelocity”}\big\rangle$$

The average heart rate and the average velocity are jointly handled in the same data stream with the name “fitnessStatistics”. We denote this data stream as fitness statistics stream and elements of this data stream as fitness statistics stream elements. The data stream defines that every fitness statistics stream element contains the interval time in seconds as well as the average heart rate and the average velocity of a player (or team) encoded as a semicolon-separated String. Note that the data stream is further defined as atomic and thus enforces that fitness statistics stream elements contain neither an event identifier nor a phase. Assume dse_1 to dse_{12} to be sample fitness statistics stream elements.

$$
\begin{aligned}
dse_1 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 1, 119907, \lambda, \langle\text{“A4”}\rangle, \langle\text{“A”}\rangle, \text{“120;100.52;11.43”}\right\rangle\\
dse_2 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 2, 119932, \lambda, \langle\text{“B7”}\rangle, \langle\text{“B”}\rangle, \text{“120;93.12;8.49”}\right\rangle\\
dse_3 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 3, 119932, \lambda, \lambda, \langle\text{“B”}\rangle, \text{“120;95.37;10.21”}\right\rangle\\
dse_4 &= \left\langle ds, \text{“Y”}, \lambda, \lambda, 1, 119959, \lambda, \langle\text{“A2”}\rangle, \langle\text{“A”}\rangle, \text{“120;113.24;14.25”}\right\rangle\\
dse_5 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 4, 239907, \lambda, \langle\text{“A4”}\rangle, \langle\text{“A”}\rangle, \text{“120;105.39;12.42”}\right\rangle\\
dse_6 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 5, 239932, \lambda, \langle\text{“B7”}\rangle, \langle\text{“B”}\rangle, \text{“120;97.56;9.32”}\right\rangle\\
dse_7 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 6, 239932, \lambda, \lambda, \langle\text{“B”}\rangle, \text{“120;103.76;11.87”}\right\rangle\\
dse_8 &= \left\langle ds, \text{“Y”}, \lambda, \lambda, 2, 239959, \lambda, \langle\text{“A2”}\rangle, \langle\text{“A”}\rangle, \text{“120;108.67;10.43”}\right\rangle\\
dse_9 &= \left\langle ds, \text{“X”}, \lambda, \lambda, 7, 299907, \lambda, \langle\text{“A4”}\rangle, \langle\text{“A”}\rangle, \text{“300;104.28;11.83”}\right\rangle\\
dse_{10} &= \left\langle ds, \text{“X”}, \lambda, \lambda, 8, 299932, \lambda, \langle\text{“B7”}\rangle, \langle\text{“B”}\rangle, \text{“300;96.23;9.04”}\right\rangle\\
dse_{11} &= \left\langle ds, \text{“X”}, \lambda, \lambda, 9, 299932, \lambda, \lambda, \langle\text{“B”}\rangle, \text{“300;99.93;11.14”}\right\rangle\\
dse_{12} &= \left\langle ds, \text{“Y”}, \lambda, \lambda, 3, 299959, \lambda, \langle\text{“A2”}\rangle, \langle\text{“A”}\rangle, \text{“300;110.52;12.31”}\right\rangle
\end{aligned}
$$

The timestamp of each fitness statistics stream element is the largest timestamp of all data stream elements which have contributed to the fitness statistic and thus the generation time of the statistic. Again timestamps are given in milliseconds, the key of every data stream element is set to be the match identifier, and the elements of all resulting partitions are totally ordered by means of the sequence numbers. dse_1, dse_5, and dse_9 contain the average heart rate and velocity of player A4 belonging to team A in match X for the first two minutes, the second two minutes, and the first five minutes, respectively. dse_2, dse_6, as well as dse_{10} comprise the same for player B7 belonging to team B in match X and dse_4, dse_8, as well as dse_{12} comprise the same for player A2 belonging to team A in match Y. dse_3, dse_7, and dse_{11} contain the aggregated average heart rate and velocity of all players of team B in match X for the first two minutes, the second two minutes, and the first five minutes, respectively. As the constraint given in Definition 4.5 is fulfilled ($ato = \top$), ds is a well-formed statistics stream.

5 System Model

In Chapter 4, we have regarded the data stream analysis system that consumes raw input stream elements, performs analyses on the basis of these elements, and produces event, state, and/or statistics stream elements as a black box as illustrated in Figure 3.1. However, as discussed in Section 3.1, there are many diverse data stream analysis systems.

For the reasons which we have presented in Section 3.1, we have decided to follow the worker-based data stream analysis system approach in our work. In consequence, modeling and discussing the implications of all different types of data stream analysis systems in detail is out of the scope of this thesis. Instead, we confine the focus of the remainder of this thesis on worker-based systems. More precisely, we restrict all discussions, such as the extensive stream time model discussion given in Chapter 6, on data stream analysis systems that are designed according to our model of a worker-based data stream analysis system which we have constructed to be as precise as necessary but as generic as possible. Consequently, we refer in the remainder of this thesis with the term data stream analysis system to an arbitrary data stream analysis system that is designed according to our system model, unless otherwise specified.

In this chapter, we will present our system model in five steps. First, we will continue regarding the data stream analysis system as a black box and model its sources. Second, we will start lighting this black box by presenting a simplified version of our worker-based data stream analysis system model which regards neither parallelism on a worker level nor the coexistence of other worker-based data stream analysis systems. In a third step, we will extend this model with the objective of achieving full parallelism by means of introducing data parallelism. Fourth, we will define the machine and network related aspects of our model. Finally, we will regard our system model from a global perspective by means of considering not only a single but multiple data stream analysis systems.

5.1 Raw Input Stream Generating Device Model

Before we model the data stream analysis system itself, we first model the devices which produce and emit the input data of a data stream analysis system. While doing so we continue regarding the data stream analysis system as a black box as illustrated in Figure 3.1. As presented in Section 4.5, raw input stream elements contain the raw input data produced by external devices, i.e., by devices which are not a component of the data stream analysis system.

Literature Discussion 5.1 Raw Input Stream Generating Device

Since the external devices are typically physical or virtual sensors, there is literature which refers to them as *sensor devices*, *sensor hardware*, or simply *sensors* [PGS16a; Bre08]. In other literature these devices are denoted as *input sources* or *data sources* [TTS$^+$14; NPP$^+$17]. Moreover, in literature that regards raw input to reflect events (see Section 6.1.1 for more details on this point of view) they are also referred to as *event producers* [Fli18].
Note that Storm [TTS$^+$14] and Samza [NPP$^+$17] regard the message queue or publish/subscribe system (e.g., Kafka [KNR11]) from which they pull the raw input stream elements as the input source while we regard the brokers of the message queue or publish/subscribe system as communication proxies (see Section 5.4.2 for more details) and the devices which feed these proxies as the actual sources. Moreover, note that Flink [Fli18; CKE$^+$15] is not a worker-based but a graph-based data stream analysis systems since the whole analysis task is implemented by means of concatenating higher-order functions (called operators in Flink) to a graph (see Section 3.1).

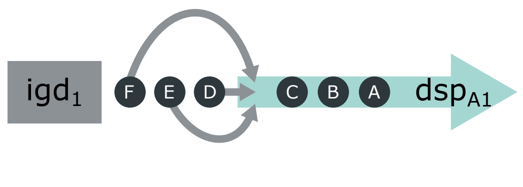
In our system model, we subsume all external devices which produce raw input data and emit these data in elements of raw input streams that are consumed by a data stream analysis system under the term *raw input stream generating devices*. We do so as we define raw input stream generating devices to emit only raw input stream elements but no event stream elements, state stream elements, or statistics stream elements. Moreover, we define raw input stream generating devices to be the only components which emit raw input stream elements. With this restriction we obtain a strict separation between the input data and the output data of a data stream analysis system: All input data are shipped by raw input stream generating devices in elements of raw input streams and all output data are emitted by components of the data stream analysis system (see below) in elements of event, state, and/or statistics streams.

Every raw input stream generating device (*igd*) is uniquely identified by a device identifier (*igdid*) and emits elements of a set of raw input stream partitions (*DSP*). Usually, a raw input stream generating device emits only elements of a single raw input stream partition (see Figure 5.1(a)). Even if the device comprises multiple sensors, the values they measure can be shipped in a single raw input stream element. For instance, a player sensor input stream element comprises the current position and heart rate of a player (see Example 4.1). However, a raw input stream generating device is also able to emit elements of multiple raw input streams (see Figure 5.1(b)). That is, the device attached to a player could emit the position and heart rate in separate player position stream and player heart rate stream elements if desired or necessary (e.g., due to different measurement frequencies). Moreover, a raw input stream generating device can emit elements of multiple partitions of the same raw input stream (see Figure 5.1(c)). For instance, a raw input stream generating device that embodies a virtual Twitter [Twi20] sensor might emit tweets in elements of multiple partitions of the same raw input stream if the key of every element is the identifier of the user who posted the contained tweet. In fact, there are no restrictions, but a raw input stream generating device can emit elements of arbitrary raw input stream partitions. That is, a raw input stream generating device is also able to emit elements with key k_1 and k_2 belonging to raw input stream ds_A and elements with key k_1 belonging to another raw input stream ds_B and thus to emit elements of three partitions of two raw input streams (see Figure 5.1(d)).

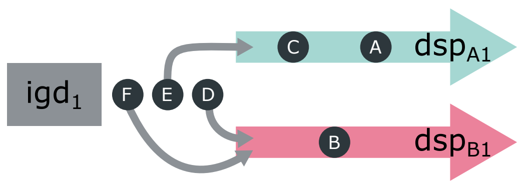
Let $\widehat{IGD}$ be the global set of all raw input stream generating devices and Dom_{igdid} be the raw input stream generating device identifier domain. We formally define a raw input stream generating device as follows:

Definition 5.1 Raw Input Stream Generating Device

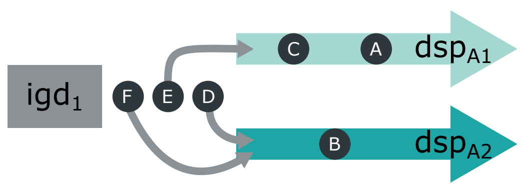
A raw input stream generating device igd is a pair $\langle igdid, DSP \rangle$ where $igdid \in Dom_{igdid}$ is a unique raw input stream generating device identifier (i.e., $UID(\widehat{IGD}) = \{igdid\}$) and $DSP \subseteq \{dsp \mid dsp \in \widehat{DSP} \wedge dsp.ds.cat = \text{"rawInput"}\}$ is the set of raw input stream partitions of which the raw input stream generating devices emits elements.



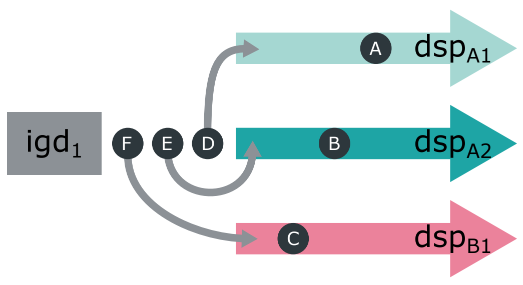
(a) Single Key and Single Stream



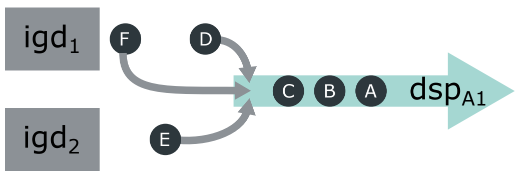
(b) Single Key and Multiple Streams



(c) Multiple Keys and Single Stream



(d) Multiple Keys and Multiple Streams



(e) Multiple Devices and Single Partition

Figure 5.1 Emission Options of Raw Input Stream Generating Devices. The gray boxes represent the raw input stream generating devices igd_1 and igd_2. The dark-gray dots represent the elements (dse_A, dse_B, dse_C, dse_D, dse_E, and dse_F) which igd_1 and igd_2 emit in various raw input stream partitions. The light-mint arrow visualizes the partition dsp_{A1} of ds_A for key k_1, the dark-mint arrow visualizes the partition dsp_{A2} of ds_A for key k_2, and the red arrow visualizes the partition dsp_{B1} of ds_B for key k_1.

Note that our model does not prohibit multiple raw input stream generating devices to emit elements of the same raw input stream partition as illustrated in Figure 5.1(e). For instance, all sensor devices attached to players of a football match can emit player sensor input stream elements whose key is set to the identifier of the match and which thus belong to the same player sensor input stream partition. In consequence, we do not assume that all elements of a raw input stream partition are produced at and emitted by the same raw input stream generating device.

5.2 Simplified Data Stream Analysis System Model

In this section, we will start lighting the black box illustrated in Figure 3.1 and thus give more details on how we model a worker-based data stream analysis system by presenting a first simplified model that will be further refined in the subsequent sections. The aim of our simplified model is to define the stepwise analysis performed by the data stream analysis system. In doing so the model gives a first conceptual overview of a worker-based data stream analysis system. However, note that this first model does not comprise all components of a worker-based data stream analysis system yet. This is due to the fact, that we omit full parallelization in the simplified version of our model for the sake of an easier understanding of the essentials of a worker-based data stream analysis system. For this purpose, we assume in this section that there are no data stream partitions but only data streams. This is equivalent to assuming that there is only a single key and thus also only a single partition for every data stream.

The fundamental idea of worker-based data stream analysis systems is to perform the analysis stepwise in a *workflow* consisting of multiple independent and freely-programmable *workers*. With this worker and workflow terminology we stay consistent with our previous work [Pro14; PGS16a; PGS16b; PBS+17; PRS+18; SRP+19]. In this section, we will first model the workers of a data stream analysis systems and then define how these workers form the workflow of the data stream analysis system.

Literature Discussion 5.2 Worker-Based Data Stream Analysis Systems

Before we start presenting our simplified system model, we want to point out explicitly that the idea of worker-based data stream analysis systems is not invented by us in the course of this thesis. Instead, the essentials of our system model base on the models of the popular worker-based data stream analysis systems Storm [TTS+14], Samza [Sam17d; NPP+17; KK15], and MillWheel [ABB+13] as well as on the models of previous worker-based data stream analysis systems developed in our research group, namely PAN [Pro14; PGS16a; PGS16b] and OSIRIS-SE [Bre08]. Especially Samza's concepts have had a substantial impact on our system model design as Samza is used as the foundation for our implementation (see Chapter 8). To facilitate aligning this thesis with other literature, we want to point out that other literature which introduces worker-based data stream analysis systems may use different terms,

such as *bolt* [TTS+14], *job* [Sam17d], or *operator* [Bre08] instead of worker and *topology* [TTS+14], *stream process* [Bre08], or *dataflow graph* [Sam17d] instead of workflow.
However, we have not merely adopted an existing system model. Instead, we have designed a system model which regards the different data stream categories of our data model and thus distinguishes between the input and output streams of a data stream analysis system. Moreover, we have established sophisticated well-formation constraints for the workflow of a data stream analysis system which we will present later in this section.

5.2.1 Workers

Each worker (w) belongs to a single data stream analysis system ($dsas$) and is uniquely identified by a worker identifier (wid). It processes elements from a set of input streams (DS_{in}). More precisely, for each element a worker receives via one of its input streams it performs some freely-programmable code (co) which defines the analysis procedure of a worker and thus the subtask of the overall analysis which the worker performs. A worker can perform simple tasks, such as stateless filtering of input stream elements (e.g., filtering player sensor input stream elements with heart rate values above 100), as well as very sophisticated tasks, such as detecting successful passes, calculating team area states, or generating fitness statistics, which require keeping state and thus stateful element processing. Every produced (intermediate or final) analysis result can be emitted as an element of one of the worker's output streams (DS_{out}).

Before we continue with defining restrictions on the input and output stream set of a worker, we want to highlight that the state of a worker is not to be confused with the state a worker can emit in a state stream element (such as the current team area state of a certain team or the current pressing intensity state). The former comprises all data a worker keeps, typically in a fault-tolerant and scalable way. The latter is only some data which can be (and most of the time are) part of the worker's state. That is, a worker's state might not only comprise the current team area state of a single team but also the current team area state of another team, a history of the latest team area states the worker has generated, and/or data for a different analysis performed by the same worker. In order to keep our model generic we do not define how the state of a worker is managed. Instead, we refer the interested reader to [TSM18] which surveys various state management approaches.

As we have defined raw input stream generating devices to be the only components which emit raw input stream elements (see Section 5.1), workers only emit event, state, and statistics stream elements restricting the set of output streams of a worker to contain only event streams, state streams, and statistics streams. We argue that this restriction is appropriate as every, albeit small, modification or action a worker performs on the elements of a raw input stream qualifies the output stream it emits to be classified as an event, state, or statistics stream. For instance, it is reasonable that a worker which solely filters raw input stream elements in a stateless way emits the content of the raw input stream elements in new event stream elements as in this case the event can be regarded as the fulfillment of the filtering condition which the worker has detected. Moreover, it is reasonable that a worker which consumes player sensor input stream elements and enriches them with velocities (calculated by keeping a history of the contained positions) or transforms some contained values (e.g., transforms the positions to another coordinate system) emits the modified elements in a state stream as every element of this stream contains information about the state of a player which the worker has calculated. In fact, we argue that even filterless copying the content of each element of a raw input stream into a new state stream element without adding additional values or transforming contained values would be valid as simply copying values from a raw input stream element can be regarded as the most trivial state calculation. The alternatives to that would be to modify our model in a way that also workers can emit raw input streams or to introduce a dedicated additional stream category only for this purpose. While the former is no option as the most important characteristic of raw input streams which we will leverage in later discussions is that a raw input stream element is emitted by an external raw input stream generating device and thus actually received by a component of the data stream analysis system, we argue that the latter is neither necessary nor reasonable since classifying such output streams simply as state streams has no negative implications but adding special-purpose categories increases the complexity of the model.

All event, state, and statistics stream elements generated by a worker are not only available for all external consumers and thus output stream elements of the data stream analysis system the worker belongs to but also available for further processing at all other workers of the same data stream analysis system. In consequence, a worker can consume raw input stream elements emitted by raw input stream generating devices, making them input stream elements of the data stream analysis system it belongs to, as well as event, state, and statistics stream elements emitted by other workers of the same data stream analysis system. However, although we permit cycles in the workflow (see below), we prohibit loops [Har69, p. 10]. That is, we prohibit a worker to consume event, state, and/or statistics stream elements itself has emitted. We do so as permitting this would introduce no benefit since a worker can keep all information contained in an element itself emits in its local state. To ensure this we define that the set of input streams and the set of output streams of a worker have to be disjoint (i.e., $DS_{in} \cap DS_{out} = \emptyset$). Admittedly, this restriction does also prevent a worker w_1 to consume elements another worker w_2 of the same data stream analysis system has emitted in a data stream w_1 has in its output stream set. Nevertheless, we argue that this is still a tolerable restriction as we have not been able to construct even a single example in which this restriction limits the analysis capabilities.

Moreover, the code of a worker is not only performed when an element of one of its input streams is received. Instead, a worker can further have a timer which triggers the execution of the code periodically with a worker-specific period (T). This feature is very useful for implementing workers which should periodically perform some work such as emitting statistics stream elements containing statistics for some interval. For instance, elements of the fitness statistics stream sketched in Example 4.5 should be emitted in two and five minute intervals. This stream can be generated by a worker which consumes the player sensor input stream and which has a timer with a one minute period. Whenever a new player sensor input stream element is received, the state for the current two and five minute interval has to be updated. Whenever the timer triggers the execution it has to be checked for every statistics interval if it has ended and thus a new fitness statistics stream element has to be generated and emitted. Both processes can be integrated into the same code as the code is freely-programmable and thus can contain if-statements to branch for different input streams and the timer. As a matter of course, setting a timer for a worker is optional. If no timer is set for a worker, the period is set to null (i.e., $T = \lambda$).

Although the timer feature is not supported by all worker-based data stream analysis systems (e.g., PAN [Pro14; PGS16a; PGS16b] does not support timers), we have decided to integrate this feature into our model. We argue that this is reasonable since our implementation supports timers (based on Samza's window function [Sam17l], see Section 8.2) and since our model is still valid for systems which do not support timers as the period can be set to null for all workers.

Let $\widehat{DSAS}$ be the global set of all data stream analysis systems, $\widehat{W}$ be the global set of all workers of all data stream analysis systems, and Dom_{wid} be the worker identifier domain. We formally define a worker as follows:

Definition 5.2 Worker

A worker w is a 6-tuple $\langle dsas, wid, DS_{in}, DS_{out}, co, T\rangle$ where $dsas \in \widehat{DSAS}$ is the data stream analysis system the worker belongs to, $wid \in Dom_{wid}$ is a unique worker identifier (i.e., $UID(\widehat{W}) = \{wid\}$), $DS_{in} \subseteq \widehat{DS}$ is the set of input streams of the worker, $DS_{out} \subseteq \left\{ds \;\middle|\; ds \in \widehat{DS} \wedge ds.cat \neq \text{"rawInput"} \wedge ds \notin DS_{in}\right\}$ is the set of output streams of the worker, co is the code the worker performs to process an element of its input streams, and $T \in \mathbb{N}_0 \cup \lambda$ is the period in which the optional timer triggers the code execution (or null if there is no timer).

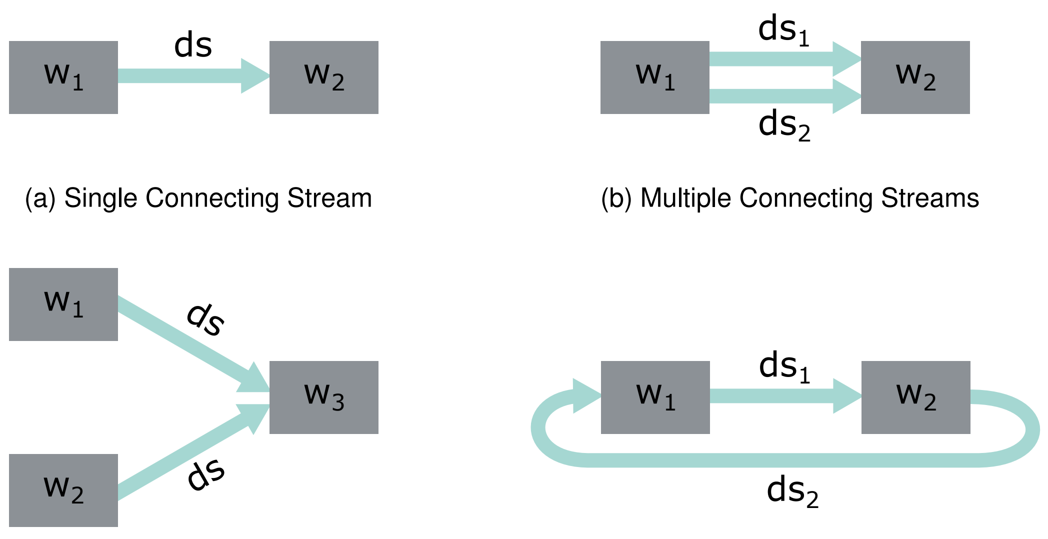
5.2.2 Workflow

In their entirety the input and output stream sets of the workers of a data stream analysis system implicitly define the overall workflow (wf) of the data stream analysis system. This results in the fact that a data stream analysis system has exactly one workflow. However, as we will show later, this does not restrict the analysis potential of a data stream analysis system since it is still possible to perform multiple different analysis tasks, each on the basis of different raw input streams, in the same data stream analysis system.

The workflow of a data stream analysis system can be best modeled as a labeled[1] directed multigraph[2] [Har69, pp. 9–10]. Each worker of a data stream analysis system is a labeled vertex in this graph. Moreover, there is a labeled directed edge $\langle w_1, ds, w_2\rangle$, i.e., a directed edge from the vertex representing worker w_1 to the vertex representing worker w_2 labeled with ds (see Figure 5.2(a)), for each data stream ds in the input stream set of w_2 which is contained in the output stream set of w_1. Note that this definition results in the fact that there can be multiple edges (e.g., $\langle w_1, ds_1, w_2\rangle$ and $\langle w_1, ds_2, w_2\rangle$ as illustrated in Figure 5.2(b)) between two vertices representing w_1 and w_2 if there are multiple different data streams (e.g., ds_1 and ds_2) which are in the input stream set of w_2 and in the output stream set of w_1. The existence of multiple data streams between two

[1] We could define the labels of a graph formally to be the unique identifier of the worker ($w.wid$) for every worker vertex, a dedicated identifier for the artificial entry components vertex that we will introduce below, and the name of the data stream ($ds.name$) for every edge. However, instead of explicitly defining the identifier of every worker and the name of every data stream, we have decided to use labels in the following figures which facilitate an easy matching of vertices and edges to information in the text.

[2] Note that the graph which models the workflow is guaranteed to be only a multigraph but no pseudograph [Har69, p. 10] as we have excluded loops ($\langle w, ds, w\rangle$) by posing the restriction that the input stream set and the output stream set of a worker have to be disjoint.



(a) Single Connecting Stream

(b) Multiple Connecting Streams

(c) Stream Emitted by Multiple Workers

(d) Cycle

Figure 5.2 Workflow Parts. (a), (b), (c), and (d) show different workflow graph parts that illustrate how workers can be connected in a workflow by means of their input and output streams. The gray boxes represent worker vertices (w_1, w_2, and w_3) of the graph which models the workflow. The mint arrows visualize directed edges of the graph, i.e., data streams (ds, ds_1, and ds_2) emitted by one and consumed by another worker vertex.

workers might be required as w_2 might, for instance, further analyze multiple events which w_1 has detected and emitted in elements of different event streams. Although these edges could be combined into one edge for all data streams (e.g., $\langle w_1, ds_1, w_2\rangle$ and $\langle w_1, ds_2, w_2\rangle$ could be combined to $\langle w_1, \{ds_1, ds_2\}, w_2\rangle$), we argue that representing this dataflow with a single edge per data stream is the clearer approach.

Since there might be a worker w_1 and a worker w_2 which contain the same data stream ds in their output stream sets, there can be two directed edges $\langle w_1, ds, w_3\rangle$ and $\langle w_2, ds, w_3\rangle$ if there is a worker w_3 which has ds in its input stream set (see Figure 5.2(c)). This is rational as, at least in theory, two (or even more) workers can emit elements of the same event, state, or statistics stream and thus the workers which have this event, state, or statistics stream in their input stream sets consume and process all elements emitted by both workers.

Moreover, albeit uncommon, the graph and thus the workflow can exhibit cycles. The most straightforward cycle is directly between two workers. This is for instance the case if there is a worker w_1 with ds_1 in its output stream set and ds_2 in its input stream set as well as worker w_2 with ds_2 in its output stream set and ds_1 in its input stream set resulting in the two directed edges

$\langle w_1, ds_1, w_2\rangle$ and $\langle w_2, ds_2, w_1\rangle$ (see Figure 5.2(d)). The potential presence of such two worker cycles is the reason why the graph which models the workflow is not oriented [Har69, p. 10] but only directed. In addition, there can be also larger cycles involving several workers. Although we take the view that cycles are rarely useful and thus should be introduced only for good reason, we argue that it is reasonable to permit cycles as they can be leveraged to implement a feedback loop in the analysis workflow. For instance, in a football analysis workflow the switch play event stream emitted by worker w_1 might be consumed as an input stream by worker w_2 which emits a defensive/offensive phase transition event stream which is in turn consumed as an input stream by w_1 as not only the switch play events might trigger phase transitions but also the current phase might affect which and how switch play events are detected.

A special case which we have not considered yet are those workers which have raw input streams in their input stream set. These workers consume raw input stream elements emitted by external raw input stream generating devices, either exclusively or in addition to event, state, and/or statistics stream elements emitted by other workers. However, the dataflow between these workers and the raw input stream generating devices is not established directly. Hence, a raw input stream generating device *igd* which emits elements of the raw input stream *ds* is not a vertex of the graph and there is no directed edge $\langle igd, ds, w\rangle$ even if *ds* is contained in the input stream set of w. Instead, we assume in our model that data stream analysis systems comprise dedicated *entry components* which receive the raw input stream elements and forward them immediately to all interested workers. More precisely, we assume that each data stream analysis system comprises a set of entry components which jointly receive all elements of all raw input streams that are contained in the input stream set of any worker belonging to the data stream analysis system and that each entry component immediately sends every received element of a raw input stream *ds* to every worker of the data stream analysis system which has *ds* in its input stream set.

Let $\widehat{EC}$ be the global set of entry components, and Dom_{ecid} be the entry component identifier domain. We formally define an entry component as follows:

Definition 5.3 Entry Component

An entry component *ec* is a pair $\langle dsas, ecid\rangle$ where $dsas \in \widehat{DSAS}$ is the data stream analysis system the entry component belongs to and $ecid \in Dom_{ecid}$ is a unique entry component identifier (i.e., $UID(\widehat{EC}) = \{ecid\}$).

Modeling entry components and thus the components where raw input stream elements enter a data stream analysis system separately is beneficial for the stream time model which we will present in Chapter 6. However, note that in practice the entry components can be also integrated in dedicated workers of the workflow or other components. For instance, in our implementation we regard the brokers of the publish/subscribe system which is used to transfer raw input stream elements from the raw input stream generating devices to the workers which consume these elements as the entry components (see Section 8.2.3.1 for more details).

Literature Discussion 5.3 Entry Component

Equivalent components have been introduced already under different terms in the system models of some but not all worker-based data stream analysis systems presented in literature. For instance, in Storm a workflow (called topology) entails so-called *spouts* which pull data stream elements from the external world and forward them to the other workers (called bolts) [TTS+14]. Similarly, MillWheel describes that input data enter the system via *injectors* [ABB+13]. Moreover, the PAN workflows presented in [PGS16b] comprise dedicated *forwarder workers* which receive data stream elements from raw input stream generating devices and provide them for other workers and external consumers.

Note that besides the jointly forwarding assumption mentioned above we pose only a single additional assumption on the entry components of a data stream analysis system. Namely, we assume that each raw input stream element is received only by a single entry component of the data stream analysis system. However, we explicitly do not assume that all elements of a raw input stream are received by the same entry component. We have decided to refrain from introducing this restriction as it is not required throughout this thesis and would thus only unnecessarily limit the applicability of our model. For the same reason, we have decided to keep the formal definition of an entry component as simple as possible. Moreover, since the fact which entry components receive which raw input stream elements and thus elements of which raw input streams are sent by which entry components to which workers might depend on runtime conditions, we have decided to do not add each individual entry component as a vertex and directed edges between individual entry components and workers with raw input streams in their input stream sets to the graph. Instead, we define the graph which models the workflow to contain only a single artificial entry components vertex denoted with ec_{all} which reflects all entry components of the data stream analysis system and directed edges between this vertex and the worker vertices. That is, the graph contains the directed edge $\langle ec_{all}, ds, w\rangle$ if ds is a raw input stream contained in the input stream set of w.

Let $\widehat{WF}$ be the global set of all workflows. We formally define the workflow formed by the workers of a data stream analysis system as follows:

Definition 5.4 Workflow

A workflow wf is a triple $\langle dsas, V, E\rangle$ where $dsas \in \widehat{DSAS}$ is the data stream analysis system the workers which form the workflow belong to, and V and E are the vertices and edges of the graph modeling the workflow, respectively.

$$\begin{aligned} V &= \left\{w \;\middle|\; w \in \widehat{W} \wedge w.dsas = dsas\right\} \cup ec_{all} \\ E &= \left\{\langle ec_{all}, ds, w\rangle \;\middle|\; w \in V \backslash ec_{all} \wedge ds \in w.DS_{in} \wedge ds.cat = \text{“rawInput”}\right\} \cup \\ &\quad \left\{\langle w_1, ds, w_2\rangle \;\middle|\; w_1, w_2 \in V \backslash ec_{all} \wedge ds \in w_1.DS_{out} \wedge ds \in w_2.DS_{in}\right\} \end{aligned}$$

Every data stream analysis system has exactly one workflow.

$$\forall\, dsas \in \widehat{DSAS} : \exists_{=1}\, wf \in \widehat{WF} : wf.dsas = dsas$$

In consequence, each workflow is uniquely identified by the data stream analysis system $dsas$ it belongs to.

$$UID(\widehat{WF}) = \{dsas\}$$

Example 5.1 Abstract Sample Workflow

Assume that there is a data stream analysis system $dsas$ consisting of eight workers with the following input stream sets and output stream sets:

$$\begin{aligned} &w_1.DS_{in} = \{ds_1, ds_2\} && w_1.DS_{out} = \{ds_3, ds_4\} \\ &w_2.DS_{in} = \{ds_2\} && w_2.DS_{out} = \{ds_5, ds_6, ds_7\} \\ &w_3.DS_{in} = \{ds_3, ds_5, ds_6\} && w_3.DS_{out} = \{ds_8, ds_9\} \\ &w_4.DS_{in} = \{ds_7\} && w_4.DS_{out} = \{ds_{10}\} \\ &w_5.DS_{in} = \{ds_8\} && w_5.DS_{out} = \{ds_{11}, ds_{12}\} \\ &w_6.DS_{in} = \{ds_9\} && w_6.DS_{out} = \{ds_{13}\} \\ &w_7.DS_{in} = \{ds_{10}\} && w_7.DS_{out} = \{ds_{14}\} \\ &w_8.DS_{in} = \{ds_{11}, ds_{13}\} && w_8.DS_{out} = \{ds_{15}\} \end{aligned}$$

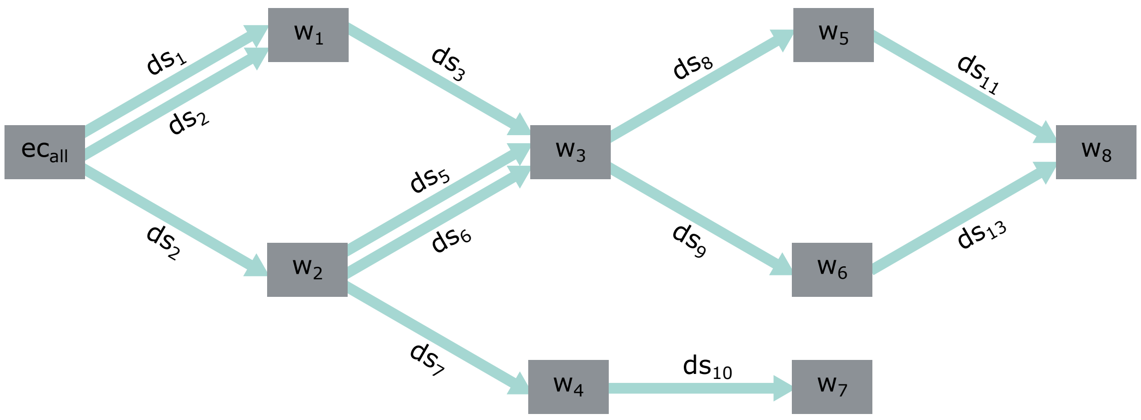


Figure 5.3 Abstract Sample Workflow. The gray boxes represent the vertices of the graph which models the workflow that is formed by the given workers (w_1 to w_8). The mint arrows visualize the directed edges of the graph, i.e., the data streams (ds_1, ds_2, etc.) emitted by one and consumed by another vertex.

Moreover, assume that ds_1 and ds_2 are raw input streams and thus the input streams of *dsas*. Figure 5.3 illustrates the graph which models the workflow formed by the workers of *dsas*. This graph contains nine vertices, one artificial entry components vertex (ec_{all}) and eight worker vertices (w_1 to w_8), and the following twelve directed edges:

$$\begin{array}{llll}
\langle ec_{all}, ds_1, w_1\rangle & \langle ec_{all}, ds_2, w_1\rangle & \langle ec_{all}, ds_2, w_2\rangle & \langle w_1, ds_3, w_3\rangle \\
\langle w_2, ds_5, w_3\rangle & \langle w_2, ds_6, w_3\rangle & \langle w_2, ds_7, w_4\rangle & \langle w_3, ds_8, w_5\rangle \\
\langle w_3, ds_9, w_6\rangle & \langle w_4, ds_{10}, w_7\rangle & \langle w_5, ds_{11}, w_8\rangle & \langle w_6, ds_{13}, w_8\rangle
\end{array}$$

This abstract sample workflow shows that not all output streams of every worker have to be consumed by other workers. Instead, there are four output streams, namely ds_4 emitted by w_1, ds_{12} emitted by w_5, ds_{14} emitted by w_7, and ds_{15} emitted by w_8, which are not consumed by any other worker of the workflow. This is by design as such output streams typically ship final analysis results of the overall analysis workflow which do not have to be further processed by additional workers. Note that these output streams are, as those which are consumed by other workers, output streams of *dsas* and can thus be consumed by external consumers.

Example 5.2 Simplified Pass Detection Workflow

In order to illustrate the stepwise nature of worker-based data stream analysis systems we sketch a system for detecting passes in football matches. Note that the analysis performed by this system is highly simplified for the sake of simplicity and brevity. More precisely, we assume that there is a raw input stream ds_{sensor} which ships elements containing the position of the players and the ball. Moreover, we assume that the system does not regard field areas. That is, we assume that the system does not detect misplaced passes and clearances but only successful passes and interceptions. A much more extensive football analysis workflow which, among other things, also comprises a more sophisticated pass detection will be presented in Section 9.2.

A football analysis system which detects passes (in a simplified way) can be built by splitting the overall analysis task into four subtasks which are performed by four different workers.

First, there is a field object state generation worker w_{obj} which calculates the state (position and velocity) of the players and the ball on the field. For this purpose, w_{obj} consumes the raw input stream ds_{sensor} containing the position of the players and the ball. Whenever w_{obj} receives a new element of ds_{sensor} for a certain player or ball, it extracts the new position, calculates the velocity using the new and the second-last position, and emits both in a new element of the field object state stream $ds_{objState}$.

Second, there is a ball possession worker w_{poss} which detects ball possession change events by means of analyzing the positions and velocities of the players and the ball shipped in elements of the field object state stream $ds_{objState}$. Whenever w_{poss} detects a ball possession change event, it emits a new element in the ball possession change event stream ds_{possEv} which represents this event. Moreover, w_{poss} generates ball possession statistics for the players and the teams and emits them in elements of the ball possession statistics stream ds_{possSt}. This can be done either periodically if w_{poss} has a timer or whenever a new ball possession change event is detected.

Third, there is a kick detection worker w_{kick} which detects if the player who is in possession of the ball kicks the ball. This is done on the basis of the player state, the ball state, and the information who is currently in possession of the ball. Therefore, w_{kick} consumes $ds_{objState}$ and ds_{possEv}. Whenever w_{kick} detects a kick event, it emits a new element in the kick event stream ds_{kickEv} which represents this event.

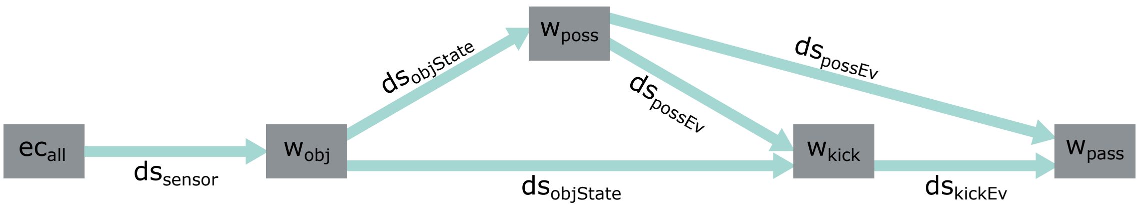


Figure 5.4 Simplified Pass Detection Workflow. The gray boxes represent the vertices of the graph which models the simplified pass detection workflow that is formed by the given workers (w_{obj}, w_{poss}, w_{kick}, and w_{pass}). The mint arrows visualize the directed edges of the graph, i.e., the data streams emitted by one and consumed by another vertex.

Fourth, there is a pass detection worker w_{pass} which detects a pass when a kick event is followed by a ball possession change event, i.e., when the ball is kicked by player X and received by player Y. To do so, w_{pass} consumes ds_{possEv} and ds_{kickEv}. Whenever w_{pass} detects a pass, it distinguishes between successful pass events (player X and Y are members of the same team) and interception events (player X and Y are members of different teams) and emits a new element in the successful pass event stream $ds_{succPassEv}$ or the interception event stream $ds_{interEv}$. Moreover, w_{pass} updates the pass statistics and emits a new element in the pass statistics stream ds_{passSt}. Alternatively, the state for generating the pass statistics (i.e., the number of successful passes, the number of interceptions, etc.) can be only updated and new pass statistics stream elements can be generated and emitted periodically if w_{pass} has a timer.
In consequence, the four workers have the following input stream sets and output stream sets:

$$\begin{aligned}
&w_{obj}.DS_{in} = \{ds_{sensor}\} && w_{obj}.DS_{out} = \{ds_{objState}\}\\
&w_{poss}.DS_{in} = \{ds_{objState}\} && w_{poss}.DS_{out} = \{ds_{possEv}, ds_{possSt}\}\\
&w_{kick}.DS_{in} = \{ds_{objState}, ds_{possEv}\} && w_{kick}.DS_{out} = \{ds_{kickEv}\}\\
&w_{pass}.DS_{in} = \{ds_{possEv}, ds_{kickEv}\} && w_{pass}.DS_{out} = \{ds_{succPassEv}, ds_{interEv}, ds_{passSt}\}
\end{aligned}$$

Figure 5.4 illustrates the graph which models the simplified pass detection workflow formed by these workers. This graph contains five vertices, one artificial entry components vertex (ec_{all}) and four worker vertices (w_{obj}, w_{poss}, w_{kick}, and w_{pass}), and the following six directed edges:

$$\begin{array}{lll}
\langle ec_{all}, ds_{sensor}, w_{obj}\rangle & \langle w_{obj}, ds_{objState}, w_{poss}\rangle & \langle w_{obj}, ds_{objState}, w_{kick}\rangle\\
\langle w_{poss}, ds_{possEv}, w_{kick}\rangle & \langle w_{poss}, ds_{possEv}, w_{pass}\rangle & \langle w_{kick}, ds_{kickEv}, w_{pass}\rangle
\end{array}$$

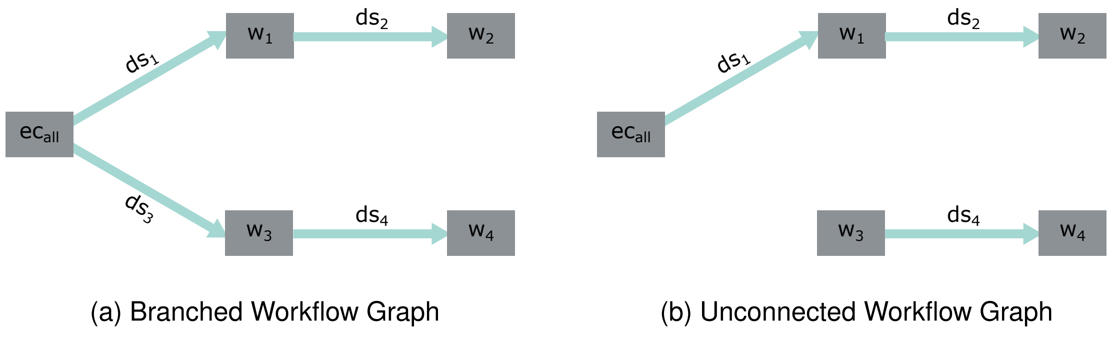


Figure 5.5 **Abstract Sample Workflow with Two Different Analysis Tasks.** The gray boxes represent the vertices of the graph which models an abstract sample workflow that performs two different analysis tasks with two disjoint sets of workers ($\{w_1, w_2\}$ and $\{w_3, w_4\}$). The mint arrows visualize the directed edges of the graph, i.e., the data streams emitted by one and consumed by another vertex. In (a) the graph is branched after the ec_{all} vertex and in (b) the graph is unconnected.

As mentioned above, the fact that a data stream analysis system has exactly one workflow is not restrictive. A data stream analysis system can still perform two different analysis tasks with two disjoint sets of workers which do not share any input and/or output stream. But even in this case in which the workers are designed to form two workflows that perform different analysis tasks, they actually form a single workflow which is either branched completely after the ec_{all} vertex or not represented by a connected [Har69, p. 13] graph (see Figure 5.5).[3] The reason for this is that we define a workflow to be the graph formed by the workers of a data stream analysis system no matter how this graph looks like.

5.2.3 Well-Formation

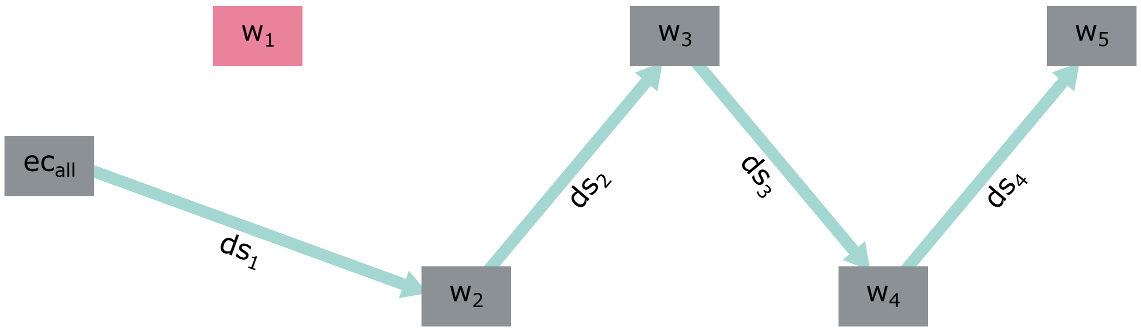
The consequence of the implicit workflow definition is that a workflow formed by the workers of a data stream analysis system can contain workers which not only in practice but by definition never perform any work. If there is a worker in the workflow which does not have any raw input stream or any event, state, or statistics stream which is emitted by any other worker in its input stream set (i.e., if there is a worker represented by a vertex without an incoming edge as illustrated in Figure 5.6(a)), this worker will never receive any data

[3] The workflow graph is completely branched after the ec_{all} vertex if there is no worker with an event, state, or statistics stream in its input stream set which is in the output stream set of a worker that is dedicated for the other analysis task and not connected at all if the workers of one or even both analysis tasks do not contain any raw input stream in their input stream sets.

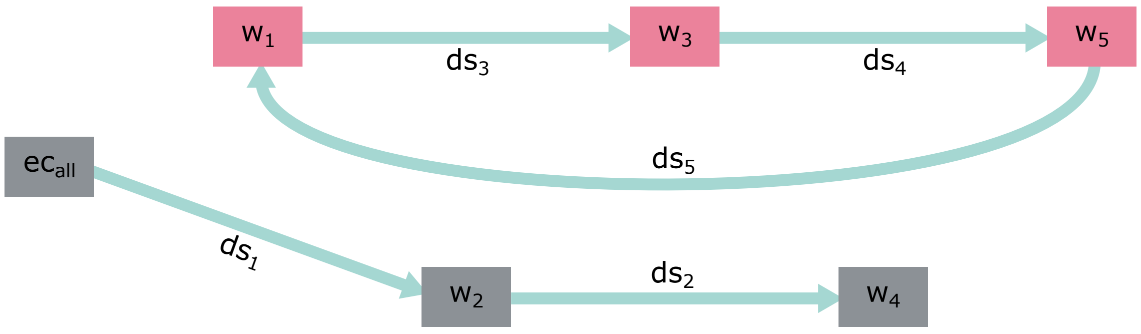
stream element. If this worker additionally does not have a timer, it will never perform its code and can thus be regarded as dead. Moreover, as a matter of course, also workers without timers which have only event, state, and/or statistics streams emitted by dead workers in their input stream sets can by definition never perform any work and can thus be regarded as dead. For instance, if we add a worker w_6 without a timer which solely consumes ds_5 emitted by w_1 to the example illustrated in Figure 5.6(a), i.e., if we add a worker vertex for w_6 and an edge $\langle w_1, ds_5, w_6 \rangle$, not only w_1 but also w_6 will be dead.

The same is true for a set of workers without timers which all have at least one event, state, or statistics stream in their input stream set which is in the output stream set of another worker of this set if none of the workers in this set has an event, state, or statistics stream which is emitted by another worker (not contained in the set) or a raw input stream in its input stream set. This is due to the fact that none of these workers will ever receive an event, state, or statistics stream element that could cause it to perform its code as these elements are only emitted by other workers of the set when they perform their code. Hence, the whole set of workers will never perform any work and can thus be regarded as dead. In the graph that models the workflow such a set of dead workers is represented by a set of worker vertices with the following conditions: First, there is at least one walk [Har69, p. 13] from another worker vertex in this set to each worker vertex in this set. Second, there is no walk from the artificial entry components vertex or a worker vertex which is not contained in this set to any worker vertex in this set. An example for such a set of dead workers is illustrated in Figure 5.6(b). Note that an additional edge from one of the dead worker vertices to any of the other (alive) worker vertices (e.g., $\langle w_1, ds_6, w_2 \rangle$) would not corrupt the illustrated example. However, a reverse edge (e.g., $\langle w_2, ds_6, w_1 \rangle$) or an edge from the artificial entry components vertex to a dead worker vertex (e.g., $\langle ec_{all}, ds_6, w_1 \rangle$) would render w_1, w_3, and w_5 alive.

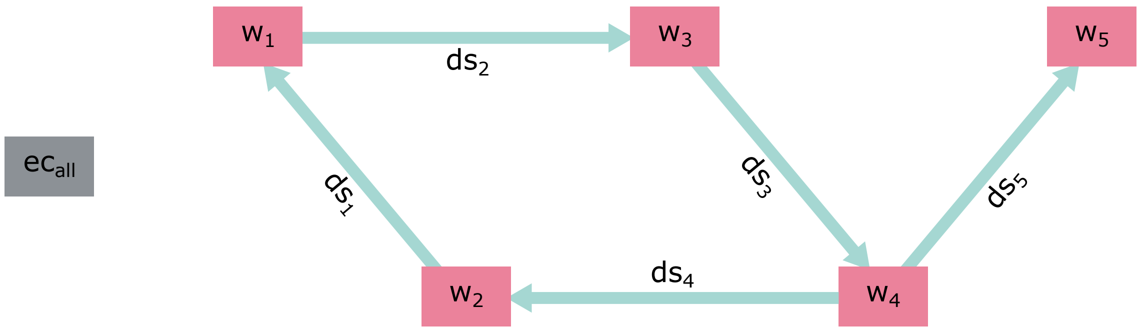
In fact, it is even possible that all workers of the data stream analysis system are dead. This is the case if none of them has a raw input stream in its input stream set (i.e., if there is no edge from the artificial entry components vertex to any worker vertex as illustrated in Figure 5.6(c)) and if none of them has a timer. In this case, the whole workflow will by definition never perform any work and can thus be regarded as dead. This situation can only be solved by setting a timer for one of the workers or by adding a raw input stream to the input stream set of one of the workers. Taken all together, a workflow is neither completely dead nor contains any dead workers if and only if there is a walk



(a) Workflow containing a dead worker (w_1)



(b) Workflow containing a set of dead workers (w_1, w_3, and w_5)



(c) Completely dead workflow

Figure 5.6 **Ill-Formed Workflows.** Under the assumption that none of the workers has a timer, this figure encompasses three ill-formed workflows shown in (a), (b), and (c). The boxes illustrate the vertices of the graphs which model the three ill-formed workflows. Vertices representing dead workers are highlighted in red. The mint arrows visualize the directed edges of the graphs, i.e., the data streams emitted by one and consumed by another vertex.

from the artificial entry components vertex or a vertex representing a worker with a timer to every vertex representing a worker without a timer.

In addition to dead workers, there might be workers which perform some work but by definition never emit the results of this work. This is the case if their output stream set is empty as we assume the event, state, and statistics streams emitted by the workers of a data stream analysis system to be its sole output. Although these workers cannot be regarded as dead as they might perform work, they can be regarded as useless.

We formally define a *well-formed workflow* without these issues as follows:

Definition 5.5 Well-Formed Workflow

A workflow *wf* is well-formed if the following constraints are fulfilled:

1. There is a walk from the artificial entry components vertex or a vertex representing a worker with a timer to every vertex representing a worker without a timer.

$$\begin{aligned}
&\forall\, w \in wf.V \backslash ec_{all} \ : \\
&\quad w.T \neq \lambda \vee \\
&\quad \Big(\exists\ \langle e_1, e_2, \ldots, e_n \rangle \text{ with } e_i \in wf.E \text{ for all } 1 \leq i \leq n \ : \\
&\qquad \Big(e_1.1 = ec_{all} \ \vee\ e_1.1.T \ \neq\ \lambda\Big) \ \wedge\ e_n.3 = w \wedge \\
&\qquad \Big(e_j.3 = e_{j+1}.1 \text{ for all } 1 \leq j \leq n-1\Big)\Big)
\end{aligned}$$

2. No worker of the workflow has an empty output stream set.

$$\forall\, w \in wf.V \backslash ec_{all} \ : w.DS_{out} \neq \emptyset$$

Workflows which are not well-formed are called ill-formed.

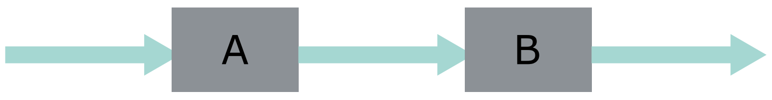
Both sample workflows, the abstract workflow described in Example 5.1 and the simplified pass detection workflow described in Example 5.2, are well-formed as they contain neither dead nor useless workers. In the remainder of this thesis, we assume that the workflow of every data stream analysis system fulfills the constraints given in Definition 5.5 and is thus a well-formed workflow, unless otherwise specified.

5.3 Fully-Parallelized Data Stream Analysis System Model

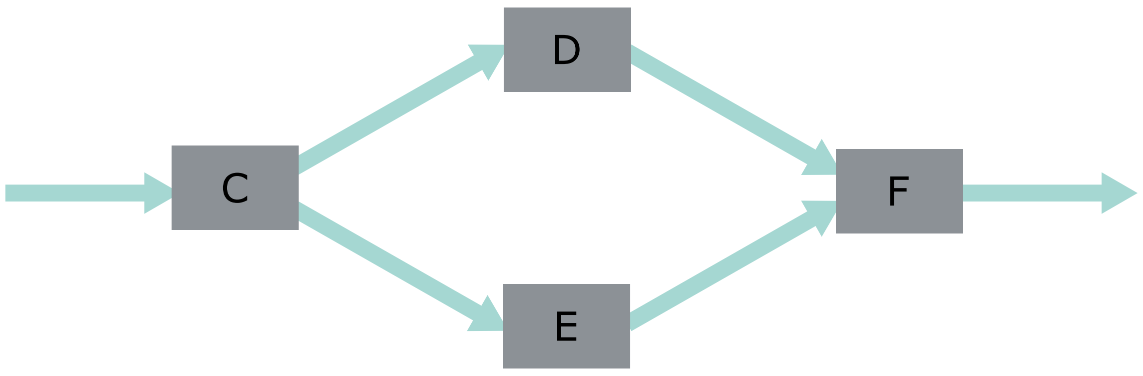
Hirzel et al. [HSS+14] describe parallelism in stream processing graphs consisting of operators by means of introducing three different notions of parallelism, namely *pipeline parallelism*, *task parallelism*, and *data parallelism*. More precisely, they define pipeline parallelism to be "the concurrent execution of producer A with a consumer B" [HSS+14] (see Figure 5.7(a)), task parallelism to be "the concurrent execution of different operators D and E that do not constitute a pipeline" [HSS+14] (see Figure 5.7(b)), and data parallelism to be "the concurrent execution of multiple replicas of the same operator G on different portions of the same data" [HSS+14] (see Figure 5.7(c)). As the workflow of a worker-based data stream analysis system is a stream processing graph we adopt these terms and definitions in this thesis.

Pipeline parallelism and task parallelism are already fully supported in the simplified system model which we have presented in Section 5.2. Pipeline parallelism is covered by the fact that two workers w_A and w_B which are subsequently ordered in the workflow (i.e., worker w_B consumes data stream elements emitted by w_A) perform their work concurrently. For instance, the workers w_4 and w_7 of the abstract sample workflow illustrated in Figure 5.3 and the workers w_{kick} and w_{pass} of the simplified pass detection workflow illustrated in Figure 5.4 are pipeline-parallel. Task parallelism is provided by the fact that a workflow can branch resulting in two workers w_D and w_E which concurrently perform their work but both do not consume data stream elements emitted by each other. For instance, w_5 and w_6 of the abstract sample workflow are task-parallel.

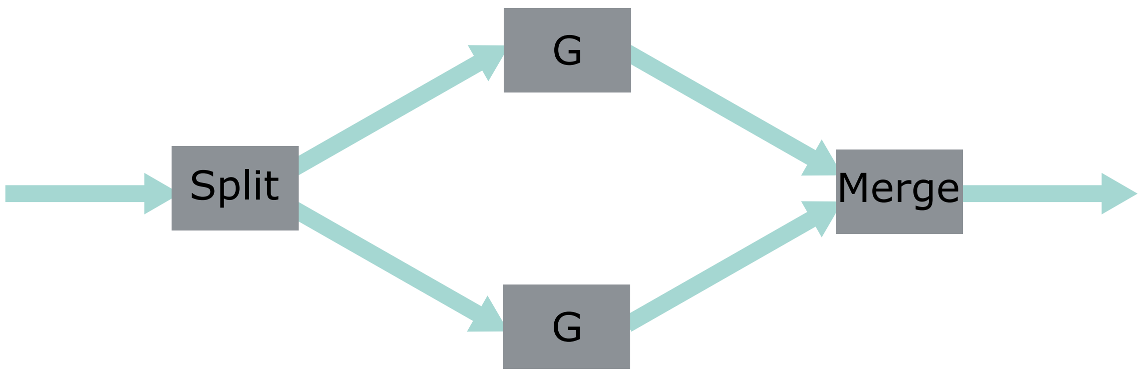
However, the simplified model does not support data parallelism. So far we have presented workers which perform different subtasks of the stepwise analysis as well as how these workers form the workflow which performs the overall analysis task. In doing so we have assumed that there are no data stream partitions but only data streams. If we simply dropped this assumption and considered how and by whom the elements of different data stream partitions are processed, according to the description given in Section 5.2 every worker would process each element of every partition of every data stream which is in its input stream set. This would not provide any data parallelism as there would still be no replicas which perform the same analysis task for a subset of the overall data. In this section, we will extend our model in a way that it supports data parallelism in order to achieve a fully-parallelized data stream analysis system model.



(a) Pipeline-parallel $A \parallel B$



(b) Task-parallel $D \parallel E$



(c) Data-parallel $G \parallel G$

Figure 5.7 Parallelism in Stream Processing Graphs [HSS+14]. (a), (b), and (c) show parts of stream processing graphs that illustrate the three different notions of parallelism. The gray boxes represent operators and the mint arrows visualize data streams emitted by one and consumed by another operator. This figure is a color and shape adapted version of Figure 1 in [HSS+14].

5.3.1 Key-Based Data Parallelism

For doing so we introduce *processors* as the components of a data stream analysis system which actually perform the analysis task defined by a worker and thus as the physical instances of purely conceptual workers which can be regarded as type definitions. Each processor (pr) of a data stream analysis system is assigned to exactly one worker (w). All processors assigned to a certain worker perform the exact same code ($w.co$) when they receive an element of one of the worker's input streams or when their timers trigger. The timer is triggered with the same period ($w.T$) at every processor of w, albeit not necessarily synchronously as different clocks might be used depending on the deployment (see Section 5.4.1) and as the triggering can be slightly deferred (see Section 5.3.2). In contrast, each processor of w receives and thus processes only those elements of every data stream contained in the input stream set of w which have a certain key (k). That is, each processor processes only the elements of a single partition of every input stream of a worker. In fact, we define the analysis task of a worker to be performed completely separately for every key. That is, multiple processors assigned to the same worker perform the analysis completely independently of each other. This leads to the fact that there is no (combined) worker state.[4] Instead, each processor has an individual local state (st) which it does not (and cannot) share. As done in Section 5.2, we do not further define this state but refer to [TSM18] which surveys various state management approaches.

Let $\widehat{PR}$ be the global set of all processors. We formally define a processor as follows:

Definition 5.6 Processor

A processor pr is a triple $\langle w, k, st \rangle$ where $w \in \widehat{W}$ is the worker the processor is assigned to, $k \in Dom_k$ is the key for which the processor performs the analysis task defined by the worker it is assigned to, and st is the unshared local state of the processor.
There is exactly one processor assigned to every worker for each key.

$$\forall\, w \in \widehat{W} : \Big(\forall\, k \in Dom_k : \exists_{=1}\, pr \in \widehat{PR} : pr.w = w \,\wedge\, pr.k = k\Big)$$

In consequence, the combination of the worker w and the key k uniquely identifies a processor.

$$UID(\widehat{PR}) = \{w, k\}$$

The analysis task of a worker is partitioned to a set of processors which perform the analysis task in parallel for different keys. More precisely, as defined in Definition 5.6, a data stream analysis system comprises for each worker exactly

[4] This is the reason why we have not added a state attribute to the worker tuple in Definition 5.2.

one processor for every key.[5] Since each processor can be regarded as a replica of the worker, the introduction of processors enriches our system model with support for data parallelism. Determining the portions of a data stream which are processed by different processors by means of the keys of the data stream elements is denoted in literature as *key-based splitting* [RM19]. Hence, we denote this kind of data parallelism as *key-based data parallelism*.

Literature Discussion 5.4 Key-Based Data Parallelism

The concept of key-based data parallelism including the full separation per partition and thus the unshared state idea is not introduced by us but supported by many state-of-the-art worker-based data stream analysis systems, such as Storm [TTS+14], Samza [Sam17d; NPP+17; KK15], and MillWheel [ABB+13]. In fact, our statement that each processor processes only the elements of a single partition of every input stream of a worker is equivalent to a statement given for Samza, namely "[e]ach task consumes data from one partition for each of the job's input streams." [Sam17d]. This is due to the fact that, as we have mentioned in Section 5.2, Samza is used as the foundation for our implementation and has thus a substantial impact on our model. However, in Samza a data stream partition comprises a subset of the key domain instead of only a single key (see Section 8.2.2 for more details).

An extensive survey which data stream analysis systems and which extensions (i.e., published approaches how to extend an existing system with support for parallelism) support which notions of parallelism and which splitting strategies is given in [RM19]. Literature which describes these systems typically uses the term *task* [TTS+14; Sam17d; NPP+17] or *instance* [RM19] instead of processor. Nevertheless, we have decided to use the term processor in our model since a processor actually processes the data stream elements and since the time notion introduced by assigning timestamps to data stream elements when they are processed by a processor is denoted in literature as processing time (see Section 6.1.3). Moreover, we argue that using the term task instead of processor to denote a replica would introduce confusion as we use the term task in this thesis already to denote the job a whole workflow or a single worker fulfills.

[5] The fact that this might result in a very large or even infinite ($|Dom_k| = \infty$ if the key is defined to be an arbitrary String) number of processors for every worker is unproblematic for a theoretical system model. We will present in Section 8.2.2 how this problem is solved in our implementation by means of adopting Samza's approach to define partitions on the basis of subsets of the key domain instead of single keys while still preserving that the analysis task is performed independently for every key by means of adding extensions to Samza.

A straightforward potential introduced by every kind of data parallelism is the avoidance of computational overload situations by means of distributing the processors assigned to a worker onto multiple machines (see Section 5.4 for more details on the physical deployment). Key-based data parallelism further exhibits the advantage that the analysis can be cleanly separated in a way that is required by the application. This is possible as the analysis task of every worker is partitioned by means of application-defined keys which are either set by the raw input stream generating devices or by preceding freely-programmable workers (more infos on repartitioning will be given below). We leverage this feature in our real-time football analysis application to independently analyze multiple football matches in parallel in the same data stream analysis system by means of setting the key of every data stream element to be the unique identifier of the match it belongs to (see Chapter 9).

Note that the keys of the elements which a processor emits in data streams contained in the output stream set of the worker it is assigned to are not restricted. Although a processor typically emits only elements which have the same key as the input stream elements it consumes, a processor is also able to emit elements with different keys and thus in multiple partitions of every output stream of the worker it is assigned to. This is required in order to enable performing analysis tasks with arbitrary partitioning schemes by adding dedicated repartitioning workers as suggested in [Sam17h; KK15]. For instance, assume that there is a raw input stream generating device which ships tweets (i.e., the tweeted text, the user identifier, and the country where the tweet has been posted) in elements of a raw input stream. Further assume that the key of every raw input stream element is set to be the user identifier. In order to separate and parallelize the analysis task of a Twitter [Twi20] analysis system by means of the country (e.g., to create Twitter statistics for each country) one has to add a worker at the beginning of its workflow which consumes this raw input stream, extracts the country from each element, constructs new elements with the same content but the country as the key, and emits these elements in its output stream.[6] Each processor of this worker performs this repartitioning task (defined in the code of the worker) for a single user and thus for a single partition of the raw input stream but as the user might move around the world the processor might emit elements with multiple different keys and thus

[6] Note that the output stream is a state stream as a worker is not allowed to emit raw input streams (see Section 5.2.1). This is valid since each new output stream element can be regarded to ship the information which tweet has been posted last in a certain country by a certain user.

in multiple different output stream partitions. In fact, the absence of an output stream element key restriction even enables partitioning different parts of the overall analysis workflow of a data stream analysis system on the basis of different keys by means of adding multiple of these workers. That is, one could add another worker to the Twitter analysis workflow which repartitions the elements of the raw input stream, for instance, by means of the length of the text (k_1 for elements representing tweets whose text is shorter than 100 characters and k_2 for the other elements). We want to highlight that such workers do not conflict with the idea of key-based data parallelism as each processor of every worker of a data stream analysis system still performs the analysis task of the worker it is assigned to for a subset of the worker's input stream elements which is defined by the keys of these elements.

Example 5.3 Key-Based Data Parallelization

Assume that there is a worker w with two input streams and one output stream (see Figure 5.8(a)) resulting in the following input and output stream sets:

$$w.DS_{in} = \{ds_A, ds_B\} \qquad w.DS_{out} = \{ds_C\}$$

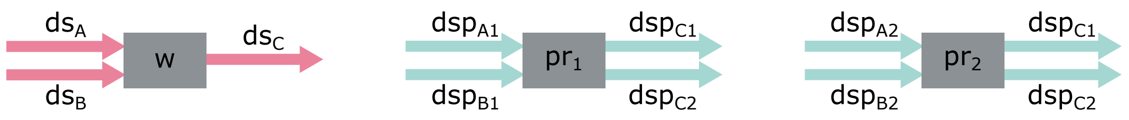
Further assume that there are two different keys ($Dom_k = \{k_1, k_2\}$) resulting in six partitions of three data streams:

$$\begin{aligned} dsp_{A1} &= \langle ds_A, k_1 \rangle \qquad & dsp_{A2} &= \langle ds_A, k_2 \rangle \\ dsp_{B1} &= \langle ds_B, k_1 \rangle \qquad & dsp_{B2} &= \langle ds_B, k_2 \rangle \\ dsp_{C1} &= \langle ds_C, k_1 \rangle \qquad & dsp_{C2} &= \langle ds_C, k_2 \rangle \end{aligned}$$

In consequence, there are two processors pr_1 and pr_2 which perform the analysis task defined by w in parallel (i.e., concurrently) for all elements with key k_1 and key k_2, respectively.

$$\begin{aligned} pr_1.w &= w \qquad & pr_1.k &= k_1 \\ pr_2.w &= w \qquad & pr_2.k &= k_2 \end{aligned}$$

According to the key-based data parallelism concept, pr_1 consumes and processes only the elements of dsp_{A1} and dsp_{B1} while pr_2 consumes and processes only the elements of dsp_{A2} and dsp_{B2} (see Figure 5.8(b) and Figure 5.8(c)). However, as the keys of the elements which the processors assigned to w emit are not restricted, pr_1 and pr_2 might both emit elements of dsp_{C1} and dsp_{C2}.



(a) Worker w (b) Processor pr_1 (c) Processor pr_2

Figure 5.8 **Key-Based Data Parallelization.** The gray boxes represent the worker w as well as the processors pr_1 and pr_2. The red arrows in (a) visualize the data streams (ds_A, ds_B, and ds_C) which the worker consumes and emits. The mint arrows in (b) and (c) visualize the data stream partitions (dsp_{A1}, dsp_{A2}, dsp_{B1}, dsp_{B2}, dsp_{C1}, and dsp_{C2}) which the processors consume and in which the processors can emit data stream elements.

5.3.2 Processing Procedure

Regarding the processing procedure of the processors, we assume that processors do not process multiple elements at the same time. Instead, we define that every processor processes only one element at a time. Moreover, we assume that the timer does not trigger the execution of the code if the code is already in execution for an input stream element. Instead, we define that the timer-triggered execution is deferred until the current execution is finished even if this procedure might slightly violate the specified timer period. Hence, there is no concurrency inside a processor.

As we have mentioned in Chapter 4, the elements of a data stream partition are totally ordered by means of their sequence numbers. On this basis, we define that every processor processes the elements of every data stream partition it consumes sequentially ordered with respect to their sequence numbers. The great benefit of this is that all processors (of all workers) which consume elements of a certain data stream partition are guaranteed to process them in the same order (more details and a proof is given in Section 6.2.3). However, since processors might receive elements out-of-order with respect to their sequence numbers, every processor has to buffer incoming elements until it is guaranteed that no more late elements (i.e., elements with a smaller sequence number) arrive in order to enforce processing elements of a data stream partition in correct order with respect to their sequence number. In our system model, we assume that every processor is able to do so. Details regarding sequence numbers, reasons for out-of-order arrivals, the additional time-related assumptions implied by the buffering assumption[7], and the consistency of the order in which processors process elements of a data stream partition with the orderings introduced by the timestamps of different time notions will be presented in Section 6.2.

In contrast, there are no ordering regulations based on sequence numbers for elements of different data stream partitions. This is due to the fact that elements of different data stream partitions cannot be ordered with respect to their sequence numbers (see Section 6.2.1). That is, there is no sequence number based rule which defines if a processor which performs the analysis task defined by a worker with ds_1 and ds_2 in its input stream set for key k first processes element dse_1 of the partition dsp_1 of ds_1 (with $dsp_1.k = k$) or element dse_2 of the partition dsp_2 of ds_2 (with $dsp_2.k = k$). Instead, we simply define in our system model that a processor can process elements of different data stream partitions in an arbitrary order as long as the processor preserves the guarantee that elements of the same data stream partition are processed in correct order with respect to their sequence numbers.

Literature Discussion 5.5 Processing Procedure

We want to highlight that we have adopted the processing procedure of our model, namely the fact that there is no concurrency inside a processor, the fact that the time-triggered execution might be deferred, the fact that the elements of a data stream partition are totally ordered by means of their sequence numbers and guaranteed to be processed in this (and thus the same) order, and the fact that elements of different data stream partitions are not ordered, from Samza's fundamental concept [Sam17d], its default concurrency level [Sam17f; Sam17l], and its interaction with Kafka [KK15].

Regarding entry components the statement that we do not assume that all elements of a raw input stream are received by the same entry component (see Section 5.1) has to be refined in our fully-parallelized model. In fact, we do not even assume that all elements of a raw input stream partition are received by the same entry component. Instead, two elements of the same raw input stream partition might enter a data stream analysis system via different entry components of this system.

As a final note, we want to point out that we will refer in the remainder

[7] In a nutshell, the assumption that a processor is able to buffer elements until it is guaranteed that no more late elements will arrive implies the assumption that there is a known time bound for late arrivals which is small enough to afford meeting the real-time demands of the analysis scenario (see Literature Discussion 6.5 in Section 6.2.3).

of this thesis with the term system model to this extended fully-parallelized system model and with the term data stream analysis system to a worker-based data stream analysis system that is designed according to this fully-parallelized system model, unless otherwise specified.

5.4 Machine and Network Model

So far we have modeled the architecture of a worker-based data stream analysis system by means of defining its input sources, its internal components, and how these components interact. This section covers the machine and network related aspects of our system model. That is, we will present how the components of a data stream analysis system are deployed onto physical machines. Moreover, we will pose some assumptions on the machines and the network.

5.4.1 Deployment and Machine Assumptions

The architecture of a data stream analysis system is composed of entry components and processors.[8] To execute a data stream analysis system all entry components and processors have to be deployed on a set of machines. Each entry component and processor is hosted by exactly one machine. However, a machine can host multiple processors (assigned to the same or different workers) and/or entry components. Each raw input stream generating devices is a dedicated machine itself which does not host any entry component or processor.

Example 5.4 Deployment of the Simplified Pass Detection System

In order to illustrate the deployment of worker-based data stream analysis systems Figure 5.9 shows a sample deployment of the simplified pass detection system whose workflow is presented in Example 5.2. Assume that the simplified pass detection system has three entry components (ec_1, ec_2, and ec_3). Moreover, assume that there are two raw input stream generating devices (igd_1 and igd_2) which produce and emit elements of the sole input stream ds_{sensor}. Furthermore, assume that there are two different keys ($Dom_k = \{k_1, k_2\}$) resulting in the following eigth processors which perform the analysis task defined

[8] Note that a worker is a purely conceptual component formed by the set of processors which perform the analysis task defined by the worker. The same is true for the workflow of a data stream analysis system which is formed by the workers of the system. Hence, neither the workers nor the workflow of a data stream analysis system are deployed onto any physical machine.

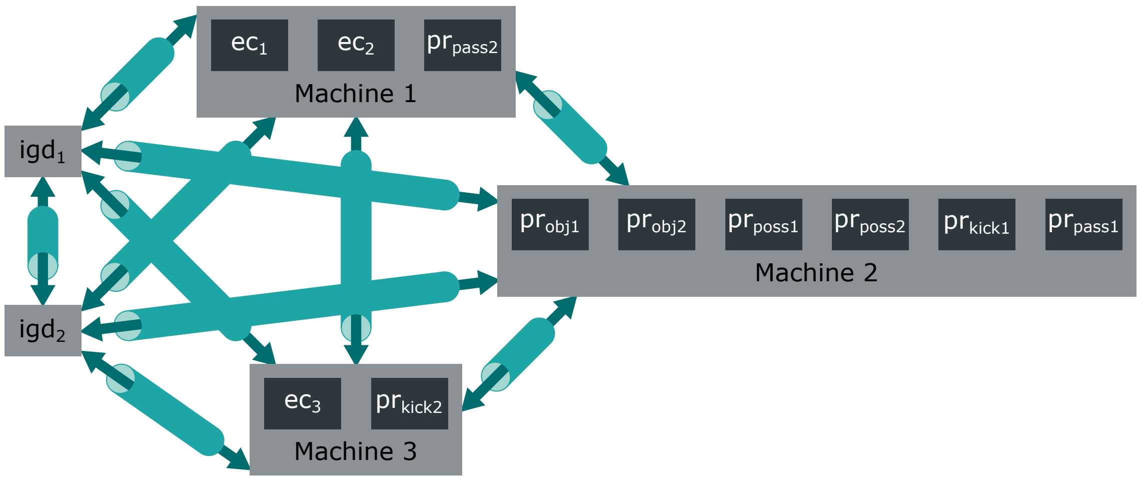


Figure 5.9 Deployment of the Simplified Pass Detection System. The dark-gray boxes illustrate the entry components (ec_1, ec_2, and ec_3) and the processors (pr_{obj1}, pr_{obj2}, pr_{poss1}, pr_{poss2}, pr_{kick1}, pr_{kick2}, pr_{pass1}, and pr_{pass2}). The light-gray boxes represent the raw input stream generating devices (igd_1 and igd_2) as well as the three machines which host the entry components and the processors. The mint pipes between the light-gray boxes illustrate the bidirectional network channels between the raw input stream generating devices and the machines via which packets containing data stream elements and other information are transferred.

by the four workers (w_{obj}, w_{poss}, w_{kick}, and w_{pass}) of the simplified pass detection workflow in parallel for all elements with key k_1 and key k_2.

$$
\begin{aligned}
pr_{obj1}.w &= w_{obj} & pr_{obj1}.k &= k_1\\
pr_{obj2}.w &= w_{obj} & pr_{obj2}.k &= k_2\\
pr_{poss1}.w &= w_{poss} & pr_{poss1}.k &= k_1\\
pr_{poss2}.w &= w_{poss} & pr_{poss2}.k &= k_2\\
pr_{kick1}.w &= w_{kick} & pr_{kick1}.k &= k_1\\
pr_{kick2}.w &= w_{kick} & pr_{kick2}.k &= k_2\\
pr_{pass1}.w &= w_{pass} & pr_{pass1}.k &= k_1\\
pr_{pass2}.w &= w_{pass} & pr_{pass2}.k &= k_2
\end{aligned}
$$

The two raw input stream generating devices are independent machines. The three entry components and the eight processors are deployed onto three machines. A first machine hosts ec_1, ec_2, and pr_{pass2}. A second machine hosts pr_{obj1}, pr_{obj2}, pr_{poss1}, pr_{poss2}, pr_{kick1}, and pr_{pass1}. A third machine hosts ec_3 and pr_{kick2}.

In our model we assume that each physical device, i.e., each machine which hosts entry components and/or processors and each raw input stream generating device, is equipped with a local clock. Hence, we assume that every raw input stream generating device, every entry component, and every processor is able to generate timestamps with a certain clock-specific granularity (e.g., milliseconds resolution). Moreover, the clock of a machine is also used by processors it hosts to trigger the code execution if they are assigned to a worker with a timer. More details regarding the synchronicity of these clocks and the semantics of the timestamps a raw input stream generating device, an entry component, and a processor generates and assigns to data stream elements will be given and discussed in our stream time model (see Chapter 6). Note that we denote in the remainder of this thesis the clock of a machine also as the (local) clock of the processors and the (local) clock of the entry components which the machine hosts.

In practice, any machine can fail. There are various failure models described in literature. The most prominent are the fail-stop failure model [Sch84] and the byzantine failure model [LSP82]. Moreover, there are multiple approaches how to handle machine failures in worker-based data stream analysis systems. For instance, popular state-of-the-art worker-based data stream analysis systems from the Apache family, namely Storm [TTS+14] and Samza [NPP+17; KK15], as well as Google's MillWheel [ABB+13] provides some failure handling mechanisms. Moreover, also earlier work such as [Bre08] has presented protocols for reliable worker-based data stream analysis in the presence of machine failures. However, describing and discussing the different failure models and failure handling approaches is out of the scope of this thesis as failure tolerance was not a topic of our research. Instead, we assume in this thesis that neither the machines which host entry components and/or processors nor the raw input stream generating devices fail. We argue that this assumption is valid since in the team sports analysis scenarios our work focuses on, such as the real-time football analysis scenario described in Section 2.1, failure tolerance is desirable but not necessary. In contrast, in health care scenarios, such as the telemonitoring scenario [Bre08] focuses on, failure tolerance is indispensable as every unhandled machine failure can result in severe health problems for the patients.

5.4.2 Network Assumptions and Communication Model

As we have explained above, raw input stream generating devices send elements of raw input streams to entry components of the data stream analysis system which forward these elements to processors of workers which have the corresponding raw input stream in their input stream sets. Moreover, every processor consumes elements of event, state, and/or statistics streams the worker it is assigned to has in its input stream set from processors that are assigned to workers of the same data stream analysis system which have these data streams in their output stream set. In our system model, we define that data stream elements are transferred via network channels. To support arbitrary deployments on a given set of machines, we assume that there is a bidirectional network channel between each pair of machines as well as between each raw input stream generating device and each machine. Moreover, we assume that there is a bidirectional network channel between each pair of raw input stream generating devices. These network channels are required for synchronizing the clocks of the raw input stream generating devices in order to obtain one consistent generation time space (see Section 6.1.1). The network formed by the raw input stream generating devices, the machines which host entry components and/or processors, and the bidirectional network channels can be modeled with a complete graph [Har69, p. 16] whose vertices are the raw input stream generating devices and the machines which host entry components and/or processors and whose (non-directed) edges are the bidirectional network channels (see Figure 5.9).

Note that we assume neither that the transmission delay of a network channel is constant over time nor that a network channel provides any ordering guarantees. Moreover, we do not assume that different network channels exhibits the same properties (e.g., equal transmission delays). The only assumption we pose on these network channels is that there are no long-lasting network partitions or other conditions that prevent devices from communicating and thus that every data stream element which is attempted to be transferred via a network channel eventually (potentially after multiple transferring attempts if packets can be lost) arrives and is thus delivered at least once. Again, we argue that this assumption is valid in the team sports analysis scenarios our work focuses on, such as the real-time football analysis scenario described in Section 2.1.

We want to highlight that the absence of restrictive assumptions on the network channels includes also the used communication model. On the one hand, components can send or forward data stream elements to other components and thus trigger the communication. This style of communication is denoted as *push-based* [FZ98] as the producing component pushes the data stream elements to the consuming component. On the other hand, the consuming component can trigger the communication by means of pulling the data stream elements from the producing component which provides or publishes the element. This opposite style of communication is denoted as *pull-based* [FZ98]. There are data stream analysis systems, such as Storm [TTS+14] and OSIRIS-SE [Bre08], whose components communicate according to the push-based communication model as well as data stream analysis systems, such as PAN [PGS16b], whose components communicate according to the pull-based communication model.[9] Moreover, there are also data stream analysis systems which follow a hybrid approach. For instance, in Samza producing components send and thus push new data stream elements to a broker of a message queue or a publish/subscribe system (e.g., Kafka [KNR11]) which serves as a proxy between producing and consuming components, and consuming components pull new data stream elements from this proxy [Sam17k; KK15].[10] As worker-based data stream analysis systems can make use of a very diverse proxies we do not define them in detail in our system model. We only define a proxy to be an additional component which is deployed on a dedicated machine which is connected with bidirectional network channels to every other machine and thus joins the complete graph or on one of the machines which hosts entry components and/or processors. A potential use case of such proxies – assigning sequence numbers to elements of a data stream partition if they are generated and emitted by multiple components – will be presented in Section 6.2.1.1.

In our model we describe all communication according to the push-based communication model. However, we do so only in order to keep the model simple and clear. The statements we have made in the previous sections and which we will make in the remainder of the model part are also valid if the communication is performed according to the pull-based communication model or according to hybrid approaches. This is the case since the major difference between the two opposite communication models which has to be regarded in

[9] As described in [TTS+14], Storm's entry components (called spouts) pull raw input stream elements from the input sources. Similarly, in OSIRIS-SE raw input stream elements are pulled from the raw input stream generating devices [Bre08, p. 100]. However, in both systems the data stream element transfer between the workers/processors and between the workers/processors and external consumers is performed in a purely push-based fashion. In contrast, PAN assumes that the raw input stream generating devices send and thus push raw input stream elements to its forwarder workers (i.e., to PAN's entry components) but performs the data stream element transfer between the workers and between the workers and external consumers in a purely pull-based fashion [PGS16b]. We argue that it is reasonable to classify the communication model of a data stream analysis system based on how data stream elements are transferred between workers/processors and thus to classify Storm's and OSIRIS-SE's communication model as push-based and PAN's communication model as pull-based instead of classifying all three as hybrid.

[10] More precisely, [KK15] which describes the interaction of Samza and Kafka states that "[i]n essence, a Samza job consists of a Kafka consumer, an event loop that calls application code to process incoming messages, and a Kafka producer that sends output messages back to Kafka" [KK15] and that a Kafka consumer "polls the brokers to await the arrival of messages" [KK15].

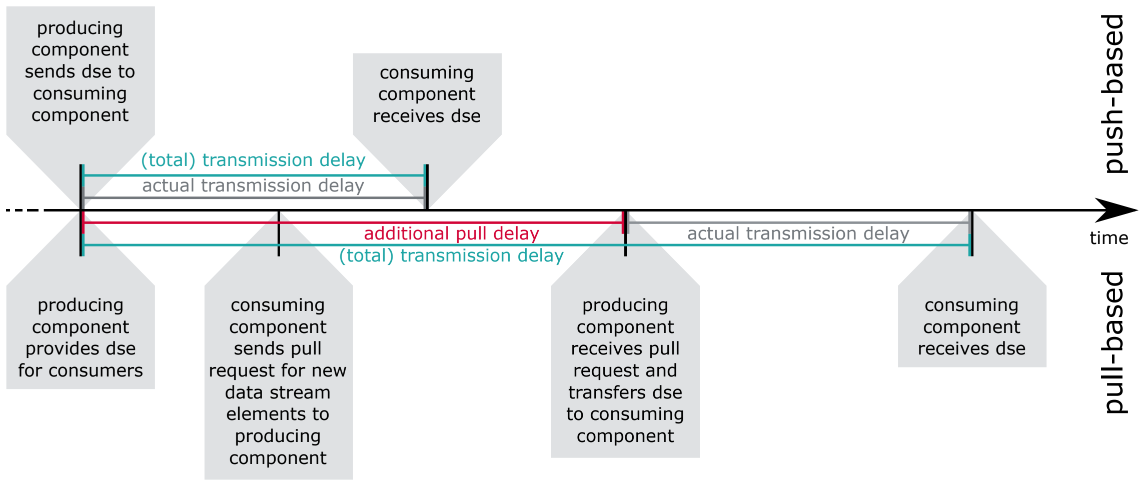

producing component sends dse to consuming component
consuming component receives dse
(total) transmission delay
actual transmission delay
push-based
time
additional pull delay
actual transmission delay
(total) transmission delay
producing component provides dse for consumers
consuming component sends pull request for new data stream elements to producing component
producing component receives pull request and transfers dse to consuming component
consuming component receives dse
pull-based

Figure 5.10 **Additional Pull Delay.** The gray boxes visualize the actions of a producing component and a consuming components on a temporal scale. The actual transmission delay as well as the additional pull delay introduced by using the pull-based instead of the push-based communication model to transfer a data stream element (*dse*) directly (i.e., without an intermediate proxy) from the producing component to the consuming component is highlighted in gray and red, respectively. The aggregated delay which we regard as the (total) transmission delay in our model is highlighted in mint.

our model can be easily eliminated. More precisely, if the pull-based communication model is used the consuming component does not necessarily pull the data stream element immediately when the producing component has provided it and the pull request has to be transfered before the actual data stream element can be transfered. In contrast, if the push-based communication model is used the data stream element is immediately transfered as the producing component triggers the communication. Hence, using the pull-based communication model introduces an additional pull delay between the data stream element provision and the actual pull as well as between the pull and the data stream element transmission (see Figure 5.10). By means of regarding the delayed data stream element transmission as a prolonged transmission and thus regarding this additional pull delay as a part of the transmission delay the major difference between the two opposite communication models vanishes. For instance, in our stream time model (see Chapter 6) it does not matter if a consuming component receives a data stream element after the producing component has sent (and thus pushed) it or after the producing component has provided it and the consuming component has pulled it. Solely the time of arrival matters and since the only factor of the data stream element transfer that influences the time of arrival is the transmission delay on which we pose no strict assumptions and which we regard to include the potential pull delay this time of arrival depends on the same factors regardless which communication model is used. Moreover, we want to highlight that all examples and counterexamples which we describe with the push-based communication model in mind and which potentially even pose assumptions which hold only for the push-based communication model, such as that a received data stream element is immediately forwarded, are merely examples. For each example we present in this thesis it is possible to construct a similar example which is based on the pull-based communication model or a hybrid approach.

For the sake of brevity, we denote in the remainder of this thesis the network channel between a machine hosting component A (e.g., an entry component) and a machine hosting component B (e.g., a processor) also as the network channel between A and B. Equally, we denote the network channel between a raw input stream generating device and a machine hosting an entry component also as the network channel between the raw input stream generating device and the entry component. Moreover, in order to describe communication procedures (especially in Chapter 6) concisely, we do not treat sending data stream elements from one component hosted by a certain machine to another component hosted by the same machine as a special case where no network channel is required. Instead, the only difference in this case is that the network channel via which the elements are transferred is only imaginary and thus defined to provide First In First Out (FIFO) guarantees and to do not introduce any latency.

5.5 Global Perspective

So far our system model has only regarded a single data stream analysis system. However, in practice, there might be multiple data stream analysis systems which share input streams and/or machines. In this section, we will discuss the coexistence of such data stream analysis systems and thus change our perspective from a local single data stream analysis system perspective to a global multiple data stream analysis systems perspective.

Multiple data stream analysis systems can share input streams. That is, workers of different data stream analysis systems, or more precisely the processors of these workers, can consume and analyze elements of the same raw input stream. Moreover, albeit uncommon, multiple coexisting data stream analysis systems can share machines. That is, components (i.e., processors and/or entry components) of multiple data stream analysis systems can be deployed on the same machine. This is possible without adding additional separation mechanisms since processors and entry components anyway perform their work independently (e.g., do not share state) regardless of whether they are deployed on the same machine or not and regardless of whether they belong to the same data stream analysis system or not.

Example 5.5 Coexisting Data Stream Analysis Systems

As an example for coexisting data stream analysis systems, assume that there are two abstract sample data stream analysis systems ($dsas_1$ and $dsas_2$) with very simple workflows. Assume that $dsas_1$ comprises a single entry component (ec_1) and two workers (w_A and w_B) with the following input and output stream sets:

$$\begin{aligned} w_A.DS_{in} &= \{ds_S\} \qquad & w_A.DS_{out} &= \{ds_X\} \\ w_B.DS_{in} &= \{ds_X\} \qquad & w_B.DS_{out} &= \{ds_Y\} \end{aligned}$$

Moreover, assume that $dsas_2$ comprises two entry components (ec_2 and ec_3) and a single workers (w_C) with the following input and output stream set:

$$w_C.DS_{in} = \{ds_S, ds_T\} \qquad w_C.DS_{out} = \{ds_Z\}$$

That is, ec_1 receives each element of ds_S and sends it immediately to the correct (w.r.t. the key) processor of w_A. Additionally, each element of ds_S is received either by ec_2 or by ec_3 and forwarded to the correct processor of w_C. In consequence, $dsas_1$ and $dsas_2$ share the input stream ds_S.
Assume that there are two raw input stream generating devices igd_1 and igd_2 which produce and emit all elements of the input streams ds_S and ds_T, respectively. Furthermore, assume that there are two different keys ($Dom_k = \{k_1, k_2\}$) resulting in the following six processors which perform the analysis task defined by the three workers of the two data stream analysis systems in parallel for all elements with key k_1 and key k_2.

$$\begin{aligned} pr_{A1}.w &= w_A \qquad & pr_{A1}.k &= k_1 \\ pr_{A2}.w &= w_A \qquad & pr_{A2}.k &= k_2 \\ pr_{B1}.w &= w_B \qquad & pr_{B1}.k &= k_1 \\ pr_{B2}.w &= w_B \qquad & pr_{B2}.k &= k_2 \\ pr_{C1}.w &= w_C \qquad & pr_{C1}.k &= k_1 \\ pr_{C2}.w &= w_C \qquad & pr_{C2}.k &= k_2 \end{aligned}$$

Figure 5.11 shows a sample deployment of $dsas_1$ and $dsas_2$. The two raw input stream generating devices are independent machines. The three entry components and the six processors are deployed onto three machines. A first machine

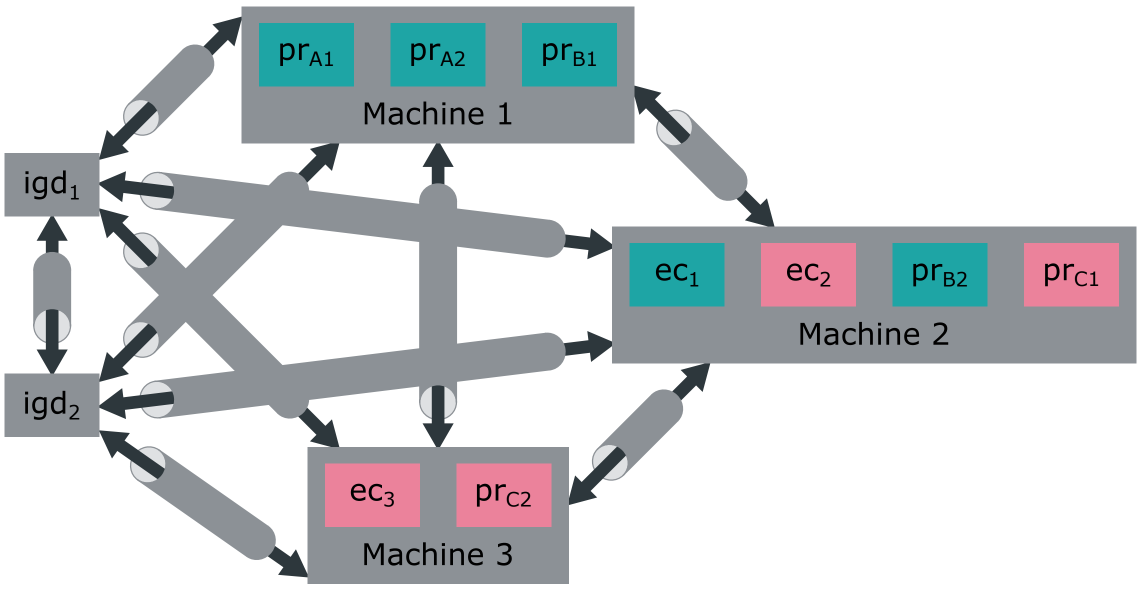


Figure 5.11 **Deployment of Two Coexisting Data Stream Analysis Systems.** The mint boxes illustrate the entry component (ec_1) and the processors (pr_{A1}, pr_{A2}, pr_{B1}, pr_{B2}) of the first data stream analysis system ($dsas_1$). The red boxes illustrate the entry components (ec_2 and ec_3) and the processors (pr_{C1} and pr_{C2}) of the second data stream analysis system ($dsas_2$). The light-gray boxes represent the raw input stream generating devices (igd_1 and igd_2) as well as the three machines which host the entry components and the processors of both data stream analysis systems. The pipes between the light-gray boxes illustrate the bidirectional network channels between the raw input stream generating devices and the machines via which packets containing data stream elements and other information are transferred.

hosts pr_{A1}, pr_{A2}, and pr_{B1}, and thus only components of $dsas_1$. A second machine hosts ec_1, ec_2, pr_{B2}, and pr_{C1}, and thus components of $dsas_1$ and $dsas_2$. A third machine hosts ec_3 and pr_{C2}, and thus only components of $dsas_2$.

In contrast, we define that multiple data stream analysis systems cannot share an output stream. That is, a processor of a worker of a data stream analysis system cannot generate and emit elements of the same event, state, or statistics stream as a processor of a worker of another data stream analysis system. This is even the case if both processors perform the same analysis task and produce the same analysis results. For instance, two football analysis systems which analyze the same match might both involve a pass detection worker and thus comprise a processor assigned to this worker which detects successful passes in this match and emits an event stream element for each detected successful pass event with the match identifier as the key. However, the event stream elements produced by the two processors cannot be emitted in the same successful pass event stream.

Instead, they have to be emitted in two different data stream analysis system specific event streams.

The reason for this is that we have defined each data stream analysis system to have its own isolated workflow which is formed by the workers of the data stream analysis system. More precisely, we have defined a worker (and thus the processors assigned to this worker) to be only able to consume elements of raw input streams which are emitted by raw input stream generating devices as well as elements of event streams, state streams, and statistics streams which are emitted by (processors assigned to) other workers of the same data stream analysis system (see Section 5.2.1). If two workers of two different data stream analysis systems (or more precisely their processors) were able to emit elements of the same event, state, and statistics streams, the workflows of both data stream analysis systems could merge. This is the case since it is impossible to formally define and to guarantee in practice based on the information contained in the elements of an event, state, or statistics stream that (processors assigned to) other workers of the two data stream analysis systems which consume elements of the event, state, or statistics stream consume only those elements of the stream which have been emitted by (processors assigned to) workers of the same data stream analysis system as, according to our data model, the data stream element tuple (see Definition 4.2) does not comprise an attribute that defines the component which has emitted the element.

Example 5.6 Merged Workflows without Output Stream Sharing Restriction

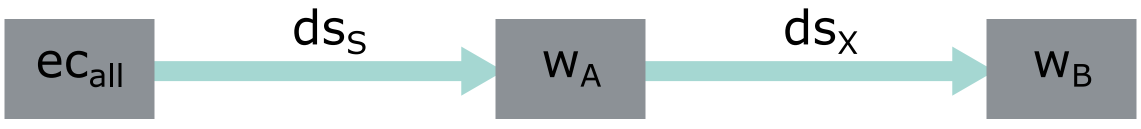
To give an example that the workflows of multiple data stream analysis systems can merge if they are allowed to share output streams we drop the output stream sharing restriction.

Assume that there are two abstract sample data stream analysis systems ($dsas_1$ and $dsas_2$) with very simple workflows. As illustrated in Figure 5.12(a), $dsas_1$ comprises two workers (w_A and w_B) with the following input and output stream sets:

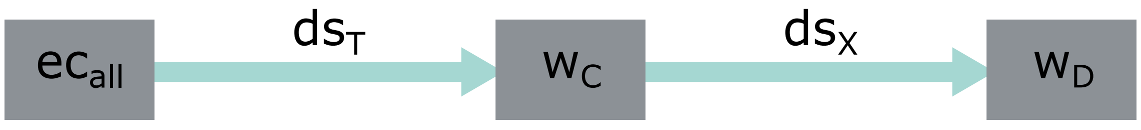
$$w_A.DS_{in} = \{ds_S\} \qquad w_A.DS_{out} = \{ds_X\}$$
$$w_B.DS_{in} = \{ds_X\} \qquad w_B.DS_{out} = \{ds_Y\}$$

Moreover, as depicted in Figure 5.12(b), $dsas_2$ comprises two workers (w_C and w_D) with the following input and output stream sets:

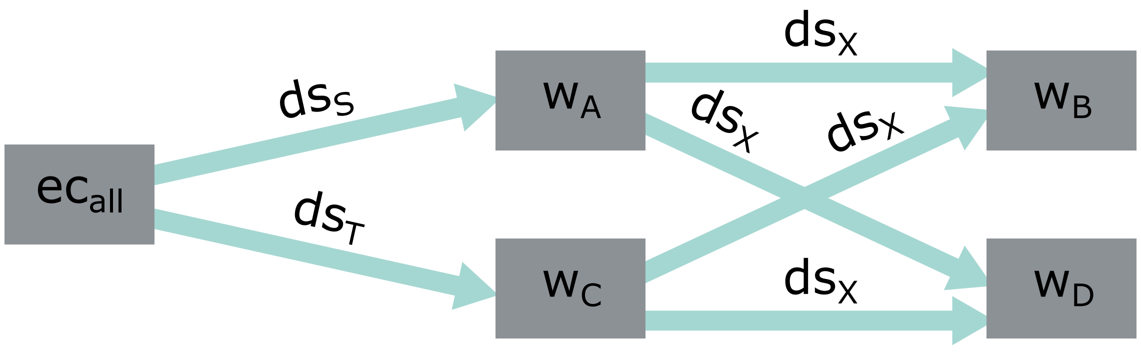
$$w_C.DS_{in} = \{ds_T\} \qquad w_C.DS_{out} = \{ds_X\}$$
$$w_D.DS_{in} = \{ds_X\} \qquad w_D.DS_{out} = \{ds_Z\}$$



(a) Workflow of $dsas_1$



(b) Workflow of $dsas_2$



(c) Joint workflow of $dsas_1$ and $dsas_2$

Figure 5.12 Merged Workflows without Output Stream Sharing Restriction. The gray boxes represent the vertices of the graph which models the workflow that is formed by the given workers, i.e., by w_A and w_B in (a), by w_C and w_D in (b), and by all four in (c). The mint arrows visualize the directed edges of the graph, i.e., the data streams (ds_S, ds_T, and ds_X) emitted by one and consumed by another vertex.

In consequence, $dsas_1$ and $dsas_2$ share the output stream ds_X.
It is impossible to formally define and to guarantee in practice based on the information contained in the elements of ds_X that w_B consumes and processes only those elements of ds_X which have been emitted by w_A but not those which have been emitted by w_C and that w_D consumes and processes only those elements of ds_X which have been emitted by w_C but not those which have been emitted by w_A. The same is true on a processor level when regarding data stream partitions no matter how many keys exist. Hence, the workflows of both data stream analysis systems loose their isolation and merge to one joint workflow illustrated in Figure 5.12(c).

Formally, we define the additional workflow isolation constraints which arise when regarding multiple data stream analysis systems as follows:

Definition 5.7 Workflow Isolation Constraints

The input stream sets and the output stream sets of all workers of all data stream analysis systems have to fulfill the following constraints:

1. Each worker contains only event, state, and statistics streams in its input stream set which are solely contained in the output stream sets of workers of the same data stream analysis system.

$$\forall w_1 \in \widehat{W} : \\ (\forall ds \in w_1.DS_{in} \text{ with } ds.cat \in \{\text{“event”}, \text{“state”}, \text{“statistics”}\} : \\ \Big(\forall w_2 \in \widehat{W} \text{ with } ds \in w_2.DS_{out} : w_1.dsas = w_2.dsas\Big)\Big)$$

2. Each event, state, and statistics stream is only contained in the output stream sets of the workers of a single data stream analysis system.

$$\forall ds \in \widehat{DS} \text{ with } ds.cat \in \{\text{“event”}, \text{“state”}, \text{“statistics”}\} : \\ \Big(\nexists w_1, w_2 \in \widehat{W} : w_1.dsas \neq w_2.dsas \wedge ds \in w_1.DS_{out} \wedge ds \in w_2.DS_{out}\Big)$$

The first condition ensures that every event, state, or statistics stream consumed by a worker (and thus by the processors assigned to this worker) is solely emitted by (processors assigned to) workers of the same data stream analysis system.[11] This guarantees that no output stream which is consumed by (processors assigned to) any worker is shared with other data stream analysis systems or even solely emitted by (processors assigned to) workers of another data stream analysis system which would cause the workflows to merge as well, and thus that the workflows of multiple data stream analysis systems do not merge but are completely isolated. The second constraint further guarantees that also event, state, and statistics streams which are not consumed by any worker (or more

[11] Note that we do not require that every event, state, and statistics stream contained in the input stream set of a worker is emitted by at least one worker (or more precisely by at least one processor) of the same data stream analysis system. We only required with the well-formation constraints of a workflow (see Definition 5.5) that at least one data stream contained in the input stream set of a worker without a timer is emitted by any raw input stream generating device or contained in the output stream set of any alive worker of the same data stream analysis system. Otherwise the worker is regarded as dead and the workflow is regarded as ill-formed.

precisely the processors of any worker) are not shared with other data stream analysis systems. This guarantees that external consumers which consume an event, state, or statistics stream consume only the analysis results of a single data stream analysis system. We argue that this is rational as it provides external consumers full control about which analysis results, or more precisely whose analysis results, it consumes by means of choosing which event, state, and statistics streams to consume. In the remainder of this thesis, we assume that these workflow isolation constraints are fulfilled, unless otherwise specified.

An easy approach to fulfill these additional workflow isolation constraints which works for all technical implementations and deployments, even if the same stream communication infrastructure is used by multiple data stream analysis systems (e.g., if multiple data stream analysis systems use the same Kafka brokers), is to leverage the names of the data streams. In order to ensure that elements of every event, state, and statistics stream are emitted only by processors of workers of a single data stream analysis system and thus that multiple data stream analysis systems do not share output streams a unique identifier of the data stream analysis system[12] has to be appended to the name ($ds.name$) of every data stream (ds) contained in the output stream set of every worker of the data stream analysis system. That is, for instance, the names of the successful pass event stream produced by the processors of the workers of one data stream analysis system ($dsas_1$) and by the processors of the workers of another data stream analysis system ($dsas_2 \neq dsas_1$) can be "successfulPassEvent-1" and "successfulPassEvent-2", respectively. In order to further ensure that the processors of the workers of a data stream analysis system consume only event, state, and statistic streams which are emitted by processors of workers of the same data stream analysis systems, the input stream set of every worker has to be restricted to contain only event, state, and statistics streams whose name is appended with the correct data stream analysis system identifier. That is, for instance, a pass sequence analysis worker of $dsas_1$ is only allowed to contain the event stream with the name "successfulPassEvent-1" but not the event stream with the name "successfulPassEvent-2" in its input stream set.

With the first constraint of Definition 5.7, we have defined a worker in a way that its processors cannot directly consume event, state, and/or statistics

[12] We have relinquished to add a formal definition of a data stream analysis system as such a definition is not needed for our model. However, for the purpose of constructing a formal foundation for such a unique identifier one could define a data stream analysis system $dsas$ as a singleton $\langle dsasid \rangle$ where $dsasid$ is a unique data stream analysis system identifier (i.e., $UID(\widehat{DSAS}) = \{dsasid\}$).

stream elements which have been emitted by processors of workers of another data stream analysis system. For instance, the processors of the workers of an advanced football analysis system cannot directly consume the successful pass event stream elements emitted by the processors of a worker of a simple football analysis system. This restriction is required to keep the workflows of different data stream analysis systems isolated. However, there might be scenarios, such as this split football analysis system scenario, in which the processors of a worker of a data stream analysis system need to consume the elements emitted by the processors of a worker of another data stream analysis system. In order to support that, i.e., in order to enable the processors of the workers of a data stream analysis system ($dsas_1$) to consume and further process the elements of a certain event, state, or statistics stream which is emitted by the processors of the workers of another data stream analysis system ($dsas_2$), one has to leverage a dedicated raw input stream generating device which consumes the event, state, or statistics stream of $dsas_2$ and emits the contained data in an element of a raw input stream (with a different name) which $dsas_1$ consumes.[13]

[13] As defined in Definition 4.2, each data stream element has an explicit timestamp to which we will refer later as generation timestamp (see Section 6.1.1). Note that these generation timestamps can only be copied without further considerations if all raw input stream elements which $dsas_1$ consumes are converted event, state, or statistics stream elements from $dsas_2$. Alternatively, the raw input stream generating device can assign new generation timestamps using its local (synchronized) clock. In all other situations it has to be ensured that the generation timestamps of all raw input stream elements which $dsas_1$ consumes are from the same consistent generation time space (see Section 6.1.1) as otherwise the consistent generation time space guarantee for $dsas_1$ would be corrupted.

6

Stream Time Model

Temporal information is a very important aspect in team collaboration analysis. For instance, reliably detecting passes in a football match based on data stream elements representing lower level events (e.g., ball kicks and ball possession changes) requires besides comparing other information of these lower level events also checking which moments in time the lower level events reflect.

In this chapter, we will present, define, and compare different time notions as well as the orderings introduced by the sequence numbers contained in and by the timestamps which can be assigned to the data stream elements. In doing so, we will use recent literature which discusses time notions in data stream analysis systems, namely [ABC+15], [CKE+15], [ATM+17], and [Fli18], as the starting point. However, to the best of our knowledge, the detailed description and comprehensive discussion of the time notions and orderings that we will present in this chapter in the context of our model goes far beyond existing literature. For instance, there is no work yet that analyzes for each pair of orderings if they are guaranteed to be consistent or not, that expresses the findings in formal theorems, and that proves these theorems as we will do in Section 6.2.5. Moreover, we will introduce a novel concept of simultaneousness in Section 6.3.

6.1 Time Notions

In Chapter 4 we have already presented that every data stream element comprises a single timestamp. However, this timestamp contains only temporal information regarding one notion of time. There are several time notions and thus also a multitude of timestamps. More precisely, different entities can assign different timestamps to the same data stream element specifying different moments in time with respect to different time spaces.

The time space with respect to which a timestamp specifies a moment in time affects if the timestamp can be compared with another timestamp. Formally, we define a time space as follows:

Definition 6.1 Time Space

A time space $\mathfrak{T}$ is a space with respect to which a timestamp can specify a moment in time. Each timestamp specifies a moment in time with respect to exactly one time space. All timestamps which specify a moment in time with respect to the same time space are created by the same set of synchronized clocks and thus comparable.

In this section, we will (i) identify different time notions and define which moment in time a timestamp of a certain time notion specifies and with respect to which time space this moment is specified, (ii) present how and by which entities timestamps of a certain time notion are created and assigned, (iii) investigate and compare properties of the time notions and the corresponding timestamps, and (iv) relate different timestamps assigned to the same data stream element to each other.

6.1.1 Generation Time

According to our model, every data stream element contains temporal information in form of a globally unambiguous timestamp $ts \in \mathbb{N}_0$ that specifies the time of generation (see Definition 4.2). That is, the timestamp specifies with respect to a consistent generation time space when a raw input was generated (e.g., when a sensor value was measured[1]), when an atomic event occurred, or to which time the latest update of a non-atomic event, a state, or a statistic refers. We refer to the time notion introduced by these timestamps as *generation time* as done in [Bre08], [ABB$^+$13], and [RZG$^+$18] and to the timestamps as generation timestamps. Formally, we define the generation time of a data stream element as follows:

[1] As a matter of course, there are also sensors which do not perform their measurements atomically but by means of monitoring something and aggregating values over a certain time interval (e.g., heart rate sensors monitor the body for some time to perform a new heart rate measurement). However, since we are not interested in modeling the measurement procedure of sensors (or the value generation procedure of other raw input stream generating devices), we regard every sensor to perform its measurements atomically by simply defining the end of the measurement interval as the point in time at which the sensor value was measured.

Definition 6.2 Generation Time

The generation timestamp *ts* of a data stream element specifies the time of generation with respect to the consistent generation time space $\mathfrak{T}_{gen}$.
Depending on the category and atomicity of the data stream element the time of generation is the moment in time when the raw input was generated, when the atomic event occurred, or to which time the latest update of the non-atomic event, the state, or the statistics refers.

Literature Discussion 6.1 Generation Time

In recent literature that discusses time notions in data stream analysis systems, the term *event time* is used to denote the explicit timestamp assigned to each data stream element when it is generated (e.g, by a sensor) [ABC+15; CKE+15; ATM+17; Fli18]. This term is used since every input stream element is regarded to reflect an event and the timestamp of the data stream element is assumed to specify the time when the event occurred [ABC+15].
Although this matches the generation semantics of the timestamp we assign in our model to every data stream element we argue that using the term event time may cause confusion. This is due to the fact that we distinguish in this thesis between raw input streams, event streams, state streams, and statistics streams, and thus do not regard every data stream element to reflect an event. More precisely, we classify only events detected by a data stream analysis system as events but sensor measurements as raw inputs for the data stream analysis system.
Hence, in order to avoid confusion, we use the term generation time as done in [Bre08], [ABB+13], and [RZG+18]. Moreover, in contrast to [ABC+15], [CKE+15], [ATM+17], [Fli18], [Bre08], [ABB+13], and [RZG+18] which do not specify how (and partially even if) the generation timestamps of data stream elements which are generated by a processor of a worker of the data stream analysis system are obtained[2], we specify in detail how the generation timestamp of every event, state, and statistics stream element is created according to our model.

[2] The authors of [ABC+15] and [ABB+13] merely present an example how the timestamps of data stream elements resulting from a window operation can be set and some minimal requirements for the timestamp of a window operation result in particular (in [ABC+15]) and an arbitrary new output stream element in general (in [ABB+13]). However, they do not define an explicit rule which exact generation timestamp an event, state, or statistics stream element has to be assigned.

As we will discuss in detail in Section 6.1.4, it is impossible to compare timestamps which specify moments in time with respect to different time spaces. In consequence, since we want the generation timestamps of all data stream elements to be comparable for analysis purposes we require all generation timestamps to be from one consistent generation time space ($\mathfrak{T}_{gen}$). In order to obtain generation timestamps from one consistent generation time space we assume all raw input stream generating devices to have synchronized clocks. This is in accordance with the statement given in [Bre08, p. 42] that using the generation timestamps when processing data stream elements from different raw input stream generating devices requires synchronizing their clocks. Although discussing the problem how to synchronize clocks is out of the scope of this thesis, we want to mention that there are a multitude of approaches and standards, such as WeakTrueTime [Pro14], the Network Time Protocol (NTP) [MMB+10], and the Precision Time Protocol (PTP) [Eid08], which can be used to synchronize the clocks of the raw input stream generating devices. Moreover, we want to highlight at this point that this is the only inevitable synchronization requirement which we pose in our model. The clocks of all other components, i.e., the clocks of the entry components and the processors, do not have to be synchronized (see Section 6.1.3 for more details). If we nonetheless pose a restrictive assumption on the synchronization of the clocks of the entry components and/or the processors (e.g., that the clocks of all processors of a data stream analysis system are synchronized) in this chapter, we do so only temporarily to build a foundation for a theoretical discussion.

Based on the assumption that all raw input stream generating devices have synchronized clocks, we define that generation timestamps are created and assigned as follows: For raw input stream elements generated by raw input stream generating devices with synchronized clocks the generation timestamp is created using one of these clocks when generating the input data. For instance, the generation timestamp of a player sensor input stream element is the measurement time of the sensor (see Example 4.1). In a data stream element produced by a data stream analysis system the generation timestamp is the generation timestamp of and thus inherited from the last data stream element that has contributed to the data of the produced element where last refers to the ordering implied by their generation timestamps. That is, the generation timestamp of an event stream element, a state stream element, and a statistics stream element is the largest generation timestamp of all data stream elements that have contributed to the event (event update in case of a non-atomic event), the state, and the statistic, respectively. As this is the case for all event, state, and statistics stream elements, the generation timestamp of every event, state, and statistics stream element can be eventually traced back to the generation timestamp of a raw input stream element. Consequently, the synchronized clocks of the raw input stream generating devices ensure that the generation timestamps of all data stream elements consumed or produced by a data stream analysis system are from the same consistent generation time space.

Example 6.1 Generation Timestamps in Football Analysis

Figure 6.1 illustrates the generation timestamp creation and inheritance for three sample data stream elements consumed or emitted by a small football analysis system *dsas* with two workers, namely a ball possession change detection worker and a pass detection worker. For the sake of keeping the example simple we assume that there is only a single key and thus only a single processor assigned to every worker. Moreover, we assume in this example that all processors process all input stream elements no only in correct order with respect to their sequence numbers as defined in (see Section 5.3.2) but also in correct order with respect to their generation timestamps even across data stream partitions and thus that the last observed generation timestamp is always the largest yet observed generation timestamp.

The generation timestamp of the successful pass event stream element emitted by processor pr_2 (of the pass detection worker) is the generation timestamp of the ball possession change event stream element which has indicated the receive of the pass and thus triggered the detection of the successful pass event at pr_2.

The generation timestamp of a ball possession change event stream element emitted by processor pr_1 (of the ball possession change detection worker) in turn is the generation timestamp of the ball position input stream element that has triggered the detection of the ball possession change event at pr_1.

The ball position input stream element is a raw input stream element. That is, its generation timestamp is created by means of the synchronized clock of the raw input stream generating device *igd* that has generated the ball position input stream element and thus from the consistent generation time space. In consequence, also the generation timestamp of the ball possession change event stream element and the generation timestamp of the successful pass event stream element are from the consistent generation time space.

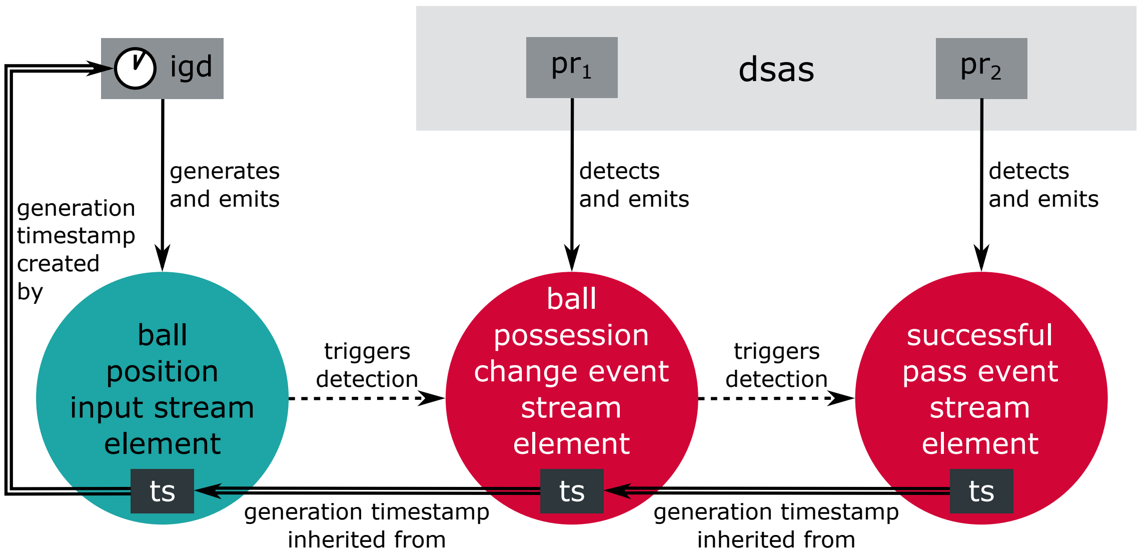

igd
dsas
pr₁
pr₂
generates and emits
detects and emits
detects and emits
generation timestamp created by
ball position input stream element
triggers detection
ball possession change event stream element
triggers detection
successful pass event stream element
ts
ts
ts
generation timestamp inherited from
generation timestamp inherited from

Figure 6.1 Generation Timestamps in Football Analysis. The large dots represent the three sample data stream elements. The event stream elements are colored in red and the raw input stream element is colored in mint. The dark-gray *ts*-boxes in the dots represent the generation timestamps of the data stream elements. The light-gray box represents the football analysis system (*dsas*). The gray boxes represent the raw input stream generating device (*igd*) with its synchronized clock and the processors (pr_1 and pr_2). The normal arrows show the actions which the raw input stream generating device and the processors perform. The dotted arrows visualize which data stream element triggers the detection of which other data stream element and the double arrows show how the generation timestamps are created by the synchronized clocks or inherited from other data stream elements.

At the first glance, our assumption that all raw input stream generating devices have synchronized clocks might seem to be a very strong limitation. However, in the team sports analysis scenarios our work focuses on it is not. For instance, the workflow of our real-time football analysis application analyzes a match on the basis of raw input stream elements which contain positions of the players and the ball and a single raw input stream element which contains some meta-data about the match (see Section 9.2). As presented in Section 2.1, the positions are generated by means of sensor-based and/or video-based tracking systems. In many of these tracking systems all data are processed by a single machine or multiple well-connected and time-synchronized machines in order to calculate positions. For instance, in the LPM system [SPF04] multiple receivers forward time-of-arrival information to a single device which computes positions of players carrying transmitters by means of time-of-arrival differences. In order to additionally obtain ball positions the LPM system can be complemented by a dedicated video-based ball tracking system which is time-synchronized with the

LPM system device [Inm20]. Hence, the raw input stream elements containing the positions are generated either by a single device if only players are tracked or by multiple devices with synchronized clocks if also the ball is tracked. The additional raw input stream element which contains some match metadata can be either generated by a raw position stream element generating device or by an additional device whose clock is synchronized with the raw position stream element generating devices. In consequence, all raw input stream elements which the workers of our real-time football analysis application consume are generated by raw input stream generating devices with synchronized clocks.[3]

Moreover, if raw input stream elements generated by some low-level sensor devices whose clocks cannot be synchronized should be analyzed, the clock synchronization can be rectified by means of adjusting the generation timestamps contained in the raw input stream elements with respect to each other before they are actually analyzed. That is, the generation timestamps of all raw input stream elements have to be transformed into a common and consistent generation time space. This transformation has to be performed in a dedicated transformation component which is not part of the data stream analysis system. Moreover, in our model we do not regard the low-level sensor but the dedicated transformation component as the raw input stream generating device which generated the raw input stream elements. This is necessary as adjusting the generation timestamps in another component, such as the entry component or the processor of the first worker of the data stream analysis system, would violate our data model in which we define that data stream elements are immutable.

We want to highlight that generation timestamps are not agnostic to environmental conditions although they are either directly created and assigned by a raw input stream generating device or inherited from a raw input stream element and thus never created using the local clock of any component but the raw input stream generating devices. First, the fact from which input stream element the generation timestamp of a newly generated event, state, or statistics stream element is inherited can depend on environmental conditions. This is even true if the elements of each input stream partition are not only processed in correct order with respect to their sequence numbers as defined in Section 5.3.2 but

[3] Note that this is only the case if all matches which a deployment of our real-time football analysis application analyzes are tracked with the same tracking system. This is the case if there is a separate on-site real-time football analysis application deployment for each football field or if there is a global tracking system (e.g., a centralized video-based tracking system which receives and processes the videos of all matches of a football league). Otherwise, i.e., if the same real-time football analysis application deployment should be used to analyze matches which have been tracked with different tracking system, all devices of all tracking systems need to have synchronized clocks.

additionally in correct order with respect to their generation timestamps since the sequence number ordering and the generation time ordering are consistent (see Section 6.2). For instance, if the detection of a certain event is triggered by the next processed element of one of two input streams (e.g., detecting a match pause event can be triggered by a field leave event stream element or a foul event stream element) the fact which input stream element is processed first determines the generation timestamp of the emitted event stream element. Since there is no regulation in which order elements of different data stream partitions are processed (see Section 5.3.2), the fact which input stream element is processed first might depend on environmental conditions like network properties (e.g., message ordering guarantees or transmission delay characteristics) and processing conditions (e.g., system overloads or processing delays). Moreover, even if there is only a single input stream partition, the fact if the next or only a later input stream element (which has a greater generation timestamp) triggers the detection of an event can depend on the moment at which they are processed which in turn depends on environmental conditions (see Section 6.1.3). For instance, if the football match is divided into non-overlapping processor clock based 30 seconds intervals the generation timestamp of the first high pass frequency event which informs about the fact that more than five successful passes have been played in a 30 seconds interval might be the generation timestamp of the last successful pass event stream element of the first interval or the generation timestamp of the last successful pass event stream element of the second interval depending on the fact if the sixth successful pass event stream element of the match with the generation timestamp 29983 was processed in the first processor clock based 30 seconds window or in the second processor clock based 30 seconds window. Second, also the generation timestamps of raw input stream elements are not agnostic to environmental conditions since the moment of raw input data generation at the raw input stream generating device can depend on environmental conditions. For instance, a sensor might perform the next measurement a few milliseconds later than planned due to an overload situation.

Moreover, even if the generation timestamps were agnostic to environmental conditions using them as time indicators for time-dependent analyses and thus performing the analyses with generation time semantics (originally called "event time semantics" in [ATM+17]) would only ensure deterministic results if all data stream elements were processed by every processor in correct order with respect to their generation timestamps [ATM+17]. However, as we will present and discuss in more detail in Section 6.2.5, data stream elements might be processed out-of-order regarding their generation timestamps because of environmental conditions. This cannot be solved by reordering the data stream elements since reordering data stream elements with respect to their generation timestamps conflicts with the requirement of our system model that data stream elements are processed sequentially with respect to their sequence numbers and is thus not supported. As it usually depends on the environmental conditions if the order in which data stream elements are processed is consistent with how they are ordered by means of their generation timestamps (see discussion about consistency between sequence number ordering and generation time ordering in Section 6.2.5) and if the orderings differ how they differ, using generation timestamps as time indicators for time-dependent analyses does not guarantee full determinism in general.

Example 6.2 Non-Deterministic Generation Time Window Statistics

As an example which shows that using generation timestamps does not guarantee deterministic results of time-dependent analyses if data stream elements are not guaranteed to be processed in correct order with respect to their generation timestamps, assume that there is a processor of a worker which computes statistics, such as the fitness statistics presented in Example 4.5, over generation time windows. Moreover, assume that this processor first emits a statistics stream element after processing a certain data stream element since the generation timestamp of this data stream element indicates that a time window has closed, and then consumes and processes another data stream element which is late with respect to its generation timestamp and which is within the time window for which the statistics stream element has already been emitted.
In consequence, the statistics stream element emitted by the processor and thus the produced result is not correct as it contains wrong data or more precisely statistical values which do not reflect the information contained in the late data stream element.[4]
Since different data stream elements can be late, the emitted statistics stream element can be incorrect in different ways. For instance, the average heart rate in a fitness statistics stream element can be too low if a player sensor input stream element with a high heart rate value was late and too high if a player

[4] An alternative approach presented in [ABC+15] would be to enable the processor to emit an update of the already emitted statistics stream element. However, since this would pose the requirement that processors of subsequent workers and external consumers are able to handle this, we do not support this in our model.

sensor input stream element with a low heart rate value was late. Hence, the statistics the processor calculates and emits are not deterministic.

6.1.2 Production and Emission Time

As we have explained in Section 6.1.1, we acquire the generation timestamp of event, state, or statistics stream elements in our work by inheriting the largest generation timestamp of all data stream elements which have contributed to the atomic event detection, to the non-atomic event information update, to the state calculation, or to the statistics generation. That is, the generation timestamp does not specify the moment when the event, state, or statistics data shipped in a data stream element were actually produced, i.e., when an atomic event was actually detected or when the information of a non-atomic event, a state, or a statistic was actually updated, at a processor. Moreover, the generation timestamp does not specify the moment when the data stream element containing the event, state, or statistics data was emitted by a processor. In this thesis we introduce two novel time notions, the *production time* and the *emission time*[5], to discuss these two moments in time when regarded with respect to the consistent generation time space $\mathfrak{T}_{gen}$. More precisely, we formally define the production timestamp and the emission timestamp of a data stream element as follows:

Definition 6.3 Production Time

The production timestamp ts_{prod} of an event, state, or statistics stream element specifies the moment in time with respect to the consistent generation time space $\mathfrak{T}_{gen}$ at which the data shipped in the element were produced or updated last at a processor.
Raw input stream elements do not have a production timestamp as their data are not produced at a processor.

[5] If pull-based communication is used (instead of push-based communication), there is no emission time but a *provision time* (i.e., moment when the data stream element is provided for pull requests) and a consumer-specific *pull time* (i.e., moment when the pull request is answered by actually transferring the data stream element). As mentioned in Section 5.4.2, we describe all communication according to the push-based communication model. In consequence, we refrain from modeling these time notions. We argue that this is reasonable although we do not restrict components to communicate in a push-based fashion since, as we will show below for emission time, both time notions are only theoretical concepts but unavailable in practice. However, note that it would have been also possible to show the potential unintuitiveness and the advantageousness of the generation time by means of comparing the provision time and/or the pull time with the generation time.

Definition 6.4 Emission Time

The emission timestamp ts_{emis} of an event, state, or statistics stream element specifies the moment in time with respect to the consistent generation time space $\mathfrak{T}_{gen}$ at which the element was emitted by a processor.
Raw input stream elements do not have an emission timestamp since they are not emitted by a processor.

Example 6.3 Generation, Production, and Emission Time

Figure 6.2 illustrates the generation time, the production time, and the emission time of a sample event, state, or statistics stream element. Assume that there is a raw input stream generating devices *igd* which generates after 173 milliseconds some raw input data and emits them immediately in a new element dse_A of the raw input stream ds_{in}. Further assume that there is a data stream analysis system *dsas* with an entry component *ec* that receives this raw input stream element and immediately forwards it to a processor *pr* which consumes the element and uses it to produce some event, state, or statistics data. Assume that the transmission delay between *igd* and *ec*, the transmission delay between *ec* and *pr*, and the time *pr* processes the raw input stream element before producing the event, state, or statistics data sum up to 31 milliseconds. Moreover, assume that after the data production *pr* waits for 38 milliseconds and then emits the event, state, or statistics data in a new element dse_1 of the event, state, or statistics stream ds_{out}.
The generation timestamp of dse_A and dse_1 are as follows:

$$dse_A.ts = 173 \qquad dse_1.ts = 173$$

The production timestamp and the emission timestamp of dse_1 are as follows:

$$dse_1.ts_{prod} = 204 \qquad dse_1.ts_{emis} = 242$$

Every event, state, and statistics stream element is generated only by a single processor of a single worker of a single data stream analysis system (see Chapter 5) and uniquely identifiable by its sequence number and the data stream partition it belongs to (see Chapter 4). Even if processors of multiple workers of the same or different data stream analysis systems perform the same analysis task and produce the same analysis result, each processor ships its analysis result in a clearly distinguishable data stream element. This is the case since two

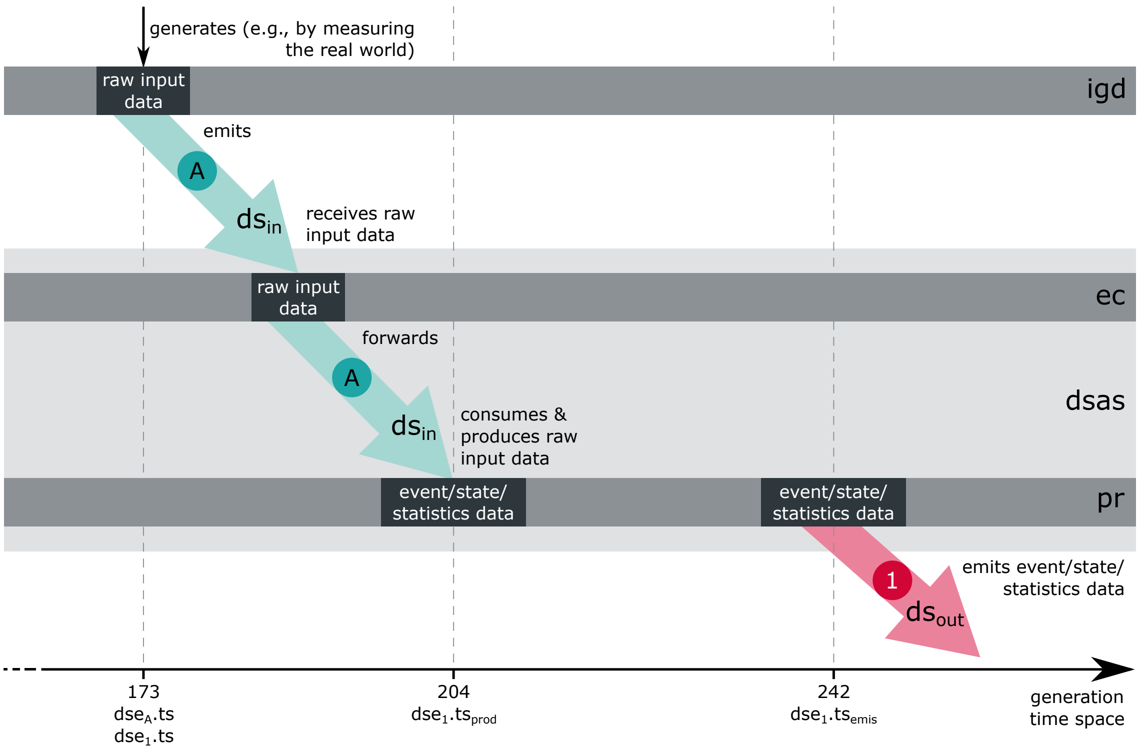

generates (e.g., by measuring the real world)
raw input data
igd
emits
A
ds_in
receives raw input data
raw input data
ec
forwards
A
ds_in
consumes & produces raw input data
dsas
event/state/ statistics data
event/state/ statistics data
pr
emits event/state/ statistics data
1
ds_out
173
dse_A.ts
dse_1.ts
204
dse_1.ts_prod
242
dse_1.ts_emis
generation time space

Figure 6.2 Generation, Production, and Emission Time. Actions of a raw input stream generating device *igd*, an entry component *ec*, and a processor *pr* illustrated on a temporal scale with respect to the generation time space. The light-gray box represents the data stream analysis system (*dsas*). The gray boxes represent the raw input stream generating device, the entry component, and the processor. The dark-gray boxes inside the gray boxes represent data in the memory of the component represented by the gray box. The mint arrow and the mint dot visualize the transmission of the raw input data as dse_A in the raw input stream ds_{in}. The red arrow and the red dot visualize the transmission of the event, state, or statistics data as dse_1 in the event, state, or statistics stream ds_{out}.

processors of two different data stream analysis systems cannot emit the analysis result in an element of the same data stream partition (see Section 5.5) and since all processors which belong to the same data stream analysis system and emit the analysis result as an element of the same data stream partition assign a different sequence number to the element they emit (see Section 6.2.1). Hence, the production timestamp and the emission timestamp of each data stream element are globally unambiguous.

As already mentioned above, synchronized clocks create timestamps of the same time space. Hence, under the assumption that the clocks of the processors of a data stream analysis system are synchronized with the clocks of the raw input stream generating devices (processor-synchronization assumption), these processors can create timestamps from the generation time space and thus acquire the production time and the emission time by means of their local clocks when producing (i.e., creating or updating) the data and emitting the data stream element, respectively.

The later a processor receives and the longer a processor processes a data stream element on the basis of which it produces new or updates existing data, the higher are the production timestamp and the emission timestamp of the data stream element containing this data. In consequence, production timestamps and emission timestamps depend on transmission and processing delays and are thus not agnostic to the environmental conditions.

If there were no transmission and processing delays (zero-delay assumption)[6], the production time, the emission time, and the generation time would match for each event, state, and statistics stream element generated by processors of a data stream analysis system if all data stream elements generated by any processor of the data stream analysis system are emitted immediately after data production. However, even under these strong assumptions, the production time, the emission time, and the generation time of a data stream element would not necessarily match anymore if there are data streams whose elements are not emitted immediately after data production, i.e., if there are for instance non-atomic event, state, and/or statistics streams for which only periodically (e.g., every two minutes based on the local clock of a processor by means of using the timer feature we have presented in the system model) a new data stream element containing the latest event updates, state information, or statistical values is emitted.

Example 6.4 Difference between the Generation, the Production, and the Emission Timestamp under Processor-Synchronization and Zero-Delay Assumptions

Figure 6.3 illustrates the mismatch of the generation timestamp, the production timestamp, and the emission timestamp for fitness statistics stream elements (see Example 4.5) and long time high heart rate event stream elements under processor-synchronization and zero-delay assumptions.
Assume that the fitness statistics are generated and that the long time high heart rate events are detected by a data stream analysis system *dsas* with two workers. For reasons of clarity and comprehensibility we assume that there is

[6] Please note that this assumption does not hold in practice as there are transmission delays in every network and processing delays introduced by every computation. Nevertheless, we argue that making this assumption for theoretical discussions is reasonable.

only a single key and thus only a single processor assigned to every worker processing all elements of every input stream of the worker. Moreover, we regard only the player sensor input stream elements, the fitness statistics stream elements and the long time high heart rate event stream elements for player A4. Assume that processor pr_1 of the first worker consumes elements of the player sensor input stream ds_{in} to calculate the fitness statistics for player A4 and that it emits the latest statistical values periodically every two minutes (based on its local clock using the timer feature) in an element of the fitness statistics stream ds_{st}. Moreover, assume that dse_A, dse_B, and dse_C are the latest elements of ds_{in} for player A4 and that dse_1 is the first element of the ds_{st} for player A4 containing the statistical values of the first two minutes. Further assume that the generation timestamps of dse_A, dse_B, and dse_C are as follows:

$$dse_A.ts = 119'707 \qquad dse_B.ts = 119'807 \qquad dse_C.ts = 119'907$$

The generation timestamp of dse_1 is inherited by dse_C and thus 119'907. The production timestamp of dse_1, i.e, the timestamp specifying the time when the data shipped in dse_1 have been produced (last updated), is also 119'907 due to the zero-delay assumption. However, the emission timestamp of dse_1 is 120'000 and thus does not match the generation timestamp.

$$dse_1.ts = 119'907 \qquad dse_1.ts_{prod} = 119'907 \qquad dse_1.ts_{emis} = 120'000$$

Assume that an atomic long time high heart rate event is detected and emitted immediately as an element of the long time high heart rate event stream ds_{ev} when a fitness statistics stream element which contains statistics for a two minutes interval and whose average heart rate value is above a certain threshold is processed by processor pr_2 of the second worker. Moreover, assume that the average heart rate contained in dse_1 is above the selected threshold. The generation timestamp of the generated long time high heart rate event stream element dse_2 is inherited by dse_1 and thus 119'907. However, even when there are no processing and transmission delays, the production timestamp and the emission timestamp of dse_2 are 120'000.

$$dse_2.ts = 119'907 \qquad dse_2.ts_{prod} = 120'000 \qquad dse_2.ts_{emis} = 120'000$$

At the first glance, the generation timestamp assigned to and thus the explicit timestamp of a data stream element might be perceived as unintuitive. For

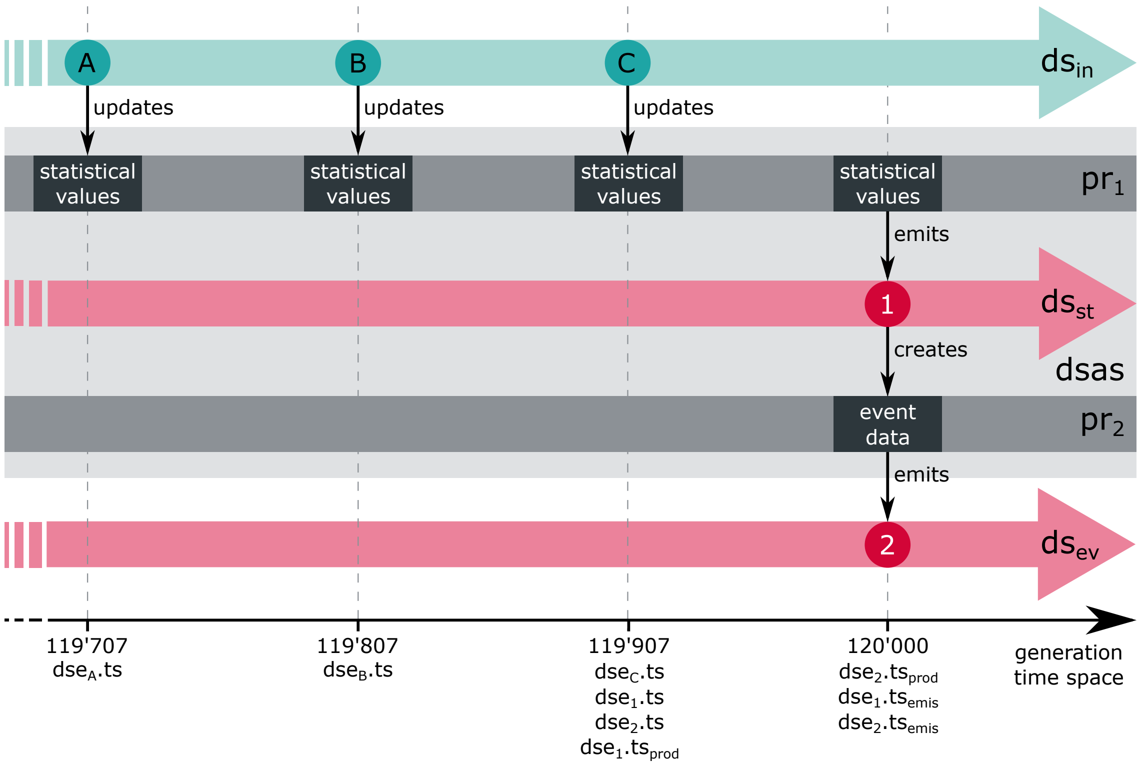


Figure 6.3 Difference between the Generation, the Production, and the Emission Timestamp under Processor-Synchronization and Zero-Delay Assumptions. Data stream elements of the player sensor input stream (ds_{in}) colored in mint as well as the fitness statistics stream (ds_{st}) and the long time high heart rate event stream (ds_{ev}) colored in red illustrated as dots on a temporal scale with respect to the moment in time at which they have been emitted with respect to the generation time space. The light-gray box represents the data stream analysis system ($dsas$), the gray boxes represent the processors (pr_1 and pr_2), and the dark-gray boxes visualize the statistics and event data every time when they are created or updated.

instance, the fact that a data stream element which contains statistics of the first two minutes and which is emitted after two minutes has a timestamp smaller than 120'000 may be perceived as counterintuitive. Therefore, one might come to the conclusion that assigning the emission timestamp instead of the generation timestamp to every data stream element would be the better option.

However, using the generation timestamp (as defined in Section 6.1.1) instead of the production timestamp or the emission timestamp has the great advantage that the clocks of the processors of a data stream analysis system do neither have to be synchronized with the clocks of the raw input stream generating devices, nor among each other. Instead, only the clocks of the raw input stream generating devices have to be synchronized. In practice, the raw input stream generating devices might be managed by a different entity as the processors of a data stream analysis system. This prevents or at least complicates the clock synchronization between processors and raw input stream generating devices. Hence, removing the processor-synchronization assumption and using the generation timestamp has a great impact on the applicability of our model.

In fact, we have made the restrictive processor-synchronization assumption and the unrealistic zero-delay assumption only to enable a discussion of the production time and the emission time on a theoretical level. Outside of this section, we drop these assumptions for our model. In consequence, the production time and the emission time cannot be acquired. Doing so would require either making the processor-synchronization assumption and using the processor clock timestamps directly (as proposed above) or providing a way to transform processor clock timestamps to timestamps from the generation time space which is actually equivalent to synchronizing the processor clock itself. This is the reason why the data stream element tuple (see Definition 4.2) does neither contain a production timestamp nor an emission timestamp.

6.1.3 Processing and Ingestion Time

So far we have discussed three different time notions: Generation time, production time, and emission time. All three have in common that timestamps of these notions specify moments in time with respect to the same consistent generation time space. That is, we have considered timestamps created by one pool of mutually synchronized clocks. In this section, we will present further time notions which encompass timestamps that specify moments in time with respect to other time spaces since they are created by clocks which are not synchronized with the raw input stream generating devices.

As we have defined in Section 5.4.1, every processor which processes data stream elements has a local clock. This clock can be used to assign a timestamp $\tau \in \mathbb{N}_0$ to each data stream element when processing it [ABC+15]. We follow recent literature [ABC+15; CKE+15; ATM+17; Fli18] and refer to this notion of time as *processing time*.

Literature Discussion 6.2 Processing Time

Literature about processing time [ABC+15; CKE+15; ATM+17; Fli18] defines that processing time means using the local clock of the processor to create timestamps during the processing procedure. However, in contrast to our model this literature does not define a dedicated processing timestamp or more precisely which moment in time during the processing procedure a processing timestamp of a data stream element specifies.

In practice the time span during which a data stream element is processed ranges from the moment when the processing of the element starts to the moment when the potentially long-running processing procedure finishes.[7] In order to obtain for our model an unambiguous processing timestamp with respect to each processor processing a data stream element we define the processing timestamp assigned by a processor to a data stream element to specify the moment in time the processor started processing the element. That is, we assume every processor upon start of processing a data stream element to immediately create a timestamp using its clock and to assign this timestamp as the processing timestamp to the element. Please note that this does not prevent the processor from generating further timestamps during the processing procedure if this is required for the analysis.

As [ABC^{+}15], we do not assume the clocks of the processors to be synchronized[8] – neither with the raw input stream generating devices nor among each other. In consequence, every processor pr creates and assigns processing timestamps with respect to its individual processing time space $\mathfrak{T}_{proc(pr)}$. Only processors which are deployed on the same machine have the same processing time space as they share a clock (i.e., $\mathfrak{T}_{proc(pr_1)} = \mathfrak{T}_{proc(pr_2)}$ if pr_1 and pr_2 are deployed on the same machine). However, in general there are multiple different processing time spaces, one for each processor.

Note that if a data stream element is consumed and processed by multiple processors (e.g., a player sensor input stream element is consumed by one processors of a worker that detects penalty box entry events and by another processor of a worker that detects ball possession changes), each of these processors might assign a different processing timestamp to the element [ABC^{+}15]. This is due to their unsynchronized clocks and due to the different positions which the workers they are assigned to have in the workflow [ABC^{+}15]. That is, a data stream element dse that is consumed by processor pr_A and by processor pr_B might be assigned $\tau_A = 194$ at pr_A and $\tau_B = 216$ at pr_B. In contrast, the

[7] As we have presented in Section 5.3.2, there is no concurrency in a processor. Instead, processors process data stream elements sequentially. According to this, the time during which information of a data stream element is kept in state and used when processing later data stream elements (e.g., calculating velocities by means of the two latest positions) is not included in the processing time interval.

[8] Remember that the processor-synchronization assumption was only used in Section 6.1.2 to discuss the production time and the emission time on a theoretical level.

generation timestamp of a data stream element is always the same [ABC+15].[9]

Moreover, since processing timestamps depend on environmental conditions, such as the processing speed of processors of preceding workers and transmission delays of the network channels, factors which the developer cannot control, using them as the time indicator for time-dependent analyses and thus performing the analyses with processing time semantics does not lead to deterministic results [ATM+17; Fli18].[10] For instance, if processing timestamps are used for a time window based statistics calculation, i.e., if the local clock of the processor is used to trigger the code execution periodically as defined in our system model (see timer concept in Chapter 5) and all data stream elements processed between the last timer-triggered code execution (marking the start of the window) and the new timer-triggered code execution (marking the end of the window) are regarded to be within the window, the resulting statistics stream elements, containing data such as the average velocity of a player (see Example 4.5), depend on the environmental conditions since these conditions influence which data stream elements are processed between which timer-triggered code executions and thus belong to which time window regarding their processing timestamps. In consequence, the analysis results which are emitted by processors assigned to workers with timers exhibit no determinism guarantees. The same is true for the analysis results of processors of all workers which base (directly or transitively) on non-deterministic analysis results shipped in event, state, or statistics streams emitted by processors of other workers of the same data stream analysis system.

In addition, raw input stream elements can be assigned a timestamp $\tau \in \mathbb{N}_0$ when they are received by the first component of the data stream analysis system [CKE+15; ATM+17]. This time notion has been introduced by Flink as *ingestion time* [CKE+15].

[9] Note that we refer to the statement "Event time for a given event essentially never changes [...]" given on page 1794 of [ABC+15]. In contrast, the statement "[...] event-time timestamps [...] may also be modified at any point in the pipeline [...]" given on page 1795 of [ABC+15] is not valid in our model as we have defined every data stream element and thus also the generation timestamp contained in the data stream element tuple to be immutable (see Chapter 4).

[10] Note that although we add the additional constraint that processors process data stream elements in correct order with respect to their sequence numbers and thus deviate in our system model from the original processing time semantics idea (regarding the order in which elements of a data stream partition are processed) the determinism statement from [ATM+17] and [Fli18] to which we refer here is still true.

Literature Discussion 6.3 Ingestion Time

In Flink there are dedicated source operators which consume data stream elements from a preceded message queue and assign timestamps to them before they are handed to the actual analysis operators [Fli18]. In Flink this is used to automatically assign timestamps to raw input stream elements which are handled like generation timestamps (called event timestamps in Flink) [Fli18]. In our model, we adapt the idea of assigning timestamps to raw input stream elements when they are received by the data stream analysis system and refer to these timestamps as ingestion timestamps. However, we deviate from Flink in the way ingestion timestamps are handled in the data stream analysis system as in our model every raw input stream element is defined to have already a generation timestamp (see Definition 4.2).

According to our model only raw input stream elements are assigned an additional ingestion timestamp since only raw input stream elements are actually received by the data stream analysis system. In contrast, event, state, and statistics stream elements are not received from external devices but produced at and emitted by processors of workers which are part of the data stream analysis system. Consequently, there is no first component which receives event, state, and statistics stream elements. Therefore, these elements are also not assigned ingestion timestamps.

An alternative approach would be to interpret the processor which produces an event, state, or statistics stream element as the first component who receives the element and thus to assign an ingestion timestamp to every event, state, and statistics stream element when it is produced using the local clock of the processor. However, we have decided not to follow this approach since this interpretation does not match our understanding of “receiving” or “ingesting”. In fact, we argue that following this alternative approach would induce more confusion than benefit.

According to our system model (see Chapter 5) every raw input stream element enters the data stream analysis system via an entry component (our equivalent of Flink’s source operator). Hence, the ingestion timestamp of a raw input stream element is created by the local clock of the entry component through which the element entered the data stream analysis system. Since this clock is not assumed to be synchronized with the raw input stream generating devices, the ingestion timestamp does not specify a moment in time with respect to the generation time space. In contrast, the ingestion timestamp specifies a moment in time with respect to an ingestion time space. If a data stream analysis system has a single central entry component through which all raw input stream elements enter the data stream analysis system (as assumed in [ATM+17]), if the clocks of all entry components are synchronized (as suggested in [Roh16]), or if all entry components are deployed on the same machine and thus share a clock, there is a single consistent ingestion time space. If there are entry components whose clocks are not shared or synchronized – a situation which is supported by our model – there is a separate ingestion time space $\mathfrak{T}_{ing(EC)}$ for each set of entry components EC with the same or synchronized clocks.[11] Flink for instance might, as explained in [Roh16], create ingestion timestamps using multiple unsynchronized clocks and thus from different ingestion time spaces if no additional synchronization mechanism is realized.

As we have stated in Section 5.2.2, we assume every raw input stream element to enter a data stream analysis system only via a single entry component even if the data stream analysis system comprises multiple entry components. In consequence, for every data stream element there is only one entry component which assigns an ingestion timestamp to it. Moreover, in accordance with [Fli18], we assume that the ingestion timestamp cannot be changed by any subsequent component of the data stream analysis system. Consequently, the ingestion timestamp of every data stream element is unambiguous and immutable. Furthermore, even if there are multiple ingestion time spaces, the ingestion time of every data stream element is only specified with respect to one of these ingestion time spaces. Remember that in contrast to this, the processing time of a data stream element is specified multiple times with respect to different processing time spaces when processed at multiple processors.

Since ingestion timestamps are assigned by the entry components of a data stream analysis system, they depend on the network conditions[12] between the raw input stream generating devices and the entry components. Thus, we argue that the requirement[13] stated in [ATM+17] for achieving deterministic results by means of using ingestion timestamps as the time indicator for time-dependent analyses and thus performing the analyses with ingestion time semantics is not met. Instead we state that, as processing timestamps, they cannot be leveraged

[11] Note that this set can also be a unit set, i.e., contain only a single entry component ($|EC| = 1$), if the clock of this entry component is not shared with another entry component or synchronized with the clock of any other entry component.

[12] Remember that if the communication between the raw input stream generating devices and the entry components is pull-based the additional pull delay is regarded as a part of the transmission delay and thus involved in the network conditions (see Section 5.4.2).

[13] Quote: "Given a specific timing of arrival of input elements, ingestion time ensures that the engine produces deterministic results." [ATM+17]

as time indicators to generate deterministic results of time-dependent analyses. Another reason for the non-determinism caused by our system model, or more precisely by the assumption that processors process data stream elements in correct order with respect to their sequence numbers, is that the fact that data stream elements are processed in correct order with respect to their ingestion timestamps cannot be enforced but depends on environmental conditions (see discussion about consistency between sequence number ordering and ingestion time ordering in Section 6.2.5) and thus the same issues as when using the generation timestamps as the time indicator may arise (see Section 6.1.1).

It is important to note that the ingestion time unambiguity is only true when considering a single data stream analysis system. As described in Section 5.5, in practice there might be multiple different data stream analysis systems which consume the same raw input stream elements. The entry components of these data stream analysis systems might assign different ingestion timestamps to the same raw input stream elements. Hence, the ingestion timestamp of a data stream element depends not only on the element but also on the data stream analysis system which consumed the element. Moreover, we have defined in our system model that a data stream analysis system can and usually does consist of multiple workers. As explained above, a data stream element which is processed by processors of multiple workers is assigned a (potentially different) processing timestamp at each processor. Thus, the processing timestamp of a data stream element even depends on a specific processor of a data stream analysis system. As a consequence of these additional[14] dependencies and the fact that we define a data stream element to be immutable, the data stream element tuple (see Definition 4.2) does neither contain an ingestion timestamp nor a processing timestamp. Instead, we include the processing time and the ingestion time of a data stream element into our model by means of formally defining the following functions:

[14] Remember that the generation time of a data stream element is globally unambiguous thus included in the data stream element tuple.

Definition 6.5 Processing Time

The function $\tau\,(dse, pr)$ returns the processing timestamp τ of the data stream element *dse* at processor *pr*.

$$\tau\,(dse, pr) = \begin{cases} \text{time at which } pr \text{ started processing } dse & \text{if} \quad dse \text{ is processed by } pr \\ \text{undefined} & \text{otherwise} \end{cases}$$

where time is measured using the local clock of *pr* and thus specified with respect to $\mathfrak{T}_{proc(pr)}$.

Definition 6.6 Ingestion Time

The function $\textsc{t}\,(dse, dsas)$ returns the ingestion timestamp $\textsc{t}$ of the data stream element *dse* in the data stream analysis system *dsas*.

$$\textsc{t}\,(dse, dsas) = \begin{cases} \text{time at which } dse \text{ entered } dsas & \text{if} \quad dse \text{ is consumed by } dsas \\ & \quad\;\; \text{and } dse.ds.cat = \text{“rawInput”} \\ \text{undefined} & \text{otherwise} \end{cases}$$

where time is measured using the local clock of the entry component *ec* through which *dse* entered *dsas* and thus specified with respect to $\mathfrak{T}_{ing(EC)}$ with $ec \in EC$.

6.1.4 Timestamp Comparison

Altogether there are five different notions of time. First, every data stream element is assigned a globally unambiguous and environmental condition dependent generation timestamp *ts* specifying the moment in time when a raw input was generated, when an atomic event occurred, or to which the latest update of a non-atomic event, a state, or a statistic refers. This moment in time is specified with respect to the single globally consistent generation time space $\mathfrak{T}_{gen}$. Second and third, every event, state, and statistics stream element could in theory additionally be assigned a globally unambiguous but environmental condition dependent production timestamp ts_{prod} specifying the moment in time when the data (of an event, state, or statistics stream element) was produced at a processor and a globally unambiguous but environmental condition dependent emission timestamp ts_{emis} specifying the moment in time when the data was emitted by a processor. Both timestamps specify the moment in time with respect to the same globally consistent generation time space $\mathfrak{T}_{gen}$. Fourth, every data stream element can be assigned at each processor *pr* which processes the data stream element an environmental condition dependent processing timestamp $\tau\,(dse, pr)$ specifying the moment the processing started. This moment in time is specified with respect to the processors's processing time space $\mathfrak{T}_{proc(pr)}$ which is individual in general since only processors which are deployed on the same machine have the same processing time space as they share a clock. Moreover, the processing timestamp is only unambiguous when regarding a single processor but might be ambiguous when regarding it on a data stream analysis system level or on a global scope since elements of an event, state, or a statistics stream partition can be processed by multiple processors of the same data stream analysis system and since elements of a raw input stream partition can be even processed by processors of different data stream analysis systems. Fifth, every raw input stream element can further be assigned an environmental condition dependent ingestion timestamp $\text{T}\,(dse, dsas)$ specifying the moment when an entry component *ec* of the data stream analysis system *dsas* received the raw input stream element. This moment in time is specified with respect to the ingestion time space $\mathfrak{T}_{ing(EC)}$ with $ec \in EC$. Regardless of whether there is only a single ingestion time space per data stream analysis system or not – a factor which depends on the number of entry components, their deployment, and the synchronicity of their clocks – the ingestion timestamp of a raw input stream element is unambiguous when regarding only a single data stream analysis system. However, this is not the case on a global scope, i.e., when regarding multiple data stream analysis systems.

Table 6.1 contrasts the five different time notions regarding multiple properties and Table 6.2 compares the three time spaces with respect to their consistency. Since the production time and the emission time have already been compared extensively to the generation time in Section 6.1.2 and since both time notions are further solely theoretical concepts but unavailable in practice, we focus in the remainder of this section on the comparison of the generation time, the processing time, and the ingestion time.

It is clear that the moments in time specified by two timestamps cannot be compared if the timestamps are created by two unsynchronized clocks with an unknown skew and thus specify moments in time with respect to different time spaces. In consequence, one cannot arbitrarily compare generation timestamps, processing timestamps and ingestion timestamps. For instance, it is impossible to compare the processing timestamp assigned by a processor to one data stream element with the generation timestamp of another data stream element to check

Time notion	Assigned to	Semantics	Time space	Timestamp unambiguity
Generation time	All data stream elements	Specifies moment when the raw input was generated, when the atomic event occurred, or to which the non-atomic event update, the state, or the statistic refers	Generation time space	Globally unambiguous
Production time	Event, state, and statistics stream elements	Specifies moment of data production at a processor	Generation time space	Globally unambiguous
Emission time	Event, state, and statistics stream elements	Specifies moment of data emission by a processor	Generation time space	Globally unambiguous
Processing time	All data stream elements	Specifies moment when a processor started processing the element	Processing time space	Potentially different at each processor of every data stream analysis system
Ingestion time	Raw input stream elements	Specifies moment when an entry component of a data stream analysis system received the element	Ingestion time space	Unambiguous in a single data stream analysis system but potentially different in different data stream analysis systems

Table 6.1 Time Notion Overview. The table contrasts the different time notions with respect to multiple properties.

Time space	Consistency
Generation time space	One globally consistent generation time space
Processing time space	Only processors which are deployed on the same machine have the same processing time space as they share a clock. However, in general there is an individual processing time space at each processor in every data stream analysis system.
Ingestion time space	If there is only a single central entry component or all entry components of a data stream analysis system either have synchronized clocks or are deployed on the same machine and thus share a clock, there is a single consistent ingestion time space per data stream analysis system. Otherwise there is an individual ingestion time space for each set of entry components with the same or synchronized clocks.

Table 6.2 Time Space Overview. Consistency properties of the different time spaces.

if the former was processed more than one minute after the latter was generated since the clock of the processor might be shifted by an unknown amount of time with respect to the clocks of the raw input stream generating devices. In fact, it is even possible that the processing timestamp a processor assigns to a raw input stream element is smaller than the generation timestamp of the very same element, although a data stream element can of course not be processed before it has been generated (see thought experiment below), if the processors's clock is shifted some seconds to the past with respect to the clocks of the raw input stream generating devices. Moreover, it is impossible to compare processing timestamps assigned by different processors and ingestion timestamps assigned by different entry components if their clocks are not synchronized. For instance, it is not possible to check in the course of an analysis at a processor of a worker if a raw input stream element was received by an entry component of the data stream analysis system more than one minute later than another raw input stream element was received by an entry component of the data stream analysis system if it cannot be guaranteed that both raw input stream elements were received by the same entry component or that both entry components either have synchronized clocks or are deployed on the same machine and thus share a clock. Instead only timestamps specifying moments in time with respect to the same time space, such as two processing timestamps assigned by the same processor or two generation timestamps since we assume in our model that all raw input stream generating devices have synchronized clocks, can be compared.

In order to nevertheless obtain an intuition how timestamps of the different time notions relate to each other we conduct a thought experiment[15]: Assume that all timestamps specify moments in time with respect to a single global time space. To achieve this the clocks of all raw input stream generating devices, entry components, and processors would have to be synchronized (full-synchronization assumption). In this case the generation timestamp of a raw input stream element would be always smaller or equal to the ingestion timestamp assigned to it by any entry component of any data stream analysis system. The ingestion timestamp an entry component of a data stream analysis system assigns to a raw input stream element would in turn be smaller or equal to every processing timestamp this element might be assigned by any processor of the same data stream analysis system. Both follows from the fact that, regarding only a single data stream analysis system, a raw input stream element is first generated by a raw input stream generating device, then received and forwarded by an entry component, and then received and processed by processors. Moreover, the generation timestamp of an event, state, or statistics stream element would be always smaller or equal to every processing timestamp this element might be assigned by any processor. This is due to the fact that an event, state, or statistics stream element cannot be received and processed by a processor before the raw input stream element from which it inherited its generation timestamp directly or transitively is generated (see inheritance of generation timestamps in Section 6.1.1).

However, even with the full-synchronization assumption in this though experiment the exact differences between the timestamps that are assigned to a data stream element depend on environmental conditions. More precisely, the pair-wise differences between the generation timestamp of a data stream element, the ingestion timestamps assigned by entry components of different data stream analysis systems to the element (in case the element belongs to a raw input stream), and the processing timestamps assigned by different processors of the same or different[16] data stream analysis systems to the element depend on the environmental conditions. The reason for this is that, as shown in the previous sections, neither generation timestamps, nor ingestion timestamps, nor

[15] Please note, that the assumptions that we make in this and the two subsequent paragraphs are only posed in the context of this thought experiment. They are not valid for the remainder of our model.

[16] According to our system model, multiple data stream analysis systems can share raw input streams (see Section 5.5). Hence, processors of multiple data stream analysis systems can process the same raw input stream elements and assign processing timestamps to them. However, this is not the case for event, state, and statistics stream elements.

processing timestamps are agnostic to environmental conditions. Furthermore, these differences are not constant but might vary over time as also the environmental conditions are not necessarily constant. That is, while the difference between the generation timestamp of an element of a certain data stream and the processing timestamp assigned to it by some processor might have been 10 milliseconds, some minutes later the difference between the generation timestamp of another element of the same data stream and the processing timestamp assigned to it by the same processor might be 200 milliseconds. These environmental condition dependent time differences have been also described and discussed in [ABC+15] under the term *time skew*.

If we kept the full-synchronization assumption and further disregarded some of the environmental conditions by additionally assuming that there are no processing and transmission delays (zero-delay assumption), the generation timestamp of a raw input stream element, the ingestion timestamps assigned to this element by the entry components of all data stream analysis systems, and the processing timestamps assigned to this element by all processors of all data stream analysis systems would be equal. In contrast, the generation timestamp of an event stream element, a state stream element, or a statistics stream element could still be smaller than the processing timestamps assigned to the element by processors processing this element. This is due to the fact that an event, state, or statistics stream element cannot be processed before it is produced and emitted by the processor of the preceding worker which generated it and thus the processing timestamp assigned to an event, state, or statistics stream element at any processor cannot be smaller than the production timestamp and the emission timestamp of the element which we showed in Section 6.1.2 to be potentially greater than the generation timestamp even under zero-delay assumptions.

6.2 Orderings

In Section 6.1 we have discussed the temporality of single data stream elements by looking at them individually. However, when regarding data streams it is also interesting to examine how the elements of a data stream partition are ordered with respect to the different time notions.

As we have presented above, each element of a data stream partition is assigned different timestamps of different time notions by different entities. Since each of these timestamps is a natural number that specifies a moment in time, the elements of a data stream partition can be ordered with respect to a certain time notion by means of comparing the corresponding timestamps. Hence, the timestamps of each time notion introduce one or in case of the processing and ingestion time even multiple orderings of every data stream partition. Moreover, as presented in Chapter 4, the elements of a data stream partition are ordered by means of sequence numbers.

In this section, we will (i) define the orderings introduced by sequence numbers of and timestamps assigned to elements of a data stream partition, (ii) investigate and compare the properties of these orderings, (iii) prove and confute the consistency between pairs of these orderings, and (iv) identify if data stream elements are guaranteed to be processed in correct order or might be processed out-of-order with respect to a certain ordering. Especially the last point is important for the analysis at the processors of the workers and thus has to be considered when designing and implementing workers of a data stream analysis system. We want to highlight that we will analyze the properties of the orderings as well as their pairwise consistency on the basis of our generic model and thus not only for one specific implementation.

However, before we can start to define and discuss these specific orderings we first need to define orderings in general. In [Opa79] an ordering is formally defined as follows:

Definition 6.7 Ordering

"A partial ordering of a set S is a relation between elements of S, denoted by $<$, satisfying the following properties for any elements a, b, c in S:

i) If $a < b$ and $b < c$ then $a < c$.

ii) If $a < b$ then $b \nless a$.

iii) $a \nless a$

A partial ordering of S is called total ordering of S if for any two distinct elements a, b in S either $a < b$ or $b < a$." [Opa79]

In this thesis, we syntactically deviate from this definition by writing $\langle a, b\rangle \in \prec$ instead of $a < b$ and $\langle a, b\rangle \notin \prec$ instead of $a \nless b$. We do so to distinct the ordering relations we define from the symbol which is used to compare real numbers (e.g., $3 < 5$) and to improve the readability since the orderings we will present in the remainder of this section have parameters. However, semantically we fully comply with the ordering definition given in [Opa79].

To compare orderings we formally define the equality and consistency of a pair of orderings as follows:

Definition 6.8 Ordering Equality

Two orderings $\prec_A$ and $\prec_B$ which order the same set S are equal if any two distinct elements x, y in S which are ordered by $\prec_A$ are ordered in the same way by $\prec_B$, and vice versa.

$$\forall x, y \in S : \langle x, y \rangle \in \prec_A \iff \langle x, y \rangle \in \prec_B$$

Orderings which are not equal are unequal.

Definition 6.9 Ordering Consistency

Two orderings $\prec_A$ and $\prec_B$ which order the same set S are consistent if any two distinct elements x, y in S which are ordered by $\prec_A$ are not ordered reversely by $\prec_B$, and vice versa.

$$\forall x, y \in S : \langle x, y \rangle \in \prec_A \implies \langle y, x \rangle \notin \prec_B$$
$$\forall x, y \in S : \langle x, y \rangle \in \prec_B \implies \langle y, x \rangle \notin \prec_A$$

Orderings which are not consistent are inconsistent.

Note that one can follow from the fact that two orderings are equal that they are also consistent, but not vice versa. That is, if two orderings are equal they are guaranteed to be consistent but there are orderings which are consistent but still unequal. Moreover, one can follow from the fact that two orderings are inconsistent that they are unequal, but not vice versa. That is, if two orderings are inconsistent they are guaranteed to be unequal but there are orderings which are unequal but still consistent.

In the following subsections, we will define and discuss the ordering relations introduced by sequence numbers, generation timestamps, processing timestamps, and ingestion timestamps. However, we decided to refrain from defining and discussing ordering relations introduced by production and emission timestamps since, as described in Section 6.1.2, both are not available in practice but only presented in this thesis as a theoretical concept to clarify the semantics of generation timestamps.

6.2.1 Sequence Number Ordering

According to our model (see Definition 4.2), every data stream element does not only contain a generation timestamp *ts* but also a logical number ξ to which we refer as *sequence number*. While each generation timestamp specifies a moment in time, the sole purpose of the sequence numbers is to specify how multiple data stream elements are logically ordered.

As done in Samza which uses Kafka's offsets presented in [KNR11] to order data stream elements, we assume that the sequence numbers order all elements belonging to the same data stream partition, i.e., belonging to the same data stream and having the same key, and thus introduce a total ordering of the elements of a data stream partition but that the sequence numbers cannot be used to order elements of different data stream partitions [Sam17d; KK15].[17] As we will show in more detail in Section 6.2.2, generation timestamps do not introduce such a total ordering as two elements of the same data stream partition can have the same generation timestamp.

Formally, we define the sequence number ordering as follows:

Definition 6.10 Sequence Number Ordering

The sequence number ordering $<_\xi(ds,k,\xi)$ is a relation that totally orders all elements of the data stream partition *dsp* identified by the data stream *ds* and the key *k* with respect to their sequence numbers. That is, it totally orders all elements in $DSE(ds,k,\xi)$ for every sequence number $\xi \in \mathbb{N}_0$.

$$<_\xi(ds,k,\xi) = \left\{ \langle dse_1, dse_2 \rangle \;\middle|\; dse_1, dse_2 \in DSE(ds,k,\xi) \wedge \underbrace{dse_1.\xi < dse_2.\xi}_{\text{ordering w.r.t. } \xi} \right\}$$

Literature Discussion 6.4 Sequence Number

In literature, other terms, such as *offset* [KNR11], *stream time* [Bre08], and *id* [ATM+17], are used for logical numbers with the same or very similar semantics. Actually, [KK15] even states that the offset presented in [KNR11] is a "per-partition monotonically increasing sequence number". Likewise, [Bre08] states that the stream time of a data stream element is a "logical sequence number" and [ATM+17] states that "SECRET identifies each element of the in-

[17] Remember that, as stated already in Literature Discussion 5.4, we deviate in our model from Samza regarding the fact that in Samza a data stream partition comprises a subset of the key domain instead of only a single key.

put stream with a unique id and defines a global order between the stream elements based on their id". The sole difference in [Bre08] and [ATM^{+}17] is that they do not consider data stream partitioning.

As generation timestamps, sequence numbers are independent of a certain processor, worker, or data stream analysis system but only depend on the data stream element. Thus, the sequence number of every data stream element is globally unambiguous. However, the sequence numbers of two data stream elements belonging to the same data stream partition are not independent. As done in [Sam17d], we assume the sequence number of a data stream element to be unique per data stream partition. Hence, two data stream elements of the same data stream partition must not have the same sequence number. Otherwise, the sequence numbers of the data stream elements would not introduce a total ordering.

In the following we will present implementation-independent[18] approaches to properly assign sequence numbers in a way that they introduce a total ordering. Moreover, we will discuss some properties of the sequence number ordering.

6.2.1.1 Assignment Approaches

There are data stream partitions whose elements are generated and emitted by a single component, i.e., by a single raw input stream element generating device or by a single processor. For instance, under the assumption that there is a data stream analysis system which analyses a certain football match and which incorporates a pass detection worker to detect successful passes, all successful pass event stream elements for this match and thus belonging to a certain partition if the match identifier is used as the key are generated by the same processor of the pass detection worker. To assign sequence numbers to elements of such data stream partitions, the component generating these elements can simply keep a local counter which is incremented for every new element and whose current value is assigned as the sequence number to a new element. In the remainder of this thesis we denote this assignment approach as the *local counter approach*.

However, there are also data stream partitions whose elements are generated and emitted by multiple components. For instance, the player sensor input

[18] Note that we do not present implementation details in Section 6.2.1.1. Instead we present three different approaches to assign sequence numbers that are conform to our model.

stream elements of one match and thus belonging to one player sensor input stream partition might be generated and emitted by multiple sensor devices (e.g., one per player). Moreover, a football analysis system might comprise two workers whose processors detect counter attack events using different algorithms but emit elements representing counter attack events with the same key (the match identifier) and in the same counter attack event stream and thus in the same data stream partition. In this case, using local counters at every component would not be sufficient. If the components should assign the sequence number by themselves, they need to be coordinated. For instance they can share a counter by means of a coordination service such as Apache Zookeeper [HKJ$^+$10].[19] An alternative approach is to use the brokers of a message queue or a publish/subscribe system as proxies which, besides providing other services, take the responsibility to assign sequence numbers to the elements of every data stream partition. For instance, each Kafka broker appends new data stream elements to its log and thus introduces sequence numbers in form of offsets with respect to the start of the first log file [KNR11; KK15]. In the remainder of this thesis we denote these two assignment approaches as the *shared counter approach* and the *proxy approach*.

If the shared counter approach or the proxy approach is used to assign sequence numbers, there can be multiple shared counters at different coordination services or multiple proxies. However, all components which generate elements of the same data stream partition have to leverage the same shared counter provided by the same coordination service to create sequence numbers or to send all elements of a certain data stream partition to the same proxy which assigns sequence numbers and forwards the elements to other components. Otherwise there could be two elements of the same data stream partition, i.e., belonging to the same data stream and having the same key, with the same sequence number, a fact which would violate the unique identification assumption of our model posed in Chapter 4. We argue that this is not a big deal for raw input stream partitions, as the requirement that raw input stream generating devices which emit elements of the same raw input stream partition leverage the same coordination service or send these elements to the same proxy is less restrictive than the clock synchronization requirement we pose already (see Section 6.1.1). The same is true for event, state, and statistics stream partitions as all workers whose processors generate elements of the same event, state, or statistics stream

[19] As with the proxies (see Section 5.4.2), we relinquish further defining the components of such a coordination service but simply assume that they are deployed on machines with bidirectional network channels to every other machine.

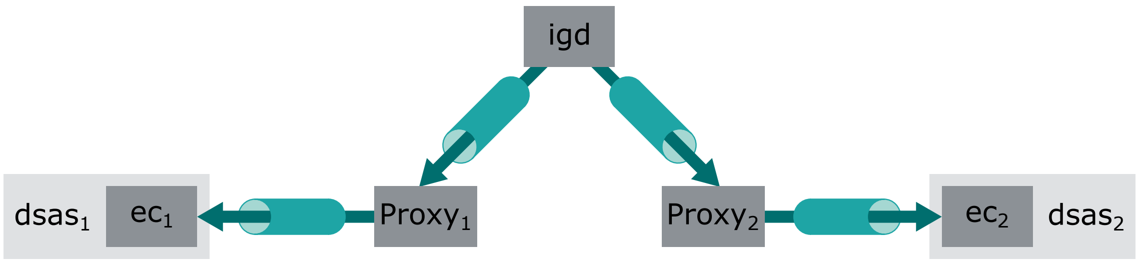
partition are part of the same[20] data stream analysis system (as exemplified in the counter attack event detection example described above) and a data stream analysis system is usually managed holistically by a single entity.

Note that if the proxy approach is used to assign sequence numbers to elements of a raw input stream partition, all data stream analysis systems that analyze elements of this partition have to consume these elements via the same proxy. That is, a raw input stream generating device that generates and emits elements of a raw input stream partition consumed by two data stream analysis systems cannot send all elements to two different proxies which each independently assign sequence numbers and then forward the elements to an entry component of one of the data stream analysis systems (see Figure 6.4(a)). Instead, the raw input stream generating device has to send all elements to a single proxy which forwards the elements to the entry components of both data stream analysis systems (see Figure 6.4(b)).[21] Consequently, every raw input stream partition is assigned to a single proxy which is shared by all data stream analysis systems. Otherwise, i.e., if different data stream analysis systems could consume elements of the same raw input stream partition via different proxies which introduce sequence numbers independently, there could be different sequence numbers for the same element in two data stream analysis systems (e.g., caused by network channels without FIFO guarantees between the raw input stream generating device and the proxies). This would invalidate the global unambiguity property, i.e., the fact that sequence numbers are independent of a certain data stream analysis system, and thus render modeling sequence numbers as part of the data stream element tuple (see Definition 4.2) impossible. Instead, sequence numbers of raw input stream elements would have to be modeled with respect to the data stream analysis system in a similar way as ingestion time is modeled in Definition 6.6.

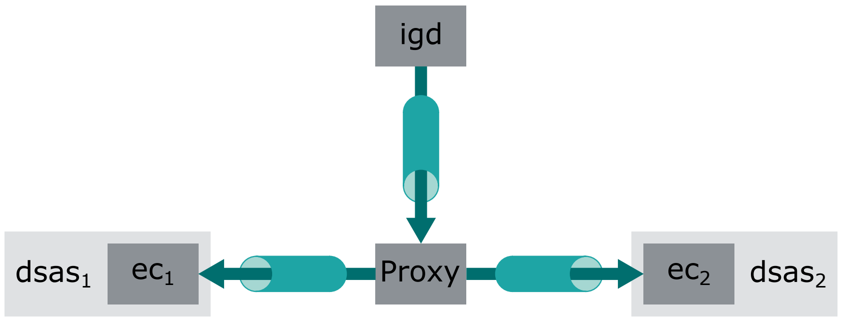
Similarly, a processor of a worker of a data stream analysis system cannot send all elements of an event, state, or statistics stream partition to two different proxies which each assign sequence numbers and then forward the elements to

[20] Remember that, according to our system model, processors of different data stream analysis systems cannot emit data stream elements in the same event, state, or statistics stream (see Section 5.5).

[21] Note that the raw input stream generating device has to send all elements of the raw input stream partition dsp_1 to proxy $Proxy_1$ which forwards the elements to the entry components of both data stream analysis systems but that it is still allowed to send all elements of another raw input stream partition dsp_2 to another proxy $Proxy_2$ which also forwards the elements to the entry components of both data stream analysis systems.



(a) Prohibited Multi Proxy Architecture: igd is not allowed to send all elements of a raw input stream partition to $Proxy_1$ which forwards the elements to ec_1 and to $Proxy_2$ which forwards the elements to ec_2.



(b) Correct Single Proxy Architecture: igd has to send all elements of a raw input stream partition to $Proxy$ which forwards the elements to ec_1 and to ec_2.

Figure 6.4 Prohibited and Correct Architecture for the Proxy Approach. The dark-gray boxes illustrate the raw input stream generating device (igd), the proxies ($Proxy_1$ and $Proxy_2$ in (a), $Proxy$ in (b)), and the entry components (ec_1 and ec_2). The light-gray boxes visualize which entry component belongs to which data stream analysis system ($dsas_1$ and $dsas_2$). The pipes between the boxes illustrate the network channels between the raw input stream generating devices, the proxies, and the entry components via which the data stream elements are transferred.

other processors of workers of the same[22] data stream analysis system and to external consumers. Instead, the processor has to send all elements to a single proxy which forwards the elements to all processors of subsequent workers and to all external consumers. Otherwise, there could be different sequence numbers for the same data stream element at different processors and/or external consumers and thus sequence numbers of event, state, and statistics stream elements would have to be modeled with respect to the processor (cf. processing time in Definition 6.5) and the external consumers.

[22] Note that we do not have to consider processors of workers of other data stream analysis systems since, as defined in our system model, a worker cannot (directly) consume event, state, or statistics streams produced by workers of another data stream analysis system (see Section 5.5).

6.2.1.2 Properties

Sequence numbers and thus also the sequence number ordering introduced by them are not agnostic to environmental conditions. If sequence numbers are assigned by a proxy, the sequence numbers of the elements of a data stream partition depend on the order at which the elements arrive at the proxy. This order depends on the transmission delays between the proxy and the raw input stream generating devices or the processors as well as on the ordering guarantees of the network channels. Similarly, if sequence numbers are created by means of the shared counter approach since the components which emit the elements of a data stream partition should assign the sequence numbers themselves, the sequence numbers depend on the transmission delays between the components and the coordination service. This is due to the fact that if two components try to access a shared counter to create a new sequence number at the same time, the component which reaches the coordination service hosting the shared counter first will receive the smaller sequence number. In fact, even if all data stream elements are only emitted by a single processor or raw input stream generating device per data stream partition and thus the local counter approach can be used without further coordination, the sequence numbers of the emitted data stream elements are not necessarily agnostic to environmental conditions since the fact which raw input are generated, which events are detected, which states are calculated, and which statistics are generated and thus emitted as raw input, event, state, and statistics stream elements, respectively, might depend on the intrinsic environmental condition dependency of the data generation of the raw input stream generating device (e.g., a measurement might be skipped due to an overload situation) or on environmental condition dependent information like generation or processing timestamps.

Regardless of whether the sequence numbers depend on environmental conditions or not, the sequence number of every data stream element is globally unambiguous. Hence, there is also only one globally unambiguous sequence number ordering for every data stream partition. This is true under all network conditions (which we tolerate in our system model, see Section 5.4.2), i.e., even in presence of message reordering or high transmission delay variance, between the raw input stream generating devices, the processors, and coordination services or proxies and under all processing conditions inside the data stream analysis systems (e.g., processor overload).

6.2.2 Generation Time Ordering

In contrast to the sequence number whose explicit purpose is to order elements of the same data stream partition, the generation timestamp specifies a moment in time. In consequence, it is possible that multiple elements of the same data stream partition have the same generation timestamp. This is by design as these data stream elements might actually refer to the same moment in time regarding the generation time space. For instance, in Example 4.5, the fitness statistics stream element for player B7 (dse_2) and the aggregated team fitness statistics stream element for team B (dse_3) both inherit and thus contain the generation timestamp (ts = 119'932) from the same player sensor input stream element for player B7 as this is the data stream element with the largest generation timestamp that has updated the fitness statistic for player B7 and team B. Moreover, it is also possible that two player sensor input stream elements shipping information for different players of the same match have the same generation timestamp as they are concurrently measured by two sensor devices. For instance, in Example 4.1 the player sensor input stream elements for player B7 (dse_2 and dse_6) contain the same generation timestamp (ts = 32 and ts = 132) as the player sensor input stream elements for player A1 (dse_3 and dse_7).

As multiple data stream elements of the same data stream partition can have the same generation timestamp, generation timestamps introduce only a partial ordering of the elements of a data stream partition [ATM+17]. Formally, we define this generation time ordering as follows:

Definition 6.11 Generation Time Ordering

The generation time ordering $\prec_{ts}(ds, k, \xi)$ is a relation that partially orders the elements of the data stream partition dsp identified by the data stream ds and the key k with respect to their generation timestamps. That is, it partially orders the elements in $DSE(ds, k, \xi)$ for every sequence number $\xi \in \mathbb{N}_0$.

$$\prec_{ts}(ds, k, \xi) = \left\{ \langle dse_1, dse_2 \rangle \;\middle|\; dse_1, dse_2 \in DSE(ds, k, \xi) \wedge \underbrace{dse_1.ts < dse_2.ts}_{\text{ordering w.r.t. } ts} \right\}$$

As the generation time ordering is introduced by the globally unambiguous generation timestamps, there is only one generation time ordering for every data stream partition which is globally unambiguous. However, since the generation timestamps depend on environmental conditions (see Section 6.1.1), also this ordering is not agnostic to environmental conditions.

6.2.3 Processing Time Ordering

As presented in Section 6.1.3, every processor can assign a processing timestamp created using its local clock to every data stream element it processes. Since we have defined in our system model that processors do not process multiple data stream elements at the same time, there is no actual concurrency inside a processor (see Section 5.3.2). Nevertheless, there might be multiple consecutively processed data stream elements that are assigned the same processing timestamp as a processors's clock can only produce timestamps with a certain granularity. For instance, if a processor processes ten data stream elements per millisecond but its clock can only create timestamps with millisecond resolution, ten data stream elements are assigned the same processing timestamp. In consequence, the processing timestamps assigned by a processor introduce only a partial ordering of the elements of a data stream partition. Since the processing timestamps of the elements of a data stream partition are unambiguous with respect to each processor, also the resulting processing time ordering is unambiguous for every processor. Formally, we define this processor-specific processing time ordering as follows:

Definition 6.12 Processing Time Ordering

The processing time ordering $\prec_{ts}(ds,k,pr,\xi)$ is a relation that partially orders the elements of the data stream partition *dsp* identified by the data stream *ds* and the key *k* with respect to the processing timestamps processor *pr* has assigned to them. That is, it partially orders the elements in $DSE(ds,k,\xi)$ for every sequence number $\xi \in \mathbb{N}_0$.

$$\prec_\tau(ds,k,pr,\xi) = \left\{ \langle dse_1, dse_2 \rangle \;\middle|\; \begin{array}{l} dse_1, dse_2 \in DSE(ds,k,\xi) \wedge \\ \underbrace{\tau\,(dse_1,pr) < \tau\,(dse_2,pr)}_{\text{ordering w.r.t. } \tau \text{ at } pr} \end{array} \right\}$$

Note that the processing time ordering $\prec_\tau(ds,k,pr,\xi)$ orders the elements of the data stream partition *dsp* identified by the data stream *ds* and the key *k* only if processor *pr* consumes and processes the elements of *dsp*. If the elements of *dsp* are not processed by *pr*, the processing timestamp $\tau\,(dse,pr)$ is undefined for every $dse \in DSE(ds,k,\xi)$ (for every $\xi \in \mathbb{N}_0$) and thus $\prec_\tau(ds,k,pr,\xi)$ is an empty set.

Although different processors which consume the same data stream partition might assign different processing timestamps to the elements of this partition, the processing time orderings introduced by the processing timestamps of the different processors are guaranteed to be consistent. This is not only true for two processors of the same but also for two processors of two different data stream analysis systems.[23] The reason for this is that, as defined in Section 5.3.2, every processor processes elements of a data stream partition sequentially ordered by their sequence numbers and that sequence numbers are globally unambiguous and thus independent of the processor which process the data stream elements. However, the processing time orderings introduced by the processing timestamps of different processors are not guaranteed to be equal. In the following we express and prove these statements formally:

Theorem 6.1 Processing Time Ordering Consistency

Processing timestamps assigned by different processors of the same or different data stream analysis systems introduce consistent processing time orderings. That is, if two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of the processing timestamps assigned by processor pr_1, they are not ordered reversely by means of the processing timestamps assigned by processor pr_2.

$$\begin{aligned}
&\forall dse_1, dse_2 \in \widehat{DSE} \wedge pr_1, pr_2 \in \widehat{PR} : \\
&\quad \langle dse_1, dse_2 \rangle \in \prec_\tau(ds, k, pr_1, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\
&\quad \Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_\tau(ds, k, pr_2, \xi) \text{ for any } \xi \in \mathbb{N}_0
\end{aligned}$$

Proof 6.1 Processing Time Ordering Consistency

Assume that dse_1 and dse_2 with $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$ are ordered by means of the processing timestamps assigned by processor pr_1.

$$\langle dse_1, dse_2 \rangle \in \prec_\tau(ds, k, pr_1, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

Hence, according to Definition 6.12, the processing timestamp pr_1 has assigned to dse_1 is smaller than the processing timestamp pr_1 has assigned to dse_2.

[23] Note that processors of different data stream analysis systems can only process elements of the same raw input stream partition but not of the same event, state, or statistics stream partition (see Section 5.5).

$$\tau\left(dse_1, pr_1\right) < \tau\left(dse_2, pr_1\right)$$

Since we know that pr_1 processes elements of a data stream partition sequentially with respect to their sequence numbers (see Section 5.3.2), we can follow that the sequence number of dse_1 is smaller than the sequence number of dse_2.

$$dse_1.\xi < dse_2.\xi$$

As also processor pr_2 processes elements of a data stream partition sequentially with respect to their sequence numbers and since sequence numbers are globally unambiguous (see Section 6.2.1) we can follow that the processing timestamp pr_2 has assigned to dse_1 is smaller or equal to the processing timestamp pr_2 has assigned to dse_2.

$$\tau\left(dse_1, pr_2\right) \leq \tau\left(dse_2, pr_2\right)$$

According to Definition 6.12, this means that dse_1 and dse_2 cannot be ordered reversely by means of the processing timestamps assigned by pr_2.

$$\langle dse_2, dse_1\rangle \notin \prec_\tau\left(ds, k, pr_2, \xi\right) \text{ for all } \xi \in \mathbb{N}_0$$

Hence, the processing timestamps assigned by different processors introduce consistent processing time orderings. □

Theorem 6.2 Unguaranteed Processing Time Ordering Equality

The processing time orderings introduced by the processing timestamps of different processors are not guaranteed to be equal. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of the processing timestamps assigned by processor pr_1 that they are ordered in the same way by means of the processing timestamps assigned by processor pr_2.

$$\begin{aligned}&\langle dse_1, dse_2\rangle \in \prec_\tau\left(ds, k, pr_1, \xi\right) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi\\ &\not\Longrightarrow \langle dse_1, dse_2\rangle \in \prec_\tau\left(ds, k, pr_2, \xi\right) \text{ for any } \xi \in \mathbb{N}_0\end{aligned}$$

Proof 6.2 Unguaranteed Processing Time Ordering Equality

We prove that the processing time orderings introduced by the processing timestamps of different processors are not guaranteed to be equal by means of giving a counterexample for equality. Assume there are two data stream elements dse_1 and dse_2 with $dse_1.ds = dse_2.ds = ds$, $dse_1.k = dse_2.k = k$, $dse_1.\xi = 1$, and $dse_2.\xi = 2$. Further, assume that processor pr_1 and processor pr_2 have assigned them the following processing timestamps:

$$\begin{aligned} \tau\left(dse_1, pr_1\right) &= 253 & \qquad \tau\left(dse_1, pr_2\right) &= 257 \\ \tau\left(dse_2, pr_1\right) &= 254 & \qquad \tau\left(dse_2, pr_2\right) &= 257 \end{aligned}$$

According to Definition 6.12, this means that dse_1 and dse_2 are ordered by means of the processing timestamps assigned by pr_1 but not by means of the processing timestamps assigned by pr_2.

$$\begin{aligned} &\langle dse_1, dse_2 \rangle \in \prec_\tau\left(ds, k, pr_1, \xi\right) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\ &\langle dse_1, dse_2 \rangle \notin \prec_\tau\left(ds, k, pr_2, \xi\right) \text{ for all } \xi \in \mathbb{N}_0 \end{aligned}$$

In consequence, the processing time orderings introduced by different processors are not guaranteed to be equal. □

Theorem 6.1 holds under all network and processing conditions even if the network channels do not provide FIFO guarantees and thus might reorder packages containing data stream elements resulting in different data stream element arrival orders at different processors. Note that this is only the case since we assume in our system model that every processor is able to buffer incoming data stream elements until it is guaranteed that no more late data stream elements (i.e., data stream elements with a smaller sequence number) arrive and thus that the processor actually processes data stream elements ordered by their sequence numbers (see Section 5.3.2).

Literature Discussion 6.5 Buffering to Process in Correct Order

Literature [ABC$^+$15; Fli18] which regards using the generation timestamps of data stream elements as the time indicators for time-dependent analyses (e.g., window aggregates) states that it can never be guaranteed in practice that all data stream elements up to a certain generation timestamp have arrived since a processor cannot wait forever and also mechanisms like watermark-

ing [ABB+13] provide only heuristics but no guarantees about up to which generation timestamp no more data stream elements will arrive. Since at least from a delay perspective waiting for data stream elements which are out-of-order with respect to their generation timestamps is equivalent to buffering data stream elements to order them with respect to their generation timestamps, this statement can be transferred to buffering data stream elements to order them with respect to their sequence numbers. Therefore, the assumption posed in Section 5.3.2 implicitly poses the assumption that there is a known time for which the processor has to buffer incoming data stream elements and thus a known time bound for late arrivals. That is, we assume in our model (and thus require from every system implementation and scenario) that there is a known time bound for late arrivals.
Moreover, we assume in our model that this time bound is small enough to meet the real-time demands of the analysis scenario and thus that it is not so large that it introduces intolerable delays as described in [Fli18]. We argue that this is a valid assumption in the team sports analysis scenarios our work focuses on, such as the real-time football analysis scenario described in Section 2.1, since they exhibit no long-lasting network partitions as described in [ABC+15] or other drastic environmental conditions. In fact, conditions that prevent devices from communicating (incl. long-lasting network partitions) have been excluded explicitly in our system model (see Section 5.4.2).

Although all processing time orderings are consistent, they are not agnostic to environmental conditions. There are two reasons for that. First, the processing time ordering is guaranteed to be consistent with the sequence number ordering (see Theorem 6.6 for more details). Thus, it inherits the environmental condition dependency from the sequence number ordering. Second, the processing timestamps a processor assigns to data stream elements it processes are influenced by network and processing conditions (see Section 6.1.3). Hence, the fact if two data stream elements which are ordered consecutively by the total sequence number ordering are not assigned the same processing timestamp (cf. $\tau\left(dse_1, pr_2\right)$ and $\tau\left(dse_2, pr_2\right)$ in Proof 6.2) but also ordered in the partial processing time ordering of a processor (cf. $\tau\left(dse_1, pr_1\right)$ and $\tau\left(dse_2, pr_1\right)$ in Proof 6.2) is processor-specific and depends on the environmental conditions. This is also the reason why it is possible that two processing time orderings are unequal. At this point we want to highlight that Theorem 6.2 only states that the processing time orderings introduced by the processing timestamps of different processors are not guaranteed to be equal. This does not mean that there are no scenarios for which the processing time ordering introduced by multiple processors is equal for some or even all environmental conditions (e.g., since only one element is processed every minute but the processor clocks measure time in milliseconds).

6.2.4 Ingestion Time Ordering

As presented in Section 6.1.3, raw input stream elements can be assigned ingestion timestamps when they are received by the entry component(s) of a data stream analysis system. These ingestion timestamps introduce a partial ordering of the elements of a data stream partition. The ordering is not total since multiple elements of the same data stream partition can be assigned by the entry component(s) of a data stream analysis system with the same ingestion timestamp. If the elements of a data stream partition are received by different entry components[24], there is actual concurrency, i.e., two entry components can actually receive two data stream elements at the very same time. Moreover, the clock of every entry component can assign only timestamps with a certain granularity. Hence, even if all elements of a data stream partition are received by the same entry component, multiple consecutive elements of the data stream partition might be assigned the same ingestion timestamp if the velocity of the raw input stream is too high, i.e., if too many elements are received per time interval.

Regardless of whether a data stream analysis system has a single central entry component or multiple entry components and, in the latter case, whether the clocks of the entry components are identical (as the entry components are deployed on the same machine), synchronized, or unsynchronized, there is a single unambiguous ingestion time ordering per data stream partition for every data stream analysis system. This is due to the fact that, as discussed in Section 6.1.3, the ingestion timestamp of every data stream element is unambiguous when regarding only a single data stream analysis system even if there are multiple ingestion time spaces. We formally define the ingestion time ordering introduced by the ingestion timestamps of a data stream analysis system as follows:

[24] Remember that we assume only that every raw input stream element enters a data stream analysis system only via a single entry component and not that all elements of a data stream partition enter a data stream analysis system via the same entry component (see Section 5.2.2).

Definition 6.13 Ingestion Time Ordering

The ingestion time ordering $\prec_{\mathrm{T}}(ds, k, dsas, \xi)$ is a relation that partially orders the elements of the data stream partition *dsp* identified by the data stream *ds* and the key *k* with respect to the ingestion timestamps the entry components of the data stream analysis system *dsas* have assigned to them. That is, it partially orders the elements in $DSE(ds, k, \xi)$ for every sequence number $\xi \in \mathbb{N}_0$.

$$\prec_{\mathrm{T}}(ds, k, dsas, \xi) = \left\{ \langle dse_1, dse_2 \rangle \;\middle|\; dse_1, dse_2 \in DSE(ds, k, \xi) \land \underbrace{\mathrm{T}(dse_1, dsas) < \mathrm{T}(dse_2, dsas)}_{\text{ordering w.r.t. T at } dsas} \right\}$$

Note that the ingestion time ordering $\prec_{\mathrm{T}}(ds, k, dsas, \xi)$ orders the elements of the data stream partition *dsp* identified by the data stream *ds* and the key *k* only if entry components of the data stream analysis system *dsas* receive the elements of *dsp*. If the elements of *dsp* are not received by entry components of *dsas*, the ingestion timestamp $\mathrm{T}(dse, dsas)$ is undefined for every $dse \in DSE(ds, k, \xi)$ (for every $\xi \in \mathbb{N}_0$) and thus $\prec_{\mathrm{T}}(ds, k, dsas, \xi)$ is an empty set. Hence, since, as defined in Chapter 5, data stream analysis systems only receive raw input stream elements from external devices, the ingestion time ordering is an empty set for every partition of an event, state, or statistics stream (i.e., $\prec_{\mathrm{T}}(ds, k, dsas, \xi) = \emptyset$ if $ds.cat \neq$ "rawInput").

If there is only a single entry component which receives all elements of a raw input stream partition, the order in which this entry component assigns ingestion timestamps to elements of the raw input stream partition and thus the ingestion time ordering of the raw input stream partition depends on the order in which these elements arrive at the entry component. This arrival order depends on the network conditions, such as the transmission delays and the ordering guarantees, between the raw input stream generating devices and the central entry component. Moreover, also the exact ingestion timestamps which the entry component assigns and thus the fact if two consecutive elements of a raw input stream partition are not assigned the same ingestion timestamp and therefore ordered in the partial ingestion time ordering depends on these network conditions (see Section 6.1.3).

This environmental condition dependency remains valid if there are multiple entry components which receive elements of the same raw input stream partition. This is due to the fact that the environmental condition dependent ordering introduced by the ingestion timestamps assigned by a certain entry component to all elements of the raw input stream partition it received is a subset of the ingestion time ordering introduced by the ingestion timestamps assigned by all entry components of a data stream analysis system to all elements of the raw input stream partition. Hence, independent of the number of entry components, the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system depends on environmental conditions.

If a data stream analysis system comprises only a single entry component or multiple entry components whose clocks are identical or synchronized, the ingestion time ordering introduced by the ingestion timestamps assigned to the elements of a raw input stream partition is consistent with the actual arrival order of these elements with respect to the whole data stream analysis system. However, if a data stream analysis system comprises multiple entry components whose clocks are unsynchronized and elements of a raw input stream partition are received by different entry components, the ingestion time ordering is not guaranteed to reflect the actual arrival order. This is due to the fact that if the clocks are unsynchronized every entry component assigns ingestion timestamps from its individual ingestion time space. In the following we investigate the effect of the clock synchronicity[25] on the consistency of the ingestion time ordering with the actual arrival order more formally:

Theorem 6.3 Clock Synchronicity Effect on the Ingestion Time Ordering

The ingestion time ordering introduced by the ingestion timestamps of the entry components of a data stream analysis system is consistent with the actual arrival order with respect to the whole data stream analysis system if the clocks of the entry components are synchronized. However, the ingestion time ordering is not guaranteed to be equal to the actual arrival order.

Proof 6.3 Clock Synchronicity Effect on the Ingestion Time Ordering

To prove the consistency of the ingestion time ordering with the actual arrival order in case of synchronized entry component clocks we regard two elements

[25] If there is only a single entry component, we regard the clock of this entry component to be synchronized with itself. Moreover, we regard multiple entry components which are deployed on the same machine and thus actually share a clock to have (perfectly) synchronized clocks.

(dse_1 and dse_2) of the same raw input stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) received by a data stream analysis system *dsas* with two entry components (ec_1 and ec_2) with synchronized clocks.
Assume that dse_1 is received by any entry component of *dsas* before dse_2 is received by the same or another entry component of *dsas*. Hence, the actual arrival order when regarding the data stream analysis system as a whole is as follows:

$$dse_1 \prec dse_2$$

There are four different cases:

1. dse_1 is received by ec_1 before dse_2 is received by ec_1.
2. dse_1 is received by ec_2 before dse_2 is received by ec_2.
3. dse_1 is received by ec_1 before dse_2 is received by ec_2.
4. dse_1 is received by ec_2 before dse_2 is received by ec_1.

From the fact that the clocks of ec_1 and ec_2 are synchronized we can follow that in every case the ingestion timestamp assigned to dse_1 is smaller than or equal to the ingestion timestamp assigned to dse_2.

$$\tau\,(dse_1, dsas) \leq \tau\,(dse_2, dsas)$$

Thus, according to Definition 6.13, dse_2 is definitely not ordered before dse_1 by means of the ingestion timestamps.

$$\langle dse_2, dse_1 \rangle \notin \prec_{\tau}(ds, k, dsas, \xi) \text{ for all } \xi \in \mathbb{N}_0$$

The same can be shown in a similar way for more data stream elements and more entry components with synchronized clocks.
However, since due to the limited clock granularities the ingestion timestamps of dse_1 and dse_2 can be equal, dse_1 is not guaranteed to be ordered before dse_2, i.e., we cannot follow $\langle dse_1, dse_2 \rangle \in \prec_{\tau}(ds, k, dsas, \xi)$ for any $\xi \in \mathbb{N}_0$.
Hence, the ingestion time ordering is consistent with but not guaranteed to be equal to the actual arrival order if the clocks of the entry components are synchronized. □

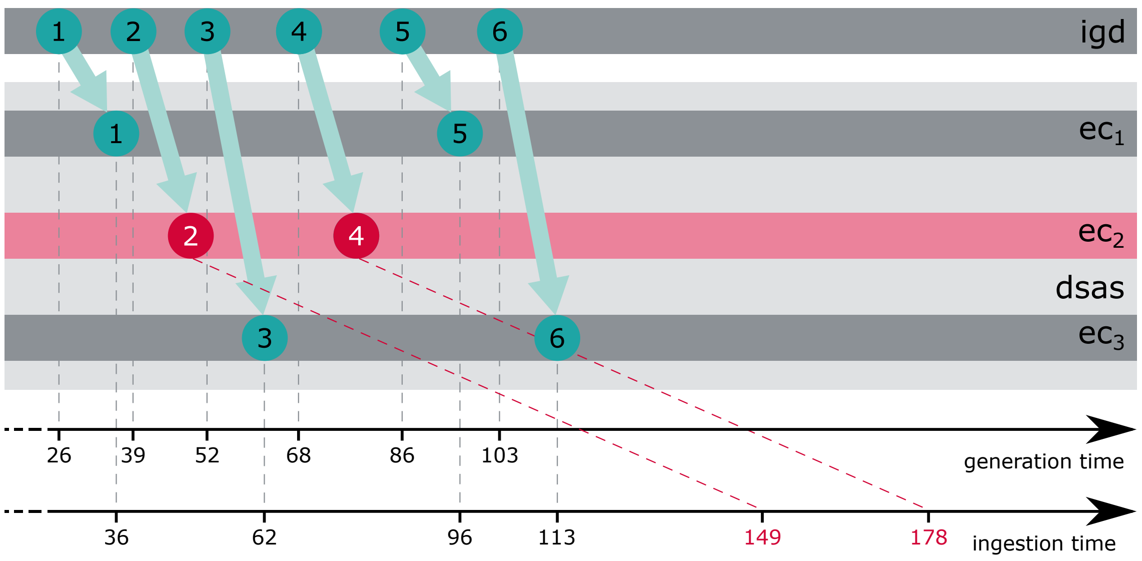


Figure 6.5 **Clock Asynchronicity Effect on the Ingestion Time Ordering.** In the topmost box, the elements of a raw input stream partition are illustrated as dots on a temporal scale with respect to when they were generated by the raw input stream generating device *igd* represented by this box. The mint arrows visualize the transmission of the raw input data as elements of a raw input stream to the entry components (ec_1, ec_2, and ec_3) of a data stream analysis system (*dsas*) that is visualized with a large light-gray box. The other three narrow boxes represent these entry components. In these boxes, the raw input stream elements are illustrated as dots on a temporal scale with respect to when they were received regarding the generation time space. The box that represents entry component ec_2 is highlighted in red as the clock of this component is shifted by 100 milliseconds.

Theorem 6.4 Clock Asynchronicity Effect on the Ingestion Time Ordering

The ingestion time ordering introduced by the ingestion timestamps of the entry components of a data stream analysis system is not guaranteed to be consistent with the actual arrival order with respect to the whole data stream analysis system if the clocks of the entry components are not synchronized.

Proof 6.4 Clock Asynchronicity Effect on the Ingestion Time Ordering

We prove that the ingestion time ordering is not guaranteed to be consistent with the actual arrival order in case of unsynchronized entry component clocks by means of giving a counterexample for consistency illustrated in Figure 6.5. Assume that there is a raw input stream generating device *igd* which generates and immediately emits six raw input stream elements belonging to the same data stream partition, i.e., belonging to the same raw input stream *ds* and having the same key *k*. Further assume that the sequence numbers and generation timestamps of these data stream elements are as follows:

$$\begin{aligned} dse_1.\xi &= 1 \qquad & dse_1.ts &= 26 \\ dse_2.\xi &= 2 & dse_2.ts &= 39 \\ dse_3.\xi &= 3 & dse_3.ts &= 52 \\ dse_4.\xi &= 4 & dse_4.ts &= 68 \\ dse_5.\xi &= 5 & dse_5.ts &= 86 \\ dse_6.\xi &= 6 & dse_6.ts &= 103 \end{aligned}$$

Thus, according to Definition 6.10 and Definition 6.11, the sequence number ordering and the generation time ordering are as follows:

$$dse_1 \prec dse_2 \prec dse_3 \prec dse_4 \prec dse_5 \prec dse_6$$

Assume that there is a data stream analysis system *dsas* with three entry components (ec_1, ec_2, and ec_3) which receive these raw input stream elements. More precisely, assume that dse_1 and dse_5 are received by entry component ec_1, dse_2 and dse_4 are received by entry component ec_2, and dse_3 and dse_6 are received by entry component ec_3. Assume that all network channels exhibit FIFO guarantees and that there is a constant transmission delay of 10 milliseconds between *igd* and every entry component. Hence, the actual arrival order when regarding the data stream analysis system as a whole is as follows:

$$dse_1 \prec dse_2 \prec dse_3 \prec dse_4 \prec dse_5 \prec dse_6$$

Assume that the clocks of ec_1 and ec_3 are in sync with the clock of *igd*. Thus, their ingestion time spaces are equal to the generation time space. Further assume that the clock of ec_2 is not synchronized with the clocks of ec_1, ec_3 and *igd* but shifted by 100 milliseconds. In consequence, the ingestion timestamps the entry components assign to the data stream elements are as follows:

$$\begin{aligned} &\textsc{t}\,(dse_1, dsas) = 36 \qquad &&\textsc{t}\,(dse_2, dsas) = 149 \qquad &&\textsc{t}\,(dse_3, dsas) = 62 \\ &\textsc{t}\,(dse_4, dsas) = 178 &&\textsc{t}\,(dse_5, dsas) = 96 &&\textsc{t}\,(dse_6, dsas) = 113 \end{aligned}$$

Thus, according to Definition 6.13, the ingestion time ordering is as follows:

$$dse_1 \prec dse_3 \prec dse_5 \prec dse_6 \prec dse_2 \prec dse_4$$

Hence, the ingestion time ordering is not guaranteed to be consistent with the actual arrival order if the clocks of the entry components are not synchronized. □

It is straightforward that the consistency part of Theorem 6.3 is valid under all environmental conditions as Proof 6.3 does not pose any assumptions on the conditions. Moreover, Theorem 6.4 only states that the ingestion time ordering introduced by the ingestion timestamps of the entry components of a data stream analysis system and the actual arrival order are not guaranteed to be consistent if the clocks of the entry components are not synchronized. This does not mean that there are no scenarios for which the ingestion time ordering is consistent (or even equal) to the actual arrival order for some or even all environmental conditions even without synchronized clocks. The same is true for the equality part of Theorem 6.3 which states that the ingestion time ordering introduced by the ingestion timestamps of the entry components of a data stream analysis system and the actual arrival order are not guaranteed to be equal even if the clocks of the entry components are synchronized.

When regarding multiple data stream analysis systems which share an input stream it is important to note that the ingestion time orderings introduced by the ingestion timestamps assigned by the entry components of different data stream analysis systems may be inconsistent. This can be caused by the fact that the entry components of one of the data stream analysis system are not synchronized.[26] Moreover, even if the clocks of all entry components of all data stream analysis systems are synchronized, inconvenient network conditions, such as unequal transmission delays or network channels without FIFO guarantees[27], can cause inconsistent ingestion time orderings as different data stream analysis systems might receive raw input stream elements in a different order. In the following we will express and prove this statement more formally:

[26] In addition to the counterexample that we will use in Proof 6.5 it is also possible to construct a counterexample with perfect network conditions (i.e., FIFO guarantees, no message loss, and equal transmission delays) if the clocks of the entry components of one of the data stream analysis systems are not synchronized.

[27] In addition to the counterexample that we will use in Proof 6.5 it is also possible to construct a counterexample in which there is only one raw input stream generating device and both data stream analysis systems contain only one entry component but the network channel between the raw input stream generating device and the entry component of one of the data stream analysis systems (if the local counter or the shared counter approach is used to assign sequence numbers) or between the proxy and the entry component of one of the data stream analysis system (if the proxy approach is used to assign sequence numbers) provides no FIFO guarantees.

Theorem 6.5 Unguaranteed Ingestion Time Ordering Consistency

The ingestion time orderings introduced by the ingestion timestamps of the entry components of different data stream analysis systems are not guaranteed to be consistent. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of the ingestions timestamps assigned by the entry components of data stream analysis system $dsas_1$ that they are not ordered reversely by means of the ingestions timestamps assigned by the entry components of data stream analysis system $dsas_2$.

$$\langle dse_1, dse_2 \rangle \in \prec_{\text{T}}(ds, k, dsas_1, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\text{T}}(ds, k, dsas_2, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

Proof 6.5 Unguaranteed Ingestion Time Ordering Consistency

We prove that the ingestion time orderings introduced by the ingestion timestamps of the entry components of different data stream analysis systems are not guaranteed to be consistent by means of giving a counterexample for consistency. Assume that there are two raw input stream generating devices igd_1 and igd_2 that emit data stream elements of the same data stream partition, i.e., belonging to the same raw input stream ds and having the same key k. Moreover, assume that there is a data stream analysis system $dsas_1$ with two entry components (ec_1 and ec_2) and another data stream analysis system $dsas_2$ with two entry components (ec_3 and ec_4). Assume that igd_1 and igd_2 assign sequence numbers using the shared counter approach (see Section 6.2.1.1) and that the transmission delay between each of the raw input stream generating devices and the coordination service is 10 milliseconds. In addition, assume that the transmission delay between igd_1 and ec_1 is 15 milliseconds, the transmission delay between igd_1 and ec_3 is 100 milliseconds, the transmission delay between igd_2 and ec_2 is 75 milliseconds, and the transmission delay between igd_2 and ec_4 is 10 milliseconds (see Figure 6.6). For the sake of simplicity, further assume that the clocks of igd_1, igd_2, ec_1, ec_2, ec_3, and ec_4 are synchronized and that all network channels provide FIFO guarantees. Assume that dse_1 is generated after 21 milliseconds by igd_1 and immediately sent to ec_1 and ec_3 after the sequence number is assigned (i.e., after 20 milliseconds) and that dse_2 is generated after 51 milliseconds by igd_2 and immediately sent to ec_2 and ec_4 after the sequence number is assigned (i.e., after 20 milliseconds). Thus, the ingestion timestamps

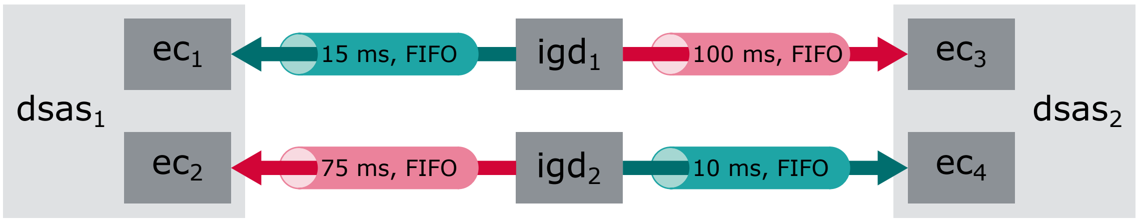


Figure 6.6 Architecture and Network Conditions in the Unguaranteed Ingestion Time Ordering Consistency Proof. The dark-gray boxes illustrate the raw input stream generating devices (igd_1 and igd_2) as well as the entry components (ec_1, ec_2, ec_3, and ec_4). The light-gray boxes visualize which entry components belong to the same data stream analysis system ($dsas_1$ and $dsas_2$). The pipes between the boxes illustrate the network channels between the raw input stream generating devices and the entry components via which the data stream elements are transferred. Network channels with low and high transmission delays are colored in mint and red, respectively. The coordination service is omitted in order to keep the figure as simple as possible.

are as follows:

$$\tau(dse_1, dsas_1) = 56 \qquad \tau(dse_1, dsas_2) = 141$$
$$\tau(dse_2, dsas_1) = 146 \qquad \tau(dse_2, dsas_2) = 81$$

According to Definition 6.13, this means that dse_1 is ordered before dse_2 by means of the ingestion timestamps assigned by the entry components of $dsas_1$ but that dse_2 is ordered before dse_1 and thus reversely by means of the ingestion timestamps assigned by the entry components of $dsas_2$.

$$\langle dse_1, dse_2\rangle \in <_{\tau}(ds, k, dsas_1, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in <_{\tau}(ds, k, dsas_2, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the ingestion time orderings introduced by different data stream analysis systems are not guaranteed to be consistent. □

We want to highlight that Theorem 6.5 states only that the ingestion time orderings introduced by the ingestion timestamps of the entry components of different data stream analysis systems are not guaranteed to be consistent. Needless to say, that it is also possible to construct an example in which the ingestion time orderings of two data stream analysis systems are consistent (or even equal) even if the clocks of the entry components are unsynchronized and the network conditions are inconvenient.

Ordering	Introduced by	Total	Ordering unambiguity
Sequence number ordering	Sequence numbers	Yes	Globally unambiguous
Generation time ordering	Generation timestamps	No	Globally unambiguous
Processing time ordering	Processing timestamps assigned by a processor	No	Unambiguous for every processor. Orderings introduced by the processing timestamps of different processors are not guaranteed to be equal but guaranteed to be consistent.
Ingestion time ordering	Ingestion timestamps assigned by the entry components of a data stream analysis system	No	Unambiguous for every data stream analysis system. Orderings introduced by the ingestion timestamps of different data stream analysis systems may be inconsistent.

Table 6.3 Ordering Overview. The table contrasts the different orderings with respect to multiple properties.

6.2.5 Ordering Comparison

In total there are four different kinds of orderings contrasted in Table 6.3. First, the globally unambiguous but environmental condition dependent sequence number ordering totally orders all elements of a data stream partition with respect to their sequence numbers. Second, the globally unambiguous but environmental condition dependent generation time ordering partially orders the elements of a data stream partition with respect to their generation timestamps. Third, if a data stream partition is consumed and processed by a processor, the processing timestamps assigned by this processor introduce an environmental condition dependent processing time ordering which partially orders the elements of this partition. This processing time ordering is unambiguous for the processor and further guaranteed to be consistent with (although not equal to) the processing time orderings introduced by the processing timestamps assigned by other processors consuming and processing the same data stream partition. Fourth, if a raw input stream partition is consumed by a data stream analysis systems, the ingestion timestamps assigned by the entry components of this data stream analysis system introduce an environmental condition dependent ingestion time ordering which partially orders the elements of this partition. This ingestion time ordering is unambiguous for the data stream analysis system. However, it may be inconsistent with ingestion time orderings introduced by the ingestion timestamps assigned by the entry components of other data stream analysis systems.

As we have presented in our system model the actual analysis of the data stream elements is performed by the processors of the workers (see Section 5.3). Therefore, the most relevant question regarding an ordering is if the processors of a data stream analysis system process the elements of a data stream partition in compliance with this ordering or if the elements might be processed out-of-order with respect to this ordering. In the remainder of this section we will answer this question for the four different ordering types.

It is a matter of course that every processor processes the elements of a data stream partition always in correct order with respect to its local processing time ordering, i.e., to the processing time ordering introduced by the processing timestamps the processor assigns itself. This is due to the fact that a processor processes data stream elements sequentially and that it assigns the processing timestamp to an element it processes not before starting processing the element.

Moreover, since according to our system model every processor processes the elements of a data stream partition not simply in the order in which they arrive at the processor – a behavior which is denoted in recent literature about time notions as processing elements with processing time semantics [ATM+17] – but sequentially ordered by their sequence numbers even if this requires buffering (see Section 5.3.2), every processor is further guaranteed to process the elements of a data stream partition always in correct order with respect to the globally unambiguous sequence number ordering. In consequence, the sequence number ordering is guaranteed to be consistent with but not guaranteed to be equal to the processing time ordering introduced by the processing timestamps assigned by any processor. In the following we express this statement more formally:

Theorem 6.6 Consistency between Sequence Number Ordering and Processing Time Ordering

The globally unambiguous sequence number ordering and the processing time ordering introduced by the processing timestamps assigned by any processor are consistent. That is, two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) which are ordered by means of their globally unambiguous sequence numbers are not ordered reversely by means of the processing timestamps assigned by processor pr, and vice versa.

$$\begin{aligned}
&\forall dse_1, dse_2 \in \widehat{DSE} \wedge pr \in \widehat{PR} : \\
&\qquad \langle dse_1, dse_2 \rangle \in <_\xi(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\
&\qquad \Longrightarrow \langle dse_2, dse_1 \rangle \notin <_\tau\big(ds, k, pr, \xi\big) \text{ for any } \xi \in \mathbb{N}_0
\end{aligned}$$

$$\begin{aligned}
&\forall dse_1, dse_2 \in \widehat{DSE} \wedge pr \in \widehat{PR} : \\
&\qquad \langle dse_1, dse_2 \rangle \in <_\tau\big(ds, k, pr, \xi\big) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\
&\qquad \Longrightarrow \langle dse_2, dse_1 \rangle \notin <_\xi(ds, k, \xi) \text{ for any } \xi \in \mathbb{N}_0
\end{aligned}$$

Proof 6.6 Consistency between Sequence Number Ordering and Processing Time Ordering

As defined in our system model, every processor processes the elements of a data stream partition sequentially with respect to their sequence numbers (see Section 5.3.2). Moreover, as defined in Section 6.1.3, every processor assigns every element a processing timestamp created by its local clock when it starts processing the element.

Together these definitions exclude that a data stream element *dse*$_1$ whose sequence number is smaller than the sequence number of another data stream element *dse*$_2$ that belongs to the same data stream partition is assigned a greater processing timestamp than *dse*$_2$ at processor *pr*.

$$\begin{aligned}
&\nexists\, dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = dse_2.ds \text{ and } dse_1.k = dse_2.k : \\
&\qquad dse_1.\xi < dse_2.\xi \wedge \tau\left(dse_1, pr\right) > \tau\left(dse_2, pr\right)
\end{aligned}$$

Moreover, these definitions exclude that a data stream element *dse*$_1$ whose processing timestamp is smaller than the processing timestamp of another data stream element *dse*$_2$ that belongs to the same data stream partition has a greater sequence number than *dse*$_2$.

$$\begin{aligned}
&\nexists\, dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = dse_2.ds \text{ and } dse_1.k = dse_2.k : \\
&\qquad \tau\left(dse_1, pr\right) < \tau\left(dse_2, pr\right) \wedge dse_1.\xi > dse_2.\xi
\end{aligned}$$

In consequence, the sequence number ordering and the processing time ordering are guaranteed to be consistent. □

Theorem 6.7 Processing Time Ordering is a Subset of the Sequence Number Ordering

The processing time ordering introduced by the processing timestamps assigned by a processor *pr* is a subset of the globally unambiguous sequence number ordering.

$$\prec_\tau(ds,k,pr,\xi) \subseteq \prec_\xi(ds,k,\xi) \text{ for all } ds \in \widehat{DS},\ k \in Dom_k,\ pr \in \widehat{PR}, \text{ and } \xi \in \mathbb{N}_0$$

That is, we can follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of the processing timestamps assigned by processor *pr* that they are ordered in the same way by means of their globally unambiguous sequence numbers, but not vice versa.

$$\begin{aligned}
&\forall dse_1, dse_2 \in \widehat{DSE} \wedge pr \in \widehat{PR} :\\
&\quad \langle dse_1, dse_2\rangle \in \prec_\tau(ds,k,pr,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi\\
&\quad \Longrightarrow \langle dse_1, dse_2\rangle \in \prec_\xi(ds,k,\xi) \text{ for any } \xi \in \mathbb{N}_0
\end{aligned}$$

$$\begin{aligned}
&\langle dse_1, dse_2\rangle \in \prec_\xi(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi\\
&\not\Longrightarrow \langle dse_1, dse_2\rangle \in \prec_\tau(ds,k,pr,\xi) \text{ for any } \xi \in \mathbb{N}_0
\end{aligned}$$

Proof 6.7 Processing Time Ordering is a Subset of the Sequence Number Ordering

We proved already that the globally unambiguous sequence number ordering and the processing time ordering introduced by the processing timestamps assigned by any processor are guaranteed to be consistent (see Proof 6.6). Moreover, we know that the sequence number ordering is total since the sequence number of a data stream element is unique per data stream partition (see Section 6.2.1). Instead, processing time orderings are only partial as every clock has a limited granularity and thus processors might assign the same processing timestamp to multiple consecutive data stream elements (see Section 6.2.3).
Hence, every data stream element dse_1 that is ordered with respect to its processing timestamp assigned by a processor *pr* before another data stream element dse_2 that belongs to the same data stream partition is guaranteed to be ordered in the same way with respect to its globally unambiguous sequence number.

$$\begin{aligned}&\forall dse_1, dse_2 \in \widehat{DSE} \land pr \in \widehat{PR} :\\ &\quad \langle dse_1, dse_2\rangle \in \prec_\tau\big(ds,k,pr,\xi\big) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi\\ &\quad\Longrightarrow \langle dse_1, dse_2\rangle \in \prec_\xi(ds,k,\xi) \text{ for any } \xi \in \mathbb{N}_0\end{aligned}$$

However, there can be a data stream element dse_1 that is ordered with respect to its globally unambiguous sequence number before another data stream element dse_2 that belongs to the same data stream partition but which was assigned the same processing timestamp at pr as dse_2.

$$\begin{aligned}&\langle dse_1, dse_2\rangle \in \prec_\xi(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi\\ &\not\Longrightarrow \langle dse_1, dse_2\rangle \in \prec_\tau\big(ds,k,pr,\xi\big) \text{ for any } \xi \in \mathbb{N}_0\end{aligned}$$

In consequence, the processing time ordering is guaranteed to be a subset of the sequence number ordering. □

Note that Theorem 6.7 states only that the processing time ordering introduced by the processing timestamps assigned by a processor is a subset of the globally unambiguous sequence number ordering but not that this subset is strict. There are also scenarios in which the processing time ordering and the sequence number ordering are equal since the processor did never assign the same processing timestamp to two consecutive data stream elements.

In contrast, processors might process the elements of a data stream partition out-of-order regarding their generation timestamps and thus not in correct order with respect to the globally unambiguous generation time ordering. This is due to the fact that depending on the approach used for assigning sequence numbers (see Section 6.2.1.1) and depending on the environmental conditions the sequence number ordering and the generation time ordering of a data stream partition might be inconsistent and that, as mentioned above, processors process the elements of a data stream partition sequentially ordered by their sequence numbers. For instance, the generation time ordering and the sequence number ordering can be inconsistent if the proxy approach is used to assign sequence numbers and the network channels do not provide FIFO guarantees and/or do not exhibit equal and constant transmission delays.[28] In consequence, the generation time ordering is neither guaranteed to be consistent with the globally

[28] In addition to the counterexample that we will use in Proof 6.8 it is possible to construct a similar counterexample in which the network channels provide FIFO guarantees but do not exhibit constant and equal transmission delays if we add a second raw input stream generating device which emits elements of the same raw input stream partition.

unambiguous sequence number ordering nor with the processing time ordering introduced by the processing timestamps assigned by a certain processor. In the following we express and prove this statement formally:

Theorem 6.8 Unguaranteed Consistency between Sequence Number Ordering and Generation Time Ordering

The globally unambiguous sequence number ordering and the globally unambiguous generation time ordering are not guaranteed to be consistent. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of their globally unambiguous sequence numbers that they are not ordered reversely by means of their globally unambiguous generation timestamps, and vice versa.

$$\langle dse_1, dse_2 \rangle \in \prec_{\xi}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{ts}(ds, k, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

$$\langle dse_1, dse_2 \rangle \in \prec_{ts}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\xi}(ds, k, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

Proof 6.8 Unguaranteed Consistency between Sequence Number Ordering and Generation Time Ordering

We prove that the globally unambiguous sequence number ordering is not guaranteed to be consistent with the globally unambiguous generation time ordering by means of giving a counterexample for consistency. Assume that there is a raw input stream generating device *igd* that emits data stream elements belonging to the raw input stream *ds* and having the key *k*. Moreover, assume that the proxy approach is used to assign sequence numbers to the raw input stream elements (see Section 6.2.1.1). For the sake of simplicity assume that there is only a single central proxy which receives all data stream elements, assigns sequence numbers to them, and in case of raw input stream elements forwards them to the entry components of the data stream analysis systems.
Assume that *igd* generates dse_1 after 10 milliseconds and immediately sends it to the central proxy. Moreover, assume that after 15 milliseconds *igd* generates dse_2 and immediately sends it to the central proxy. The generation timestamps

of dse_1 and dse_2 are as follows:

$$dse_1.ts = 10 \qquad dse_2.ts = 15$$

Further assume that the network channel between *igd* and the central proxy does not provide FIFO guarantees and that the central proxy thus receives dse_2 before it receives dse_1. In consequence, the sequence numbers of dse_1 and dse_2 are as follows:

$$dse_1.\xi = 2 \qquad dse_2.\xi = 1$$

According to Definition 6.10 and Definition 6.11, this means that dse_1 is ordered before dse_2 by means of the generation timestamps assigned by the raw input stream generating devices but that dse_2 is ordered before dse_1 and thus reversely by means of the sequence numbers assigned by the central proxy.

$$\langle dse_1, dse_2 \rangle \in \prec_{ts}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1 \rangle \in \prec_{\xi}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the sequence number ordering and the generation time ordering are not guaranteed to be consistent. □

Theorem 6.9 Unguaranteed Consistency between Generation Time Ordering and Processing Time Ordering

The globally unambiguous generation time ordering and the processing time ordering introduced by the processing timestamps assigned by a processor are not guaranteed to be consistent. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of their globally unambiguous generation timestamps that they are not ordered reversely by means of the processing timestamps assigned by processor *pr*, and vice versa.

$$\langle dse_1, dse_2 \rangle \in \prec_{ts}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Rightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\tau}(ds,k,pr,\xi) \text{ for any } \xi \in \mathbb{N}_0$$

$$\langle dse_1, dse_2 \rangle \in \prec_{\tau}(ds,k,pr,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Rightarrow \langle dse_2, dse_1 \rangle \notin \prec_{ts}(ds,k,\xi) \text{ for any } \xi \in \mathbb{N}_0$$

Proof 6.9 Unguaranteed Consistency between Generation Time Ordering and Processing Time Ordering

We prove that the globally unambiguous generation time ordering is not guaranteed to be consistent with the processing time ordering introduced by the processing timestamps assigned by a processor by means of giving a counterexample for consistency. More precisely, we extend the counterexample given in Proof 6.8.

In this example, there are two data stream elements (dse_1 and dse_2) with the following generation timestamps and sequence numbers:

$$\begin{array}{ll} dse_1.ts = 10 & dse_2.ts = 15 \\ dse_1.\xi = 2 & dse_2.\xi = 1 \end{array}$$

Moreover, assume that there is a processor pr which processes both elements sequentially with respect to their sequence numbers as defined in Section 5.3.2 and assigns the following processing timestamps to them:

$$\tau\left(dse_1, pr\right) = 35 \qquad \tau\left(dse_2, pr\right) = 34$$

According to Definition 6.11 and Definition 6.12, this means that dse_1 is ordered before dse_2 by means of the generation timestamps assigned by the raw input stream generating devices but that dse_2 is ordered before dse_1 and thus reversely by means of the processing timestamps assigned by pr.

$$\langle dse_1, dse_2 \rangle \in \prec_{ts}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1 \rangle \in \prec_{\tau}\left(ds, k, pr, \xi\right) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the processing time ordering and the generation time ordering are not guaranteed to be consistent. □

Note that inconsistencies between the sequence number ordering, the processing time ordering, and the generation time ordering are not only introduced by an additional proxy. Instead, as we show in more detail in Appendix A, there can also be inconsistencies if another sequence number assignment approach is used. Of course there are also scenarios in which the sequence number ordering, the processing time ordering, and the generation time ordering are consistent or even equal. For instance, if all elements of a raw input stream partition were emitted immediately after the contained raw input data are generated, if all elements of this partition were generated by a single raw input stream generating device, and if the local counter approach was used without further coordination (see Section 6.2.1.1) to assign sequence numbers to elements of this partition, the sequence number ordering, the processing time ordering, and the generation time ordering of this partition would be consistent. Nevertheless, the general statements expressed in Theorem 6.8 and Theorem 6.9 hold as they only state that the orderings are not guaranteed to be consistent.

Literature Discussion 6.6 Out-of-Order with respect to Generation Timestamps

The fact that data stream elements might be processed out-of-order with respect to their generation timestamps when this is not explicitly handled is also stated and discussed in recent literature [ABC+15; ATM+17; CKE+15; Fli18]. Affetti et al. [ATM+17] state that besides clock skews between raw input stream generating devices which are excluded in our model as we assume their clocks to be synchronized (see Section 6.1.1) transmission delays can be the reason for data stream elements to arrive at the processor out-of-order regarding their generation timestamps. Moreover, Affetti et al. [ATM+17] indicate that network channels without FIFO guarantees can cause reorderings. Please note that these factors (i.e., clock synchronicities, transmission delays, and FIFO guarantees) have also been frequently used by us to construct scenarios for investigating and discussing consistencies between orderings. For instance, Proof 6.8 uses a counterexample in which a network channel does not provide FIFO guarantees to prove that the sequence number ordering is not guaranteed to be consistent with the generation time ordering.

In fact, we got the idea to define, analyze, and compare the orderings which are introduced by the different timestamps and sequence numbers while reading [ABC+15; ATM+17; CKE+15; Fli18] and used this literature as a starting point. However, we want to highlight, that the extensive time notion and ordering discussion which we present in this thesis goes far beyond the few statements that are given in [ABC+15; ATM+17; CKE+15; Fli18].

Additionally, processors of a data stream analysis system might process the elements of a raw input stream partition out-of-order regarding the ingestion timestamps assigned by the entry components of this data stream analysis system and thus not in correct order with respect to the ingestion time ordering introduced by the ingestion timestamps the entry components of the data stream analysis system have assigned. This can be caused by the fact that the clocks of the entry components of the data stream analysis system are not synchronized (see Proof 6.4). Moreover, this can happen if the network channels between the components assigning the sequence numbers (no matter if the local counter approach, the proxy approach, or any other approach is used) and the entry components which assign ingestion timestamps to elements of a raw input stream partition do not provide FIFO guarantees and exhibit constant and equal transmission delays. Hence, the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system is neither guaranteed to be consistent with the globally unambiguous sequence number ordering nor with the processing time ordering of a processor of this data stream analysis system. In the following we express and prove this statement formally:

Theorem 6.10 Unguaranteed Consistency between Sequence Number Ordering and Ingestion Time Ordering

The globally unambiguous sequence number ordering and the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system are not guaranteed to be consistent. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of their globally unambiguous sequence numbers that they are not ordered reversely by means of the ingestion timestamps assigned by the entry components of data stream analysis system *dsas*, and vice versa.

$$\begin{aligned}&\langle dse_1, dse_2\rangle \in \prec_\xi(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\ &\quad \not\Longrightarrow \langle dse_2, dse_1\rangle \notin \prec_{\text{T}}(ds,k,dsas,\xi) \text{ for any } \xi \in \mathbb{N}_0\end{aligned}$$

$$\begin{aligned}&\langle dse_1, dse_2\rangle \in \prec_{\text{T}}(ds,k,dsas,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi \\ &\quad \not\Longrightarrow \langle dse_2, dse_1\rangle \notin \prec_\xi(ds,k,\xi) \text{ for any } \xi \in \mathbb{N}_0\end{aligned}$$

Proof 6.10 Unguaranteed Consistency between Sequence Number Ordering and Ingestion Time Ordering

We prove that the globally unambiguous sequence number ordering is not guaranteed to be consistent with the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system *dsas* by means of giving a counterexample for consistency. Assume that there is a single central entry component *ec* which receives all elements of

a certain raw input stream partition, i.e., belonging to raw input stream *ds* and having key *k*. Moreover, assume that there is a single component which assigns sequence numbers to all elements of this partition and which emits these elements immediately in correct order with respect to their sequence numbers. This component can be either the raw input stream generating devices which generates the raw input data or a proxy depending on the sequence number assigning approach (see Section 6.2.1.1).
Assume that the above mentioned component first assigns a sequence number to dse_1 and immediately sends it to *ec*, and then assigns a sequence number to dse_2 and immediately sends it to *ec*. Hence, the sequence number of dse_1 is smaller than the sequence number of dse_2:

$$dse_1.\xi < dse_2.\xi$$

Further assume that the network channel between the above mentioned component and *ec* does not provide FIFO guarantees and that *ec* thus receives dse_2 some milliseconds before it receives dse_1. In consequence, the ingestion timestamps of dse_1 is greater than the ingestion timestamp of dse_2:

$$\tau\,(dse_1, dsas) > \tau\,(dse_2, dsas)$$

According to Definition 6.10 and Definition 6.13, this means that dse_1 is ordered before dse_2 by means of the sequence numbers but that dse_2 is ordered before dse_1 and thus reversely by means of the ingestion timestamps assigned by *ec*.

$$\langle dse_1, dse_2\rangle \in \prec_\xi(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in \prec_\tau(ds, k, dsas, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the sequence number ordering and the ingestion time ordering are not guaranteed to be consistent. □

Theorem 6.11 Unguaranteed Consistency between Ingestion Time Ordering and Processing Time Ordering

The ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system and the processing time ordering introduced by the processing timestamps assigned by a processor of this data stream analysis system are not guaranteed to be consistent.

That is, we cannot follow from the fact that two elements (*dse_1* and *dse_2*) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of their ingestion timestamps assigned by the entry components of data stream analysis system *dsas* that they are not ordered reversely by means of the processing timestamps assigned by processor *pr* of *dsas*, and vice versa.

$$\langle dse_1, dse_2 \rangle \in \prec_{\mathrm{T}}(ds, k, dsas, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\tau}(ds, k, pr, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

$$\langle dse_1, dse_2 \rangle \in \prec_{\tau}(ds, k, pr, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\mathrm{T}}(ds, k, dsas, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

Proof 6.11 Unguaranteed Consistency between Ingestion Time Ordering and Processing Time Ordering

We prove that the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system is not guaranteed to be consistent with the processing time ordering introduced by the processing timestamps assigned by a processor of this data stream analysis system by means of giving a counterexample for consistency. More precisely, we extend the counterexample given in Proof 6.10.
In this example, there are two data stream elements (*dse_1* and *dse_2*) for which we have the following sequence number and ingestion timestamp information:

$$dse_1.\xi < dse_2.\xi$$
$$\mathrm{T}(dse_1, dsas) > \mathrm{T}(dse_2, dsas)$$

Moreover, assume that *dsas* has a processor *pr* which processes both elements sequentially with respect to their sequence numbers as defined in Section 5.3.2 and assigns the following processing timestamps to them:

$$\tau(dse_1, pr) = 22 \qquad \tau(dse_2, pr) = 23$$

According to Definition 6.12 and Definition 6.13, this means that *dse_1* is ordered before *dse_2* by means of the processing timestamps assigned by *pr* but that *dse_2* is ordered before *dse_1* and thus reversely by means of the ingestion timestamps assigned by *ec* of *dsas*.

$$\langle dse_1, dse_2\rangle \in \prec_\tau(ds,k,pr,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in \prec_\mathrm{T}(ds,k,dsas,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the processing time ordering and the ingestion time ordering are not guaranteed to be consistent. □

We want to highlight that Theorem 6.10 only states that the ingestion time ordering and the sequence number ordering are not guaranteed to be consistent but not that there are no scenarios in which they are consistent or even equal. The same is true for Theorem 6.11, i.e., for the ingestion time ordering and the processing time ordering.

A substantial implication of our system model is that data stream elements which are out-of-order with respect to their generation timestamps or their ingestion timestamps cannot be handled by means of an additional buffering mechanism. If the generation time ordering is already consistent with the sequence number ordering, the buffering mechanism of the processor that ensures processing data stream elements in sequence number order results already in processing all data stream elements in correct order with respect to their generation timestamps. However, if the generation time ordering conflicts with the sequence number ordering, reordering the data stream elements and processing them in correct order with respect to their generation timestamps violates our system model which defines that every processor processes the elements of every data stream partition sequentially ordered by their sequence numbers (see Section 5.3.2). The same is true for the ingestion time ordering.

Literature Discussion 6.7 Ordering Data Stream Elements with respect to Generation or Ingestion Timestamps

With this restriction we deviate from recent literature discussing stream time notions. [ABC+15], [CKE+15], and [Fli18] do not introduce and discuss sequences numbers at all. [ATM+17] discusses sequence numbers (denoted as "id") in the context of count windows but does not enforce processing data stream elements in correct order with respect to them. In contrast, this literature allows reordering data stream elements with respect to their generation timestamps and ingestion timestamps in order to process them with generation time semantics (originally called "event time semantics" in [ATM+17]) and ingestion time semantics, respectively, and discusses the consequences of this

reordering process.
A special focus of this discussion is put on the processing delays introduced by waiting to reorder data stream elements.[29] Neglecting reordering and thus processing elements in correct order with respect to their processing timestamps is said to exhibit the lowest latency [Fli18]. Moreover, reordering data stream elements with respect to their ingestion timestamps is said to introduce fewer additional processing delays than reordering data stream elements with respect to their generation timestamps [CKE+15].
While we agree with the former processing delay statement made in [Fli18], we argue that the latter processing delay statement made in [CKE+15] would not be true for our generic system model and our definition of ingestion timestamps if we dropped the existence of sequence numbers and thus also the requirement that processors process data stream elements sequentially ordered by their sequence numbers. For the processing delay to be smaller if data stream elements are reordered with respect to their ingestion timestamps instead of their generation timestamps, the delay required to wait for late arrivals of the data stream elements with respect to their ingestion timestamps has to be smaller than the delay required to wait for late arrivals of the data stream elements with respect to their generation timestamps. Although this might be often the case in practice, this is not true in general. Instead it is even possible that the network conditions between the raw input stream generating devices and the entry components compensate the network conditions between the entry components and the processors leading to no late arrivals with respect to generation timestamps but still to late arrivals with respect to ingestion timestamps as these are assigned first at the entry components (see Example 6.5 for a concrete example). Hence, regarding the processing delay introduced by waiting for late data stream elements there is no clear benefit from reordering data stream elements with respect to their ingestion timestamps instead of their generation timestamps.

---

[29] Note that, as already mentioned in Literature Discussion 6.5, from a delay perspective waiting for data stream elements which are late with respect to timestamps of a specific time notion is equivalent to reordering elements with respect to this time notion. Thus, we do not distinguish between statements about the delay introduced by waiting for late elements and statements about the delay introduced by reordering elements.

Example 6.5 Lower Processing Delay for Reordering Data Stream Elements with respect to their Generation Timestamps than with respect to their Ingestion Timestamps

To back the statement we made in Literature Discussion 6.7 by means of giving an example which shows that reordering data stream elements with respect to their ingestion timestamps can introduce a higher processing delay than reordering them with respect to their generation timestamps we drop the assumption that the elements of every data stream partition are processed in correct order with respect to their sequence numbers (see Section 5.3.2) from our model.

Assume that there are two raw input stream generating devices igd_1 and igd_2 that emit data stream elements of the same raw input stream partition. Moreover, there is a data stream analysis system *dsas* consisting of two entry components ec_1 and ec_2 as well as a processor *pr* which processes all data stream elements emitted by igd_1 and igd_2. Assume that igd_1 and igd_2 assign sequence numbers using the shared counter approach (see Section 6.2.1.1) and that the transmission delay between each of the raw input stream generating devices and the coordination service is 10 milliseconds. In addition, assume that the transmission delay between igd_1 and ec_1 is 110 milliseconds, the transmission delay between igd_2 and ec_2 is 15 milliseconds, the transmission delay between ec_1 and *pr* is 25 milliseconds, and the transmission delay between ec_2 and *pr* is 120 milliseconds (see Figure 6.7). For the sake of simplicity, assume for this example that the clocks of igd_1, igd_2, ec_1, ec_2 and *pr* are synchronized and that all network channels provide FIFO guarantees.

Assume that after 91 milliseconds igd_1 generates dse_1 which is immediately sent to ec_1 after the sequence number is assigned and that ec_1 immediately forwards the data stream element to *pr*. Thus, *pr* receives dse_1 after 246 (91+10+10+110+25) milliseconds.

Assume that after 101 milliseconds igd_2 generates dse_2 which is immediately sent to ec_2 after the sequence number is assigned and that ec_2 immediately forwards the data stream element to *pr*. Thus, *pr* receives dse_2 after 256 (101+10+10+15+120) milliseconds.

The generation timestamp of dse_1 and dse_2 is 91 and 101, respectively. Hence, *pr* receives all data stream elements in the correct order with respect to their generation timestamps. In consequence, no additional processing delay is required to wait for elements which are late regarding their generation timestamps.

The ingestion timestamp of dse_1 and dse_2 is 221 (91+10+10+110) and 136 (101+10+10+15), respectively. Hence, *pr* does not receive all data stream elements in the correct order with respect to their ingestion timestamps. Therefore, *pr* has to wait for 10 (256-246) milliseconds after it has received dse_1 to receive dse_2 before it can process these two data stream elements in correct order with respect to their ingestion timestamps.
The alert reader might ask himself/herself, how *pr* can know that it can directly process dse_1 to process both elements in correct order with respect to their generation timestamps but that is has to wait until it has received and processed dse_2 to process them in correct order with respect to their ingestion timestamps. Actually, as it has been discussed in literature [ABC⁺15; Fli18] already for generation timestamps, *pr* can never know that there will be no more very late arrivals with respect to any time notion (if there is no known time bound for late arrivals; see Literature Discussion 6.5). Thus, after some time (potentially determined by a heuristical mechanism like watermarking [ABB⁺13]) *pr* has to suppose that there will be no more late arrivals and just process a data stream element.[30]
Nevertheless, this example shows that there are scenarios in which it would be safe to configure a worker to instruct its processors to wait for less time to process data stream elements in correct order with respect to their generation timestamps than with respect to their ingestion timestamps. More precisely, in this example it is possible to refrain from waiting at all and thus to introduce no additional processing delay but still to process data stream elements in correct order with respect to their generation timestamps. In contrast, the processor has to wait for at least 10 milliseconds and thus introduce an additional processing delay of at least 10 milliseconds in order to prevent processing data stream elements out-of-order with respect to their ingestion timestamps.

As already indicated in Proof 6.4 and Example 6.5, the generation time ordering of a raw input stream partition is not guaranteed to be consistent with the ingestion time ordering introduced by the ingestion timestamps assigned by the entry

[30] If *pr* supposed to early that there will be no more late data stream elements and a data stream element which is late arrives, *pr* can ignore this element, simply process it out-of-order, or (if possible) use it to update an already emitted output stream element as suggested in [ABC⁺15]. Note that explaining and discussing these options in more detail is out of the scope of this thesis since we defined in our model that incoming data stream elements are processed in correct with respect to their sequence numbers and that this is always possible since there is a known time bound for late arrivals (see Section 5.3.2 and Literature Discussion 6.5).

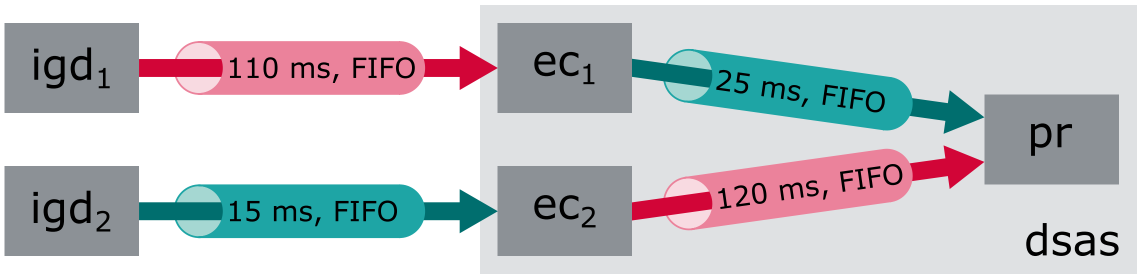


Figure 6.7 Architecture and Network Conditions in the Lower Processing Delay Example. The dark-gray boxes illustrate the raw input stream generating devices (igd_1 and igd_2) as well as the entry components (ec_1 and ec_2) and the processor (pr) of the data stream analysis system ($dsas$) visualized with a light-gray box. The pipes between the dark-gray boxes illustrate the network channels between the raw input stream generating devices and the entry components as well as between the entry components and the processor via which the data stream elements are transferred. Network channels with low and high transmission delays are colored in mint and red, respectively. The coordination service is omitted in order to keep the figure as simple as possible.

components of a data stream analysis system. Inconsistencies can be caused by unfavorable properties of the network channels between the raw input stream generating devices and the entry components (e.g., lack of FIFO guarantees or unequal transmission delays) and by unsynchronized entry component clocks. In the following, we will formally express and prove this consistency relation:

Theorem 6.12 Unguaranteed Consistency between Generation Time Ordering and Ingestion Time Ordering

The globally unambiguous generation time ordering and the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system are not guaranteed to be consistent. That is, we cannot follow from the fact that two elements (dse_1 and dse_2) of the same data stream partition (i.e., $dse_1.ds = dse_2.ds = ds$ and $dse_1.k = dse_2.k = k$) are ordered by means of their globally unambiguous generation timestamps that they are not ordered reversely by means of the ingestion timestamps assigned by the entry components of data stream analysis system $dsas$, and vice versa.

$$\langle dse_1, dse_2 \rangle \in \prec_{ts}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{\tau}(ds, k, dsas, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

$$\langle dse_1, dse_2 \rangle \in \prec_{\tau}(ds, k, dsas, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\not\Longrightarrow \langle dse_2, dse_1 \rangle \notin \prec_{ts}(ds, k, \xi) \text{ for any } \xi \in \mathbb{N}_0$$

Proof 6.12 Unguaranteed Consistency between Generation Time Ordering and Ingestion Time Ordering

We prove that the globally unambiguous generation time ordering is not guaranteed to be consistent with the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system *dsas* by means of giving a counterexample for consistency. Assume that there is a raw input stream generating device *igd* that emits data stream elements belonging to the raw input stream *ds* and having the key *k*. Moreover, assume that *igd* uses the local counter approach to assign sequences numbers and that there is a single central entry component *ec* which receives all data stream elements of this partition.
Assume that *igd* generates dse_1 after 10 milliseconds and immediately sends it to *ec*. Moreover, assume that after 15 milliseconds *igd* generates dse_2 and immediately sends it to *ec*. The generation timestamps of dse_1 and dse_2 are as follows:

$$dse_1.ts = 10 \qquad dse_2.ts = 15$$

Further assume that the network channel between *igd* and *ec* does not provide FIFO guarantees and that *ec* thus receives dse_2 some milliseconds before it receives dse_1. In consequence, the ingestion timestamps of dse_1 is greater than the ingestion timestamp of dse_2:

$$\tau\,(dse_1, dsas) > \tau\,(dse_2, dsas)$$

According to Definition 6.11 and Definition 6.13, this means that dse_1 is ordered before dse_2 by means of the generation timestamps but that dse_2 is ordered before dse_1 and thus reversely by means of the ingestion timestamps assigned by *ec*.

$$\langle dse_1, dse_2\rangle \in \prec_{ts}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in \prec_{\tau}(ds, k, dsas, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

In consequence, the generation time ordering and the ingestion time ordering are not guaranteed to be consistent. □

Note that Theorem 6.12 only states that the generation time ordering of a raw input stream partition is not guaranteed to be consistent with the ingestion time ordering introduced by the ingestion timestamps assigned by the entry components of a data stream analysis system. There are also scenarios in which the

Ordering 1	Ordering 2	Guaranteed Consistency	Theorem
Processing time ordering	Sequence number ordering	Yes	Theorem 6.6
Processing time ordering	Generation time ordering	No	Theorem 6.9
Processing time ordering	Ingestion time ordering	No	Theorem 6.11
Sequence number ordering	Generation time ordering	No	Theorem 6.8
Sequence number ordering	Ingestion time ordering	No	Theorem 6.10
Generation time ordering	Ingestion time ordering	No	Theorem 6.12

Table 6.4 Ordering Consistencies. The table summarizes the consistency guarantees between the different orderings.

globally unambiguous generation time ordering and the ingestion time ordering introduced by the ingestion timestamps of the entry components of a data stream analysis system are consistent or even equal.

To conclude this section, Table 6.4 gives an overview about the existing or non-existing consistency guarantees between the orderings introduced by the sequence numbers, the generation timestamps, the ingestion timestamps assigned by the entry components of a data stream analysis system, and the processing timestamps assigned by a processor of the same data stream analysis system.

6.3 Simultaneousness

As we have presented in Section 5.3.2, processors do not process multiple data stream elements at the same time. Hence, there is no actual concurrency inside a processor. Moreover, as we have discussed in Section 6.1.4, processing timestamps assigned by different processors are not comparable since they specify moments in time with respect to different[31] processing time spaces. Therefore, it is impossible to determine if two arbitrary data stream elements are processed by two different processors at the very same time or within some time bound (e.g., within 100 milliseconds).

However, there is still a notion of simultaneousness when regarding the moment in time the raw input data contained in a raw input stream element have

[31] Remember that only processors which are deployed on the same machine have the same processing time space as they share a clock and that this is a special case and not the norm.

been generated, the moment in time an event reflected by an atomic event stream element has occurred, the moment in time the update of a non-atomic event contained in a non-atomic event stream element refers to, the moment in time the state information contained in a state stream element refers to, and the moment in time the statistics data contained in a statistics stream element refer to. For instance, two player sensors might perform a measurement and generate a player sensor input stream element exactly or at least almost synchronously (e.g., within 20 milliseconds). Moreover, a successful pass between two players might happen while one of these or another player enters a penalty box and thus both events might occur at approximately the same time (e.g., within a second). Furthermore, the latest team area states for both teams of a football match refer typically to moments in time which differ only slightly (less then 100 milliseconds) if at all. The same is true for the latest fitness statistics for the players and the teams of a football match. This simultaneousness can be deduced from the generation timestamp *ts* contained in every data stream element. As defined in Section 6.1.1, all generation timestamps are from the same consistent generation time space and thus comparable. Based on that we denote two data stream elements as *simultaneous* if their generation timestamps differ less than a given time bound. Otherwise we denote them as *sequential*.

Assessing the simultaneousness and sequentiality of data stream elements with respect to their generation timestamps is very expedient for many analyses. For instance, in our real-time football analysis application we specify a threshold which the generation time difference between the event stream elements representing the last detected set play event and a newly detected set play event has to exceed in order to avoid duplicate set play event detections (see Section 9.2.1.5). This lower threshold can be regarded as the time bound with respect to which an event stream element representing a newly detected set play event would be simultaneousness (and thus not sequential) to the event stream element representing the last detected set play event. Moreover, if we extend the offside worker (see Section 9.2.1.7) to detect offside traps we will specify a threshold which the generation time difference between a defense forward push event stream element and a kick event stream element has to deceed for qualifying both events to trigger the detection of an offside trap event. This upper threshold can be regarded as the time bound with respect to which data stream elements have to be simultaneous in order to be considered as candidates for causing an offside trap event detection.

As the data stream analysis is fully separated by means of the key (see Section 5.3.1) we also define simultaneousness and sequentiality in a way that only data stream elements with the same key can be simultaneous or sequential. We argue that this is rational from a semantic point of view as the key is typically used to split the analysis in a way required by the application. For instance, in our real-time football analysis application (see Chapter 9) we use a match identifier as the key in order to analyze the data for each match separately and we require also simultaneousness and sequentiality to be defined on a per-match basis and thus data stream elements (e.g., representing successful pass or penalty box entry events) to be only regarded as simultaneous or sequential if they belong to the same match. Consequently, we formally define simultaneous and sequential data stream elements as follows:

Definition 6.14 Simultaneous Data Stream Elements

Two data stream elements dse_1 and $dse_2 \neq dse_1$ of the same data stream ($dse_1.ds = dse_2.ds$) or different data streams ($dse_1.ds \neq dse_2.ds$) are simultaneous with respect to a given time bound Δts if they have the same key k and the difference between their generation timestamps is smaller than the time bound.

$$\langle dse_1, dse_2 \rangle \in SIM(\Delta ts) \iff dse_1.k = dse_2.k \land |dse_1.ts - dse_2.ts| < \Delta ts$$

Definition 6.15 Sequential Data Stream Elements

Two data stream elements dse_1 and $dse_2 \neq dse_1$ of the same data stream ($dse_1.ds = dse_2.ds$) or different data streams ($dse_1.ds \neq dse_2.ds$) are sequential with respect to a given time bound Δts if they have the same key k and the difference between their generation timestamps is greater than or equal to the time bound.

$$\langle dse_1, dse_2 \rangle \in SEQ(\Delta ts) \iff dse_1.k = dse_2.k \land |dse_1.ts - dse_2.ts| \geq \Delta ts$$

Example 6.6 Simultaneous and Sequential Data Stream Elements

Figure 6.8 illustrates the time bound ($\Delta ts = 75$) as well as simultaneous and sequential data stream elements for an abstract example. The example shows 21 elements of three data streams with the following attributes:

$dse_{A1}.ds = ds_A$	$dse_{A1}.k = k_2$	$dse_{A1}.ts = 135$
$dse_{A2}.ds = ds_A$	$dse_{A2}.k = k_1$	$dse_{A2}.ts = 251$
$dse_{A3}.ds = ds_A$	$dse_{A3}.k = k_2$	$dse_{A3}.ts = 351$
$dse_{A4}.ds = ds_A$	$dse_{A4}.k = k_2$	$dse_{A4}.ts = 454$
$dse_{A5}.ds = ds_A$	$dse_{A5}.k = k_1$	$dse_{A5}.ts = 522$
$dse_{B1}.ds = ds_B$	$dse_{B1}.k = k_1$	$dse_{B1}.ts = 140$
$dse_{B2}.ds = ds_B$	$dse_{B2}.k = k_2$	$dse_{B2}.ts = 195$
$dse_{B3}.ds = ds_B$	$dse_{B3}.k = k_1$	$dse_{B3}.ts = 250$
$dse_{B4}.ds = ds_B$	$dse_{B4}.k = k_2$	$dse_{B4}.ts = 305$
$dse_{B5}.ds = ds_B$	$dse_{B5}.k = k_1$	$dse_{B5}.ts = 360$
$dse_{B6}.ds = ds_B$	$dse_{B6}.k = k_2$	$dse_{B6}.ts = 415$
$dse_{B7}.ds = ds_B$	$dse_{B7}.k = k_1$	$dse_{B7}.ts = 470$
$dse_{B8}.ds = ds_B$	$dse_{B8}.k = k_2$	$dse_{B8}.ts = 525$
$dse_{B9}.ds = ds_B$	$dse_{B9}.k = k_1$	$dse_{B9}.ts = 580$
$dse_{C1}.ds = ds_C$	$dse_{C1}.k = k_1$	$dse_{C1}.ts = 155$
$dse_{C2}.ds = ds_C$	$dse_{C2}.k = k_2$	$dse_{C2}.ts = 256$
$dse_{C3}.ds = ds_C$	$dse_{C3}.k = k_1$	$dse_{C3}.ts = 312$
$dse_{C4}.ds = ds_C$	$dse_{C4}.k = k_1$	$dse_{C4}.ts = 363$
$dse_{C5}.ds = ds_C$	$dse_{C5}.k = k_1$	$dse_{C5}.ts = 439$
$dse_{C6}.ds = ds_C$	$dse_{C6}.k = k_2$	$dse_{C6}.ts = 510$
$dse_{C7}.ds = ds_C$	$dse_{C7}.k = k_2$	$dse_{C7}.ts = 580$

dse_{A2}, dse_{B3}, dse_{B5}, and dse_{C4} are simultaneous with dse_{C3} since they have the same key as dse_{C3} and the generation time difference to dse_{C3} is smaller than Δts. dse_{A5}, dse_{B1}, dse_{B7}, dse_{B9}, dse_{C1}, and dse_{C5} are sequential to dse_{C3} since they have the same key as dse_{C3} and the generation time difference to dse_{C3} is greater than Δts. All remaining data stream elements (dse_{A1}, dse_{A3}, dse_{A4}, dse_{B2}, dse_{B4}, dse_{B6}, dse_{B8}, dse_{C2}, dse_{C6}, and dse_{C7}) are neither simultaneous with nor sequential to dse_{C3} since they have a different key.

Note that the time bound is highly application-specific. It depends on the semantics of the raw inputs, states, events, and statistics as well as on their velocities, i.e., on the frequency with which new input data is created and emitted as well as on the expected rate with which events occur, states are calculated, and statistics are generated. For instance, in a football analysis application a generation time difference of five seconds might be already too large for a successful pass event stream element and a penalty box entry event stream element to be

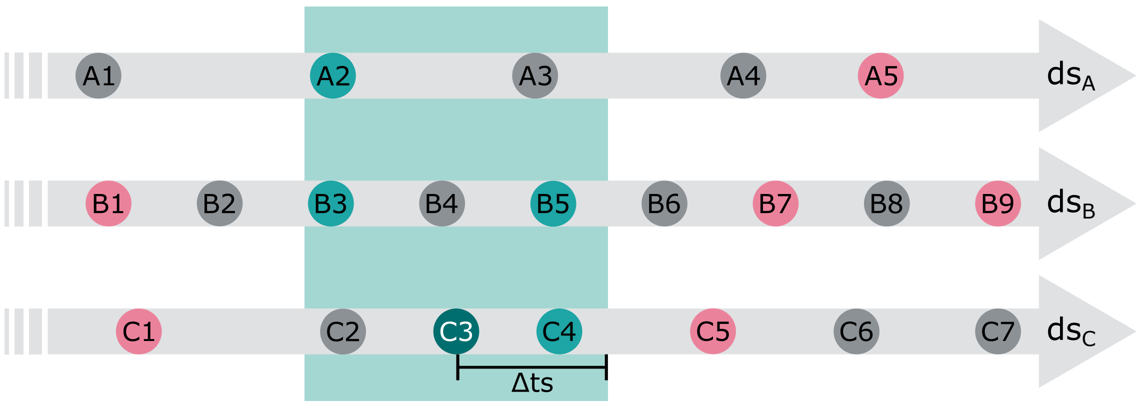


Figure 6.8 Simultaneous Data Stream Elements. Data stream elements of three data streams (ds_A, ds_B and ds_C) illustrated as dots on a temporal scale with respect to their generation timestamp. The light-mint box visualizes the time interval in which the generation timestamp of a data stream element has to lie to qualify the element for being simultaneous with dse_{C3} (colored in dark-mint). All data stream elements which are simultaneous with dse_{C3} are colored in mint, all data stream elements which are sequential to dse_{C3} are colored in red, and all data stream elements which have a different key as (and are thus neither simultaneous with nor sequential to) dse_{C3} are colored in gray.

regarded as simultaneous while in a fake news dissemination analysis [VRA18] application data stream elements representing tweets with generation time differences up to one minute might still be regarded as simultaneous. Moreover, in some applications the time bound might depend on the data streams since they exhibit crucial velocity differences. For instance, in football analysis two event stream elements (e.g., a successful pass event stream element and a penalty box entry event stream element) whose generation timestamps differ by half a second are regarded as simultaneous but two player sensor input stream elements with the same generation time difference are regarded as sequential since new player sensor input data are created and emitted with a much higher velocity than successful pass events and penalty box entry events occur.

On the data stream level we consider simultaneousness in a different way. One could argue that two data streams whose existence overlap in time are simultaneous with each other. However, in this case all data streams consumed or produced by a data stream analysis system would be inherently simultaneous as they exist concurrently to transport raw input, event, state, and statistics data. Hence, defining the simultaneousness of data streams based on the fact if their existence overlaps would not be useful for us.

In contrast, we regard the potential of two arbitrary data stream elements to be simultaneous with respect to the data streams they belong to. In the most scenarios the most data streams can contain pairs of data stream elements which are simultaneous. For instance, in a football match two players can simultaneously enter the penalty box and two players can perform a successful pass while one of these players or another player enters a penalty box. Hence, two elements of the penalty box entry event stream or an element of the penalty box entry event stream and an element of the successful pass event stream can be simultaneous according to Definition 6.14. Moreover, multiple player sensors might synchronously measure the heart rate and position of the players they are attached to and emit measurements as player sensor input stream elements which are simultaneous to each other. Similarly, team area states for multiple teams can be calculated in parallel and emitted in team area state stream elements which refer to (almost) the same time and are thus simultaneous to each other. The same is true for the fitness statistics stream. However, in some scenarios there are data streams for which it can be excluded that they contain simultaneous data stream elements. For instance, in a football match events which involve interaction with the ball typically cannot happen at (almost) the same time. Hence, elements of data streams whose elements reflect such ball interaction events cannot be simultaneous but only sequential. We denote two data streams whose elements cannot be pairwisely simultaneous but only sequential as *exclusively sequential*. Moreover, we denote a single data stream whose elements cannot be simultaneous but only sequential as *exclusively self-sequential*. For instance, way say that the successful pass event stream is exclusively self-sequential and that the successful pass event stream and the goal event stream are exclusively sequential as neither two successful pass events nor a successful pass event and a goal event can happen at the same time.

Formally, we define two data streams to be exclusively sequential and a single data stream to be exclusively self-sequential as follows:

Definition 6.16 Exclusively Sequential

Two data streams ds_1 and $ds_2 \neq ds_1$ are exclusively sequential if all data stream elements dse_1 and dse_2 of these two data streams (i.e., $dse_1.ds = ds_1$ and $dse_2.ds = ds_2$) which have the same key k are sequential with respect to Δts (according to Definition 6.15).

$$\begin{aligned}
&\langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts) \\
\iff &\forall dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = ds_1,\ dse_2.ds = ds_2 \text{ and } dse_1.k = dse_2.k : \\
&\qquad \langle dse_1, dse_2 \rangle \in SEQ(\Delta ts)
\end{aligned}$$

Definition 6.17 Exclusively Self-Sequential

A data streams ds is exclusively self-sequential if all data stream elements dse_1 and dse_2 of this data streams (i.e., $dse_1.ds = ds$ and $dse_2.ds = ds$) which have the same key k are sequential with respect to Δts (according to Definition 6.15).

$$\begin{aligned} &\langle ds, ds\rangle \in EXSEQ(\Delta ts) \\ &\iff \forall dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = ds,\ dse_2.ds = ds \text{ and } dse_1.k = dse_2.k : \\ &\qquad \langle dse_1, dse_2\rangle \in SEQ(\Delta ts) \end{aligned}$$

In the following, we will address if simultaneousness and sequentiality on the data stream element level and exclusive sequentiality on the data stream level are symmetric and transitive:

Theorem 6.13 Symmetry of Simultaneousness

If dse_1 and dse_2 are simultaneous with respect to Δts, we can follow that dse_2 and dse_1 are simultaneous with respect to Δts.

$$\forall dse_1, dse_2 \in \widehat{DSE} : \langle dse_1, dse_2\rangle \in SIM(\Delta ts) \implies \langle dse_2, dse_1\rangle \in SIM(\Delta ts)$$

Proof 6.13 Symmetry of Simultaneousness

Assume that dse_1 and dse_2 are simultaneous with respect to Δts.

$$\langle dse_1, dse_2\rangle \in SIM(\Delta ts)$$

Hence, according to Definition 6.14, dse_1 and dse_2 have the same key k and the difference between their generation timestamps is smaller than Δts.

$$dse_1.k = dse_2.k \wedge |dse_1.ts - dse_2.ts| < \Delta ts$$

According to Definition 6.14, this means we can follow that dse_2 and dse_1 are simultaneous with respect to Δts.

$$\langle dse_2, dse_1\rangle \in SIM(\Delta ts)$$

In consequence, the simultaneousness of data stream elements is symmetric. □

Theorem 6.14 Intransitivity of Simultaneousness

We cannot follow from the fact that dse_1 and dse_2 are simultaneous with respect to Δts and that dse_2 and dse_3 are simultaneous with respect to Δts that dse_1 and dse_3 are simultaneous with respect to Δts.

$$\langle dse_1, dse_2\rangle \in SIM(\Delta ts) \wedge \langle dse_2, dse_3\rangle \in SIM(\Delta ts) \;\not\Longrightarrow\; \langle dse_1, dse_3\rangle \in SIM(\Delta ts)$$

Proof 6.14 Intransitivity of Simultaneousness

We prove the intransitivity of simultaneousness by means of a counterexample for transitivity. Assume that the simultaneousness time bound in football analysis is one second (i.e., $\Delta ts = 1000$). Further, assume ds to be the data stream for shipping penalty box entry event data, i.e., the time when and the position where a player entered which penalty box. Assume that dse_1, dse_2 and dse_3 are samples for this data stream with the following generation timestamps:

$$dse_1.ts = 87728 \qquad dse_2.ts = 87913 \qquad dse_3.ts = 88835$$

dse_1 and dse_2 as well as dse_2 and dse_3 are simultaneous ($|87728 - 87913| = 185 < 1000$ and $|87913 - 88835| = 922 < 1000$).

$$\langle dse_1, dse_2\rangle \in SIM(1000) \wedge \langle dse_2, dse_3\rangle \in SIM(1000)$$

If simultaneousness was transitive, we could follow that dse_1 and dse_3 are simultaneous. However, dse_1 and dse_3 are sequential ($|87728 - 88835| = 1107 > 1000$) and thus not simultaneous.

$$\langle dse_1, dse_3\rangle \notin SIM(1000)$$

Hence, we cannot follow from the fact that dse_1 and dse_2 as well as dse_2 and dse_3 are simultaneous with respect to Δts that dse_1 and dse_3 are simultaneous with respect to Δts.

$$\langle dse_1, dse_2\rangle \in SIM(1000) \wedge \langle dse_2, dse_3\rangle \in SIM(1000) \;\not\Longrightarrow\; \langle dse_1, dse_3\rangle \in SIM(1000)$$

In consequence, the simultaneousness of data stream elements is intransitive. □

Theorem 6.15 Symmetry of Sequentiality

If dse_1 and dse_2 are sequential with respect to Δts, we can follow that dse_2 and dse_1 are sequential with respect to Δts.

$$\forall dse_1, dse_2 \in \widehat{DSE} : \langle dse_1, dse_2 \rangle \in SEQ(\Delta ts) \implies \langle dse_2, dse_1 \rangle \in SEQ(\Delta ts)$$

Proof 6.15 Symmetry of Sequentiality

Assume that dse_1 and dse_2 are sequential with respect to Δts.

$$\langle dse_1, dse_2 \rangle \in SEQ(\Delta ts)$$

Hence, according to Definition 6.15, dse_1 and dse_2 have the same key k and the difference between their generation timestamps is greater than or equal to Δts.

$$dse_1.k = dse_2.k \land |dse_1.ts - dse_2.ts| \geq \Delta ts$$

According to Definition 6.15, this means we can follow that dse_2 and dse_1 are sequential with respect to Δts.

$$\langle dse_2, dse_1 \rangle \in SEQ(\Delta ts)$$

In consequence, the sequentiality of data stream elements is symmetric. □

Theorem 6.16 Intransitivity of Sequentiality

We cannot follow from the fact that dse_1 and dse_2 are sequential with respect to Δts and that dse_2 and dse_3 are sequential with respect to Δts that dse_1 and dse_3 are sequential with respect to Δts.

$$\langle dse_1, dse_2 \rangle \in SEQ(\Delta ts) \land \langle dse_2, dse_3 \rangle \in SEQ(\Delta ts) \centernot\implies \langle dse_1, dse_3 \rangle \in SEQ(\Delta ts)$$

Proof 6.16 Intransitivity of Sequentiality

We prove the intransitivity of sequentiality by means of a counterexample for transitivity. Assume that the simultaneousness time bound in football analysis is one second (i.e., $\Delta ts = 1000$). Further, assume ds to be the data stream for shipping penalty box entry event data, i.e., the time when and the position where a player entered which penalty box. Assume that dse_1, dse_2 and dse_3 are samples for this data stream with the following generation timestamps:

$$dse_1.ts = 80001 \qquad dse_2.ts = 81554 \qquad dse_3.ts = 80234$$

dse_1 and dse_2 as well as dse_2 and dse_3 are sequential ($|80001 - 81554| = 1553 > 1000$ and $|81554 - 80234| = 1320 > 1000$).

$$\langle dse_1, dse_2 \rangle \in SEQ(1000) \land \langle dse_2, dse_3 \rangle \in SEQ(1000)$$

If simultaneousness was transitive, we could follow that dse_1 and dse_3 are sequential. However, dse_1 and dse_3 are simultaneous ($|80001 - 80234| = 233 < 1000$) and thus not sequential.

$$\langle dse_1, dse_3 \rangle \notin SEQ(1000)$$

Hence, we cannot follow from the fact that dse_1 and dse_2 as well as dse_2 and dse_3 are sequential with respect to Δts that dse_1 and dse_3 are sequential with respect to Δts.

$$\langle dse_1, dse_2 \rangle \in SEQ(1000) \land \langle dse_2, dse_3 \rangle \in SEQ(1000) \;\not\Longrightarrow\; \langle dse_1, dse_3 \rangle \in SEQ(1000)$$

In consequence, the sequentiality of data stream elements is intransitive. □

Theorem 6.17 Symmetry of Exclusive Sequentiality

If ds_1 and ds_2 are exclusively sequential with respect to Δts, we can follow that ds_2 and ds_1 are exclusively sequential with respect to Δts.

$$\forall ds_1, ds_2 \in \widehat{DS} : \langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts) \implies \langle ds_2, ds_1 \rangle \in EXSEQ(\Delta ts)$$

Proof 6.17 Symmetry of Exclusive Sequentiality

Assume that ds_1 and ds_2 are exclusively sequential with respect to Δts.

$$\langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts)$$

Hence, according to Definition 6.16, all data stream elements dse_1 and dse_2 of these two data streams (i.e., $dse_1.ds = ds_1$ and $dse_2.ds = ds_2$) which have the same key k are sequential with respect to Δts.

$$\forall dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = ds_1,\ dse_2.ds = ds_2 \text{ and } dse_1.k = dse_2.k : \\ \langle dse_1, dse_2 \rangle \in SEQ(\Delta ts)$$

Since we have shown in Proof 6.15 that the sequentiality of data stream elements is symmetric (i.e., $\langle dse_1, dse_2 \rangle \in SEQ(\Delta ts) \implies \langle dse_2, dse_1 \rangle \in SEQ(\Delta ts)$), we can follow according to Definition 6.16 that ds_2 and ds_1 are exclusively sequential with respect to Δts.

$$\langle ds_2, ds_1 \rangle \in EXSEQ(\Delta ts)$$

Hence, the exclusive sequentiality of data streams is symmetric. □

Theorem 6.18 Intransitivity of Exclusive Sequentiality

We cannot follow from the fact that ds_1 and ds_2 are exclusively sequential with respect to Δts and that ds_2 and ds_3 are exclusively sequential with respect to Δts with $ds_1 \neq ds_2 \neq ds_3$ that ds_1 and ds_3 are exclusively sequential with respect to Δts.

$$\langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts) \wedge \langle ds_2, ds_3 \rangle \in EXSEQ(\Delta ts) \not\Longrightarrow \langle ds_1, ds_3 \rangle \in EXSEQ(\Delta ts)$$

Proof 6.18 Intransitivity of Exclusive Sequentiality

Assume that ds_1 and ds_2 are exclusively sequential with respect to Δts and that ds_2 and ds_3 are exclusively sequential with respect to Δts.

$$\langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts) \wedge \langle ds_2, ds_3 \rangle \in EXSEQ(\Delta ts)$$

Hence, according to Definition 6.16, all data stream elements dse_1 and dse_2 of ds_1 and ds_2 (i.e., $dse_1.ds = ds_1$ and $dse_2.ds = ds_2$) which have the same key are sequential with respect to Δts.

$$\forall dse_1, dse_2 \in \widehat{DSE} \text{ with } dse_1.ds = ds_1,\ dse_2.ds = ds_2 \text{ and } dse_1.k = dse_2.k : \\ \langle dse_1, dse_2 \rangle \in SEQ(\Delta ts)$$

Moreover, according to Definition 6.16, all data stream elements dse_3 and dse_4 of ds_2 and ds_3 (i.e., $dse_3.ds = ds_2$ and $dse_4.ds = ds_3$) which have the same key are sequential with respect to Δts.

$$\forall dse_3, dse_4 \in \widehat{DSE} \text{ with } dse_3.ds = ds_2,\ dse_4.ds = ds_3 \text{ and } dse_3.k = dse_4.k :$$
$$\langle dse_3, dse_4 \rangle \in SEQ(\Delta ts)$$

However, from that we cannot follow that all data stream elements dse_5 and dse_6 of ds_1 and ds_3 (i.e., $dse_5.ds = ds_1$ and $dse_6.ds = ds_3$) which have the same key are sequential with respect to Δts. Instead they can also be simultaneous with respect to Δts.

$$\forall dse_5, dse_5 \in \widehat{DSE} \text{ with } dse_5.ds = ds_1,\ dse_6.ds = ds_3 \text{ and } dse_5.k = dse_6.k :$$
$$\langle dse_5, dse_6 \rangle \in SEQ(\Delta ts) \vee \langle dse_5, dse_6 \rangle \in SIM(\Delta ts)$$

Consequently, we cannot follow using Definition 6.16 that ds_1 and ds_3 are exclusively sequential with respect to Δts. Therefore, we cannot follow from the fact that ds_1 and ds_2 are exclusively sequential with respect to Δts and that ds_2 and ds_3 are exclusively sequential with respect to Δts with $ds_1 \neq ds_2 \neq ds_3$ that ds_1 and ds_3 are exclusively sequential with respect to Δts.

$$\langle ds_1, ds_2 \rangle \in EXSEQ(\Delta ts) \wedge \langle ds_2, ds_3 \rangle \in EXSEQ(\Delta ts) \not\Longrightarrow \langle ds_1, ds_3 \rangle \in EXSEQ(\Delta ts)$$

Hence, the exclusive sequentiality of data streams is intransitive. □

Example 6.7 Intransitivity of Exclusive Sequentiality in Football Analysis

The intransitivity of the exclusive sequentiality of data streams can be illustrated by means of regarding the semantics of the goal event stream, the successful pass event stream, and the freekick event stream of the football analysis scenario. Goal event stream elements are always sequential to successful pass event stream elements as successful passes and goals cannot happen at (approximately) the same time. The same is true for successful pass event stream elements and freekick event stream elements since we do not regard a freekick that is received by a player of the same team as a successful pass but only as a freekick. However, a goal event stream element and a freekick event stream element can be simultaneous since a direct freekick can lead to a goal. In consequence, the goal event stream and the successful pass event stream are exclusively sequential, the successful pass event stream and the freekick event stream are exclusively sequential, but the goal event stream and the freekick event stream are not exclusively sequential.

As a last point we want to highlight that Theorem 6.14, Theorem 6.16, and Theorem 6.18 state only that the simultaneousness of data stream elements, the sequentiality of data stream elements, and the exclusive sequentiality of data streams are intransitive but not that they are antitransitive. That means for instance that it is possible that there are three data stream elements dse_1, dse_2, and dse_3 which are all pairwisely simultaneous for a given time bound Δts (i.e., $\langle dse_1, dse_2\rangle \in SIM(\Delta ts)$, $\langle dse_2, dse_3\rangle \in SIM(\Delta ts)$, and $\langle dse_1, dse_3\rangle \in SIM(\Delta ts)$).

7 Spatial Functions and Relations

Besides temporal information also spatial information is of particular significance for many analyses, especially when regarding team behavior as we do in our real-time football analysis application (see Chapter 9). However, in contrast to the consistent temporal information[1] a data stream element comprises the quantity of spatial information is inherently inconsistent.

To highlight this difference, recall that we have been able to define simultaneousness and sequentiality of data stream elements and exclusive sequentiality of data streams in a generic way based on generation timestamps (see Section 6.3). This has been possible since every data stream element is defined to contain exactly one generation timestamp (see Definition 4.2) and hence there is a consistent form of temporal information present throughout all data streams handled by a data stream analysis system. However, there is no such restriction for the spatial information. The generic data stream element definition (see Definition 4.2) merely defines that all positions are contained in the positions tuple (pos), but neither this definition nor the additional well-formation constraints presented in Section 4.5 enforce a data stream element to contain positions at all. This is by design since there are raw inputs, events, states, and statistics which cannot be assigned to a location. For instance, a pass statistic containing the average pass success rate of a team does not correspond to a specific location and thus elements of the pass statistics stream contain no position. In these cases the positions tuple is set to null (i.e., $pos = \lambda$). We denote a data stream element

[1] Remember that although the payload of some data stream elements can also contain temporal information (e.g., Example 4.5) and although only raw input stream elements have an ingestion timestamp, each data stream element contains temporal information with a consistent encoding and quantity as every data stream element has a single globally unambiguous generation timestamp, a single globally unambiguous sequence number, and a single locally valid processor-specific processing timestamp (see Section 6.1.4).

that contains positions as *spatial* and a data stream element that does not contain positions as *non-spatial*. Formally, spatial and non-spatial data stream elements are defined as follows:

Definition 7.1 Spatial Data Stream Element

A data stream element *dse* is spatial if it contains at least one position.

$$\left|dse.pos\right| \geq 1$$

Otherwise, *dse* is non-spatial.

Moreover, even if we regard only spatial data stream elements there is no consistent number of positions. Instead, every data stream element is defined to contain a positions tuple (*pos*) that consists of an arbitrary number of three dimensional positions (see Section 4.2). This is due to the fact that different raw inputs, events, states, and statistics are assigned to a different number of locations. For instance, a penalty box entry event is only assigned to a single location (the position where the player entered the penalty box) while a successful pass event is assigned to two locations (the start and the end of the pass). Thus, elements of the penalty box entry event stream and elements of the successful pass event stream contain one and two positions, respectively. Moreover, albeit uncommon, the number of positions might even differ between elements of the same data stream. For instance, offside line state stream elements contain an inconsistent number of positions as the number of players in offside position varies during the match. It is even possible that a data stream consists of spatial and non-spatial data stream elements.

Taken together, a data stream element can have any arbitrary number (incl. zero) of positions. This lack of a consistent form of spatial information complicates defining generic spatial functions and relations on the data stream element level as they would have to be defined for two data stream elements containing an arbitrary and potentially even distinct number of positions. Furthermore, the semantics of the positions are data stream specific. For instance, in our real-time football analysis application (see Chapter 9) the positions of a data stream element representing a successful pass event are the start and end location of the pass while the positions of an offside line state stream element are the locations of the virtual offside line and the players in offside position. Consequently, it can be impossible to define a spatial function or relation on the data stream element level in a way that it is generic, i.e., applicable to arbitrary (spatial) data stream elements, but still has meaningful and unambiguous semantics.

Example 7.1 Problems with Proximity Relation for Data Stream Elements

The attempt to define the proximity of two data stream elements, i.e., the fact that they are spatially close to each other, in a generic way similar to the simultaneousness of two data stream elements (see Definition 6.14) illustrates these problems.

As a first attempt, assume two data stream elements to be defined as close if all positions of both data stream elements are close to each other (with respect to a distance function d and a distance threshold Δd, see Section 7.1).

$$\forall\ \rho_1 \in dse_1.pos, \rho_2 \in dse_2.pos\ :\ d\left(\rho_1, \rho_2\right) < \Delta d$$

This definition would enable checking the proximity of two penalty box entry events. However, on the basis of this definition two successful pass events cannot be close if the length of one of these passes is greater than the distance threshold. For the same reason, a penalty box entry event cannot be close to a successful pass event if the pass is too long.

As a second attempt, assume two data stream elements to be defined as close if there is at least one position in the first data stream element that is close to at least one position in the second data stream element.

$$\exists\ \rho_1 \in dse_1.pos\ :\ \left(\exists\ \rho_2 \in dse_2.pos\ :\ d\left(\rho_1, \rho_2\right) < \Delta d\right)$$

This definition would enable checking the proximity of penalty box entry events and successful pass events. However, as this definition does not specify which positions of the data stream elements have to be close to each other the semantics of two close data stream elements are ill-defined. For instance, if a data stream element representing a penalty box entry event and a data stream element representing a successful pass event are close according to this definition, it is ambiguous if the penalty box entry event is close to the start or the end of the pass.

Assume that the football analysis application requires a penalty box entry event to be only regarded as close to a successful pass event if it is located close to the start location of the pass. For this purpose, as a third attempt, assume two data stream elements to be defined as close if the first positions of both data stream elements are close to each other.

$$d\left(dse_1.pos.1, dse_2.pos.1\right) < \Delta d$$

While this definition would perfectly match the application demands for checking the proximity of penalty box entry events and successful pass events, the second definition would fit much better to check the proximity of a penalty box entry event and an offside line state if the application demands a penalty box entry event to be regarded as close to an offside line state if the position where the player entered the penalty box is close to the location of any player in offside position.

As Example 7.1 has illustrated, it can be impossible to define a generic spatial function or relation without introducing ambiguities or violating application demands even when only a single application is regarded. When considering position semantics and demands of not only a single but multiple applications the problems become even more apparent. The only way to solve all problems would be to define the functions and relations differently for different data stream element combinations and hence not generically. Therefore, we relinquish defining spatial functions and relations on a data stream element level in a generic way. In consequence, we further relinquish to define generic spatial functions and relations on a data stream level as they would naturally be based on generic spatial functions and/or relations for data stream elements.

Nevertheless, it is possible to obtain meaningful insights by means of applying generic spatial functions and relations to positions of data stream elements. In fact, this is massively done in workers of data stream analysis systems. For instance, in our real-time football analysis application detecting a ball possession change event requires finding the closest player to the ball and thus comparing the distances between the positions of all players and the ball (see Section 9.2.1.6). Moreover, detecting a penalty box entry event necessitates checking if a player enters a penalty box and thus performing point containment checks using the positions of the players and the penalty box (see Section 9.2.1.4). Similarly, distinguishing between an interception event and a clearance event requires checking in which area the player who shot (first position) and the player who received (second position) the ball was located (see Section 9.2.1.10).

However, note that these generic spatial functions and relations (i.e., distance, point containment, etc.) are not defined on a data stream or a data stream element level and thus cannot be applied to data streams or data stream elements directly. Instead, they are defined for arbitrary positions. Thus, if a spatial function or relation should be applied to positions of data stream elements these positions have to be explicitly selected beforehand using application knowledge about the semantics of the available positions.

In the following sections, we will present some basic spatial functions and relations for arbitrary positions. These generic functions serve as a foundation for performing real-time analyses of spatial data stream elements. In our real-time football analysis application we use them to detect, calculate, and generate collaborative team events, states, and statistics, respectively (see Chapter 9).

7.1 Distance

The most basic spatial function is the *distance* function, also denoted as *metric*, that can be used to determine the distance between two elements of a metric space. In [Shi07] a metric is formally defined as follows:

Definition 7.2 Metric

"A metric on a set X is a function $d : X \times X \rightarrow \mathbb{R}$ satisfying the following properties:

1. $d(x, y) \geq 0$ for all $x, y \in X$.
2. $d(x, y) = d(y, x)$ for all $x, y \in X$ (i.e., d is a symmetric function).
3. $d(x, y) \leq d(x, z) + d(z, y)$ for all elements x, y and z of X (triangle inequality).
4. $d(x, x) = 0$ for all $x \in X$.
5. If $d(x, y) = 0$ then $x = y$.

The pair (X, d) is called a metric space." [Shi07, p. 158]

In theory, many different metrics could be used to determine the distance between two three dimensional positions. There is a plethora of metrics developed for and used in different fields [DD09]. However, our work focuses on the real-time analysis of collaborative team scenarios. In this context, we are most interested in determining and comparing the natural distance between objects. This natural distance is covered the best by the *Euclidean metric* [Shi07, p. 158]. Therefore, in the remainder of this thesis, we use the Euclidean metric to determine the distance between two positions and thus formally define the distance between two positions as follows:

Definition 7.3 Distance between two Positions

The distance $d(\rho_1, \rho_2)$ between two positions $\rho_1 = \langle x, y, z \rangle \in \mathbb{R}^3$ and $\rho_2 = \langle x, y, z \rangle \in \mathbb{R}^3$ is the Euclidean distance between the three dimensional points in space.

$$d(\rho_1, \rho_2) = \sqrt{(\rho_1.x - \rho_2.x)^2 + (\rho_1.y - \rho_2.y)^2 + (\rho_1.z - \rho_2.z)^2}$$

On the basis of this distance function workers of data stream analysis systems can realize a multitude of simple and complex spatial functions and relations. Among others, the distance function enables calculating the length and velocity of an event (e.g., of a successful pass event), checking if two positions are close to each other (e.g., to check if there is a duel), and retrieving the nearest neighbor [PS85, p. 180] of a position (e.g., to identify the player in possession of the ball).

Example 7.2 Close Positions

For instance, a worker of a data stream analysis system can check if two positions ρ_1 and ρ_2 are close to each other by means of comparing the distance between these positions to a distance threshold (Δd).

$$d(\rho_1, \rho_2) < \Delta d$$

This comparison implicitly defines a symmetric but intransitive proximity relation containing all pairs of positions that are close to each other.

$$\langle \rho_1, \rho_2 \rangle \in CLOSE(\Delta d) \iff d(\rho_1, \rho_2) < \Delta d$$

Note that the distance threshold is not a global parameter but depends on the usage of the proximity check. In our real-time football analysis application the distance threshold for checking if a player is close enough to the ball to be assumed to have received the ball is 1.5 meters while during a duel both players are allowed to be up to 3 meters away from the ball (see Section 9.2.1.6).

7.2 Planar Projection and Areas

As humans perceive the world to be three dimensional, we have defined data stream elements to contain a tuple of three dimensional positions (see Definition 4.2). The z-coordinate can be very beneficial for some analysis tasks. For instance, in our real-time football analysis application the z-coordinate of the ball is used to distinguish reliably between goals and shots off target (see Section 9.2.1.10).

However, some position sensors and tracking systems support only two dimensional positions. In order to still have a consistent coordinate system the actually two dimensional positions of such raw input data are wrapped in three dimensional positions with the z-coordinate set to zero. We denote such positions as *planar* positions, formally defined as follows:

Definition 7.4 Planar Position

A position ρ is planar if its z-coordinate is zero.

$$\rho \in \left\{ \left\langle x, y, 0 \right\rangle \mid x, y \in \mathbb{R} \right\}$$

As there are such inherently planar positions, it might be necessary to disregard the z-coordinate of other positions for a subset of the analysis tasks in order to have a consistent level of spatial information (e.g., for checking the proximity of positions). Moreover, in the context of collaborative team analysis it can be even beneficial to discard the z-coordinate and thus to consider objects as points in a planar world as some objects (e.g., players in a football match) move mostly along the x-axis and the y-axis. In fact, doing so can simplify some analyses, such as examining the placement of football players on the field, remarkably. Therefore, we introduce a function that projects three dimensional positions into a planar world. This *planar projection* is formally defined as follows:

Definition 7.5 Planar Projection

The planar projection $planar(\rho)$ of a three dimensional position $\rho = \left\langle x, y, z \right\rangle \in \mathbb{R}^3$ onto the xy-plane sets the z-coordinate to zero.

$$planar(\rho) = \left\langle \rho.x, \rho.y, 0 \right\rangle$$

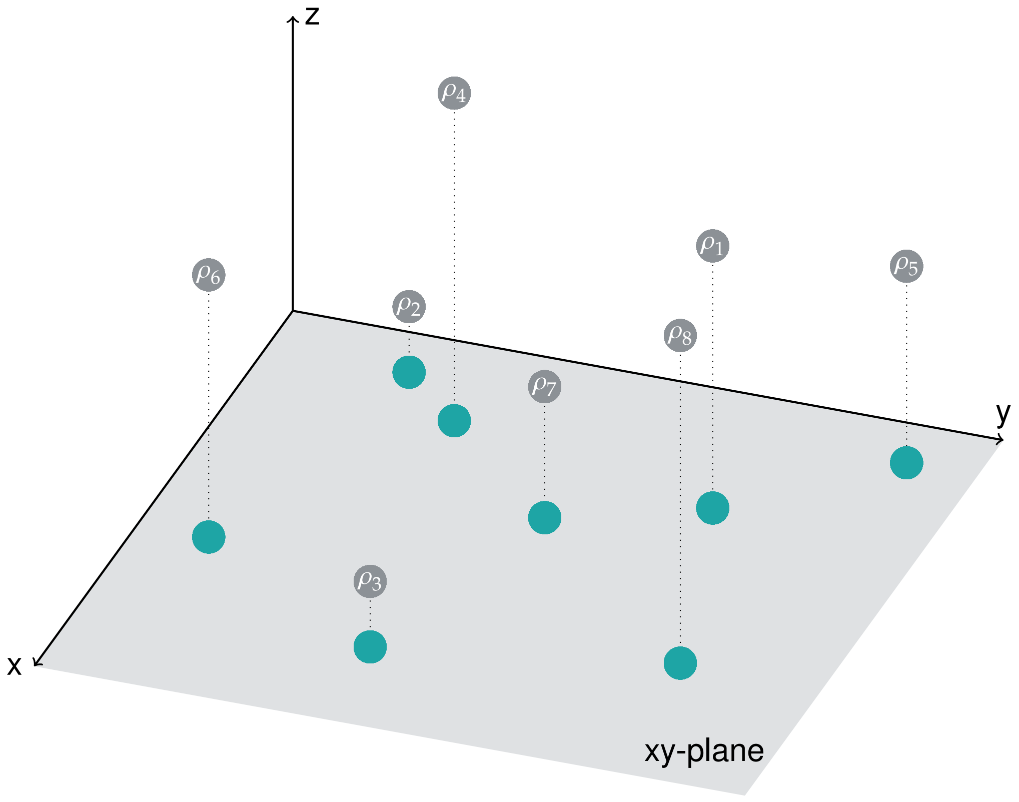


Figure 7.1 **Planar Projection.** Three dimensional positions $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8\}$ colored in gray and their planar projections onto the xy-plane $\{planar(\rho) \mid \rho \in P\}$ colored in mint.

Example 7.3 **Planar Projection**

Figure 7.1 illustrates a set of three dimensional positions P as well as their planar projections $\{planar(\rho) \mid \rho \in P\}$.

$$P = \Big\{\overbrace{\langle 3,7,4\rangle}^{\rho_1}, \overbrace{\langle 1,2,1\rangle}^{\rho_2}, \overbrace{\langle 8,4,1\rangle}^{\rho_3}, \overbrace{\langle 2,3,5\rangle}^{\rho_4}, \overbrace{\langle 1,9,3\rangle}^{\rho_5}, \overbrace{\langle 6,1,4\rangle}^{\rho_6}, \overbrace{\langle 4,5,2\rangle}^{\rho_7}, \overbrace{\langle 7,8,5\rangle}^{\rho_8}\Big\}$$

$$planar(\rho_1) = \langle 3,7,0\rangle \qquad planar(\rho_2) = \langle 1,2,0\rangle \qquad planar(\rho_3) = \langle 8,4,0\rangle$$
$$planar(\rho_4) = \langle 2,3,0\rangle \qquad planar(\rho_5) = \langle 1,9,0\rangle \qquad planar(\rho_6) = \langle 6,1,0\rangle$$
$$planar(\rho_7) = \langle 4,5,0\rangle \qquad planar(\rho_8) = \langle 7,8,0\rangle$$

Note that the distance function introduced in Definition 7.3 as well as all other spatial functions and relations implemented for three dimensional positions can be also applied to their planar projections. This is true since we have defined the planar projection of a three dimensional position $\langle x, y, z\rangle$ not to be an actual two dimensional position $\langle x, y\rangle$ but to be a three dimensional position on the xy-plane $\langle x, y, 0\rangle$.

Regarding the planar projections of three dimensional positions, it is interesting to consider if the projection of a certain position is contained in a given polygon on the xy-plane. This is very helpful for various analysis tasks as it can be used to detect or further classify events. For instance, in our real-time football analysis application, containment checks are used to detect if a player entered a specific region such as a penalty box (see Section 9.2.1.4) or to distinguish between interception and clearance events by means of checking in which areas the player who shot and the player who received the ball were located (see Section 9.2.1.10). As suggested in [BCK+08, p. 3] we define a polygon to be specified by a tuple containing the positions of its vertices in clockwise order. Based on that, we formally define a point containment function for checking if the planar projection of a three dimensional position is contained in an arbitrarily shaped polygon on the xy-plane as follows:

Definition 7.6 Point Containment

The function *contained*(ρ, pos) states if the planar projection *planar*(ρ) of a position ρ is contained in the polygon on the xy-plane specified by a tuple *pos* of planar positions (i.e., $\rho_{poly}.z = 0$ for all $\rho_{poly} \in pos$).

$$contained(\rho, pos) = \begin{cases} \top & \text{if } \; planar(\rho) \text{ is contained in the polygon specified by} \\ & \qquad \text{the planar positions in } pos \\ \bot & \text{otherwise} \end{cases}$$

Checking if the planar projection of a certain position is contained in an arbitrary polygon on the xy-plane is a solvable but non-trivial problem. However, solving this problem gets much easier when restricting the polygon space. A special subset of polygons are *axis-aligned rectangles*, i.e., rectangles whose edges are parallel to the coordinate system axes. As we are only interested in polygons on the xy-plane we define an axis-aligned rectangle to be a rectangle on the xy-plane whose edges are parallel to the x-axis and to the y-axis. Axis-aligned rectangles are of particular interest for workers of data stream analysis systems as it is trivial and fast to check with the following algorithm if the planar projection *planar*(ρ) of a given three dimensional position ρ is contained in an axis-aligned rectangle whose corners are specified by a tuple *pos* of planar positions:

Algorithm 7.1 Containment Check for an Axis-aligned Rectangle

Input: A three dimensional position ρ
A tuple of planar positions *pos* that specifies the corners of an axis-aligned rectangle

Output: $\top$ if ρ is contained in the rectangle. Otherwise $\perp$.

```
1: $x_{min} \leftarrow \min\left\{\rho_{rect}.x \mid \rho_{rect} \in pos\right\}$
2: $y_{min} \leftarrow \min\left\{\rho_{rect}.y \mid \rho_{rect} \in pos\right\}$
3: $x_{max} \leftarrow \max\left\{\rho_{rect}.x \mid \rho_{rect} \in pos\right\}$
4: $y_{max} \leftarrow \max\left\{\rho_{rect}.y \mid \rho_{rect} \in pos\right\}$
5: if $planar(\rho).x \in [x_{min}, x_{max}] \wedge planar(\rho).y \in [y_{min}, y_{max}]$ then
6:     return $\top$
7: else
8:     return $\perp$
9: end if
```

In many scenarios the most areas can be represented or at least approximated by combining multiple axis-aligned rectangles. For instance, all areas of a football field (except for the circles) can be specified using axis-aligned rectangles. However, this is not the case for all areas. For instance, specifying the fair and foul territory of a baseball field necessitates less restricted polygons. In these cases other algorithms given in literature have to be used to perform point containment checks. [Gha08, pp. 201–204] describes algorithms for arbitrary convex and concave polygons.

Example 7.4 Point Containment

Figure 7.2 and Figure 7.3 illustrate the point containment problem for a set of three dimensional positions P with an axis-aligned rectangle and a concave polygon, respectively. The rectangle is specified by pos_{rect} and the concave polygon is specified by pos_{poly}.

$$P = \left\{\overbrace{\langle 3,7,4\rangle}^{\rho_1}, \overbrace{\langle 1,2,1\rangle}^{\rho_2}, \overbrace{\langle 8,4,1\rangle}^{\rho_3}, \overbrace{\langle 2,3,5\rangle}^{\rho_4}, \overbrace{\langle 1,9,3\rangle}^{\rho_5}, \overbrace{\langle 6,1,4\rangle}^{\rho_6}, \overbrace{\langle 4,5,2\rangle}^{\rho_7}, \overbrace{\langle 7,8,5\rangle}^{\rho_8}\right\}$$

$$pos_{rect} = \left\langle \langle 2,1,0\rangle, \langle 2,9,0\rangle, \langle 6,9,0\rangle, \langle 6,1,0\rangle \right\rangle$$

$$pos_{poly} = \left\langle \langle 3,1,0\rangle, \langle 1,3,0\rangle, \langle 1,10,0\rangle, \langle 5,7,0\rangle, \langle 2,4,0\rangle, \langle 5,2,0\rangle \right\rangle$$

The planar projections of ρ_1, ρ_4, ρ_6 and ρ_7 are contained in the rectangle and the planar projections of ρ_1, ρ_4 and ρ_5 are contained in the concave polygon.

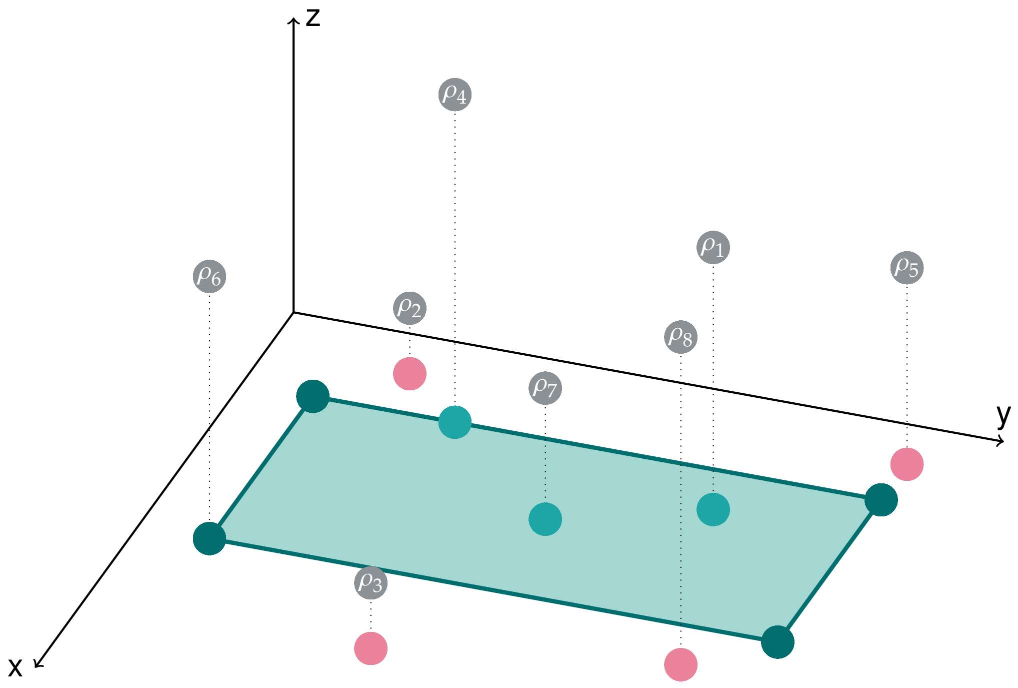


Figure 7.2 Point Containment with Axis-aligned Rectangle. The light-mint area with dark-mint edges visualizes an axis-aligned rectangle. All three dimensional positions $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8\}$ are colored in gray. The planar positions pos_{rect} that specify the rectangle (i.e., the polygon) are colored in dark-mint. The planar projections of those positions that are contained in the rectangle, i.e., $\{planar(\rho) \mid \rho \in P \wedge contained(\rho, pos_{rect}) = \top\}$, are colored in mint. The planar projections of those not contained in the rectangle, i.e., $\{planar(\rho) \mid \rho \in P \wedge contained(\rho, pos_{rect}) = \bot\}$, are colored in red. The planar projection $planar(\rho_6)$ of ρ_6 is colored in dark-mint as it is also one of the positions that specifies the rectangle.

$$contained(\rho, pos_{rect}) = \begin{cases} \top & \text{if} \quad \rho \in \{\rho_1, \rho_4, \rho_6, \rho_7\} \\ \bot & \text{if} \quad \rho \in \{\rho_2, \rho_3, \rho_5, \rho_8\} \end{cases}$$

$$contained(\rho, pos_{poly}) = \begin{cases} \top & \text{if} \quad \rho \in \{\rho_1, \rho_4, \rho_5\} \\ \bot & \text{if} \quad \rho \in \{\rho_2, \rho_3, \rho_6, \rho_7, \rho_8\} \end{cases}$$

In addition to determining if the planar projection of a certain position is contained in a given polygon on the xy-plane, it is also interesting to consider different types of areas that are spanned by a set of planarly projected positions on the xy-plane. This is for instance useful in football analysis to assess how dense a group of players is or how large an area a group of players covers is, two questions often posed by coaches, match analysts, and sports scientists.

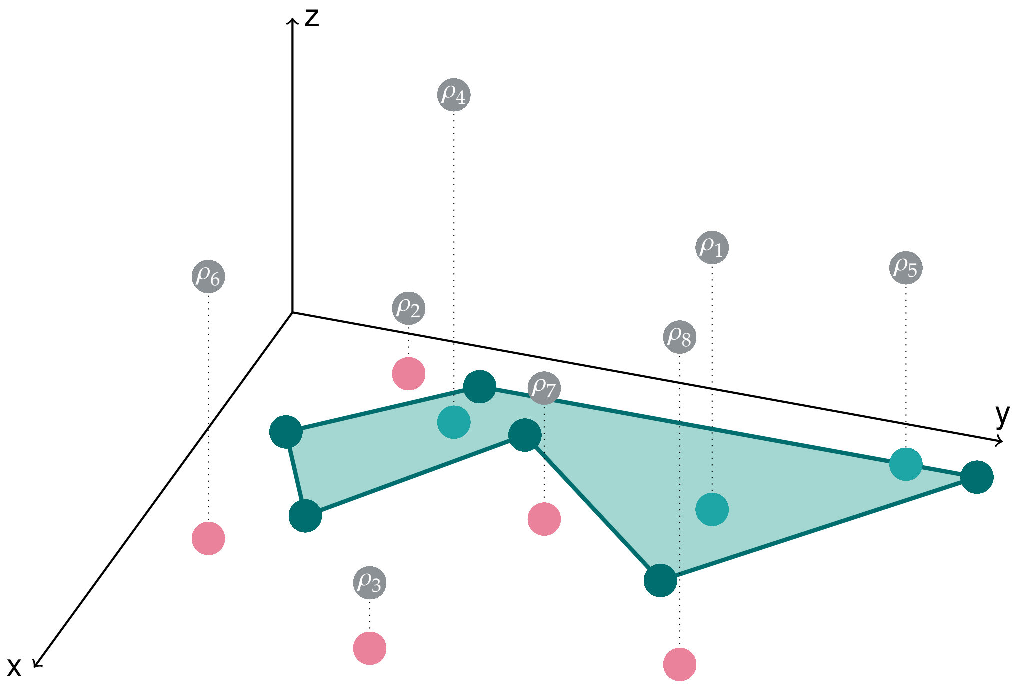


Figure 7.3 Point Containment with Concave Polygon. The light-mint area with dark-mint edges visualizes a concave polygon. All three dimensional positions $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8\}$ are colored in gray. The planar positions pos_{poly} that specify the polygon are colored in dark-mint. The planar projections of those positions that are contained in the polygon, i.e., $\left\{ planar(\rho) \;\middle|\; \rho \in P \wedge contained(\rho, pos_{poly}) = \top \right\}$, are colored in mint. The planar projections of those not contained in the polygon, i.e., $\left\{ planar(\rho) \;\middle|\; \rho \in P \wedge contained(\rho, pos_{poly}) = \bot \right\}$, are colored in red.

The simplest area type is a *minimum bounding rectangle*, i.e., the smallest possible axis-aligned rectangle that contains the planar projections of all positions. We formally define the minimum bounding rectangle for a finite set of three dimensional positions as follows:

Definition 7.7 Minimum Bounding Rectangle

The minimum bounding rectangle $mbr(P)$ of a finite set of three dimensional positions P is a tuple of four planar positions which are the corners of an axis-aligned rectangle on the xy-plane which contains the planar projections of all positions in P.

$$mbr(P) = \Big\langle \langle x_{min}, y_{min}, 0 \rangle, \langle x_{min}, y_{max}, 0 \rangle, \langle x_{max}, y_{max}, 0 \rangle, \langle x_{max}, y_{min}, 0 \rangle \Big\rangle$$

$$\text{where} \quad x_{min} = \min \{ x \mid \langle x, y, z \rangle \in P \}, \; x_{max} = \max \{ x \mid \langle x, y, z \rangle \in P \}$$

$$y_{min} = \min \{ y \mid \langle x, y, z \rangle \in P \}, \; y_{max} = \max \{ y \mid \langle x, y, z \rangle \in P \}$$

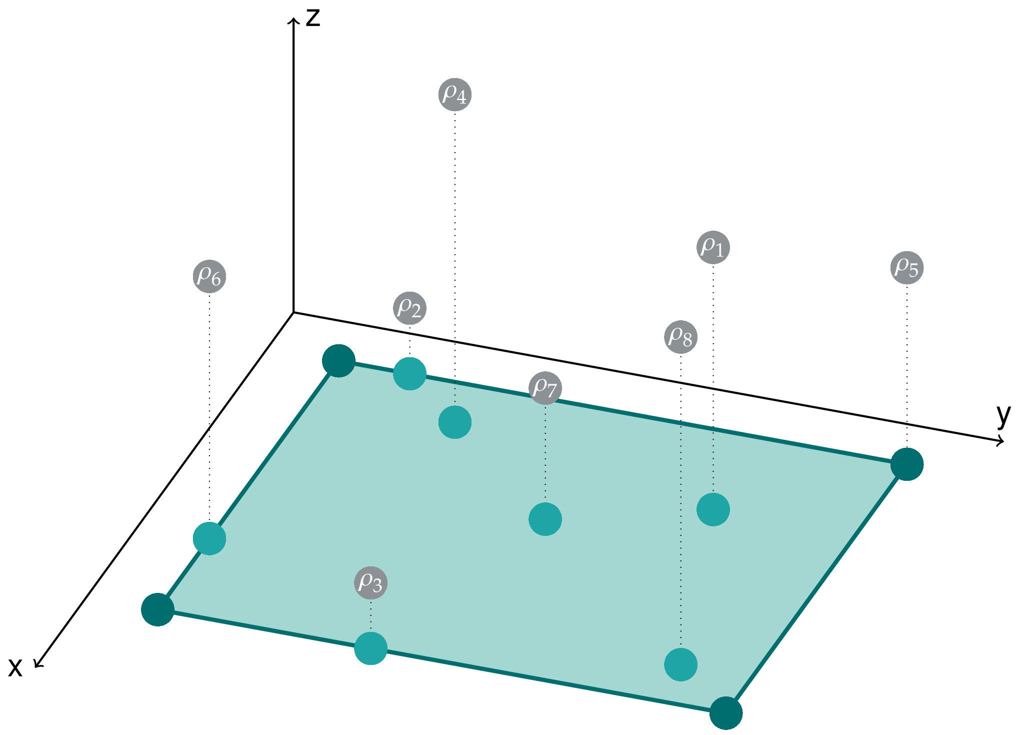


Figure 7.4 **Minimum Bounding Rectangle.** The light-mint area with dark-mint edges visualizes the minimum bounding rectangle $mbr(P)$ of a set of positions $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8\}$. The planar positions $mbr(P)$ that specify the minimum bounding rectangle (i.e., the polygon) are colored in dark-mint. The three dimensional positions P are colored in gray and their planar projections $\{planar(\rho) \mid \rho \in P\}$ are colored in mint. Only the planar projection $planar(\rho_5)$ of ρ_5 is colored in dark-mint as it is also one of the positions that specifies the minimum bounding rectangle.

Example 7.5 Minimum Bounding Rectangle

Figure 7.4 illustrates the minimum bounding rectangle $mbr(P)$ for a set of three dimensional positions P.

$$P = \left\{ \overbrace{\langle 3,7,4\rangle}^{\rho_1}, \overbrace{\langle 1,2,1\rangle}^{\rho_2}, \overbrace{\langle 8,4,1\rangle}^{\rho_3}, \overbrace{\langle 2,3,5\rangle}^{\rho_4}, \overbrace{\langle 1,9,3\rangle}^{\rho_5}, \overbrace{\langle 6,1,4\rangle}^{\rho_6}, \overbrace{\langle 4,5,2\rangle}^{\rho_7}, \overbrace{\langle 7,8,5\rangle}^{\rho_8} \right\}$$

$$mbr(P) = \left\langle \langle 1,1,0\rangle, \langle 1,9,0\rangle, \langle 8,9,0\rangle, \langle 8,1,0\rangle \right\rangle$$

Another area type of great interest is the *planar convex hull*. In theory, the concept of a convex hull is applicable to any number of dimensions. In the two dimensional case that we consider in our work, the convex hull is often explained with the rubber band analogy:

> “Imagine that the points [positions] are nails sticking out of the plane, take an elastic rubber band, hold it around the nails, and let it go. It will snap around the nails, minimizing its length. The area enclosed by the rubber band is the convex hull of P [the set of points/positions].” [BCK+08, p. 3]

According to this analogy, to the definition given in [BCK+08, p. 3], and to the input and the output of the Graham scan algorithm [Gra72], a widely-used algorithm for computing convex hulls, the convex hull of a finite set of two dimensional positions is a tuple of two dimensional positions that specifies a convex polygon containing all positions of the set and whose elements form a subset of the position set. We define the planar convex hull of a finite set of three dimensional positions as follows:

Definition 7.8 Planar Convex Hull

The planar convex hull $pch(P)$ of a finite set of three dimensional positions P is a tuple of planar positions specifying a convex polygon on the xy-plane which contains the planar projection of all positions in P.

$$contained(\rho, pch(P)) = \top \quad \text{for all } \rho \in P$$

The set formed by the elements of $pch(P)$ is a subset of the set formed by the planar projections of all three dimensional positions in P.

$$\left\{\rho \;\middle|\; \rho \in pch(P)\right\} \subseteq \left\{planar(\rho) \;\middle|\; \rho \in P\right\}$$

Example 7.6 Planar Convex Hull

Figure 7.5 illustrates the planar convex hull $pch(P)$ for a set of three dimensional positions P.

$$P = \left\{\overbrace{\langle 3,7,4\rangle}^{\rho_1}, \overbrace{\langle 1,2,1\rangle}^{\rho_2}, \overbrace{\langle 8,4,1\rangle}^{\rho_3}, \overbrace{\langle 2,3,5\rangle}^{\rho_4}, \overbrace{\langle 1,9,3\rangle}^{\rho_5}, \overbrace{\langle 6,1,4\rangle}^{\rho_6}, \overbrace{\langle 4,5,2\rangle}^{\rho_7}, \overbrace{\langle 7,8,5\rangle}^{\rho_8}\right\}$$

$$pch(P) = \left\langle \underbrace{\langle 1,2,0\rangle}_{planar(\rho_2)}, \underbrace{\langle 1,9,0\rangle}_{planar(\rho_5)}, \underbrace{\langle 7,8,0\rangle}_{planar(\rho_8)}, \underbrace{\langle 8,4,0\rangle}_{planar(\rho_3)}, \underbrace{\langle 6,1,0\rangle}_{planar(\rho_6)} \right\rangle$$

When assessing an area which is spanned by a set of planarly projected positions (e.g., a minimum bounding rectangle or a planar convex hull of a set of

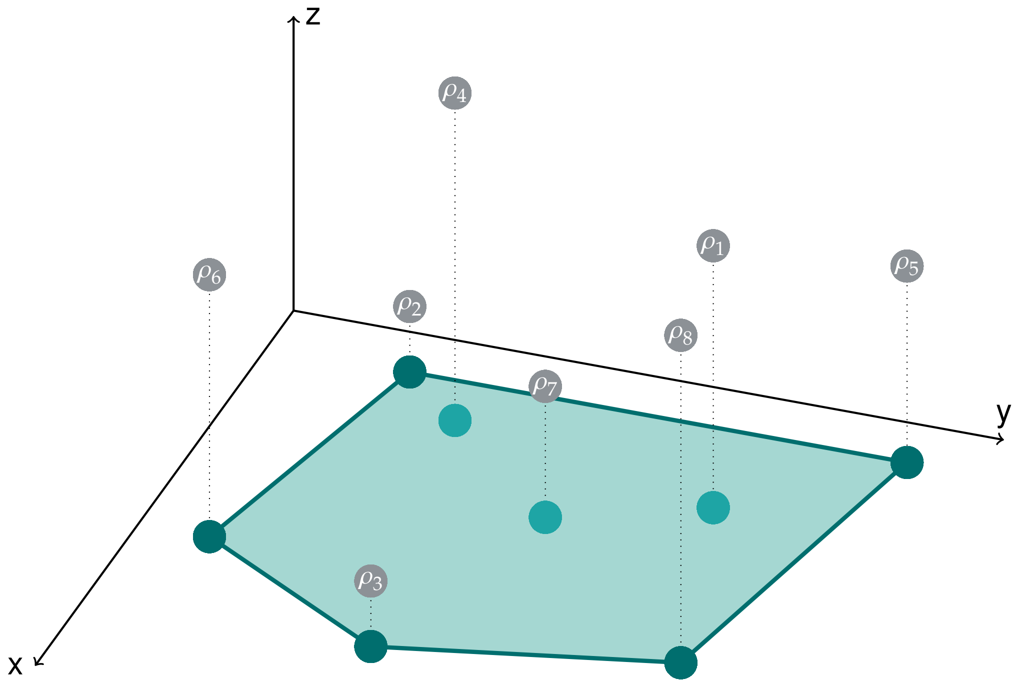


Figure 7.5 Planar Convex Hull. The light-mint area with dark-mint edges visualizes the planar convex hull $pch(P)$ of a set of positions $P = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8\}$. The three dimensional positions P are colored in gray. The planar projections (of the three dimensional positions) that specify the polygon representing the planar convex hull $pch(P)$ are colored in dark-mint. The remaining planar projections $\{planar(\rho) \mid \rho \in P\} \setminus pch(P)$ are colored in mint.

positions) the most expressive metric is the surface of the area. In a worker of a data stream analysis system the surface of an area can be used to evaluate how large a spanned area is. The result can be used as input for further computations, such as to calculate the density of a group of objects, at the same worker or if it is emitted in a state stream (such as the team area state stream exemplified in Example 4.4) at subsequent workers. As an area is specified by means of a polygon the surface of an area can be calculated by means of calculating the surface of the polygon specifying the area. We introduce a *surface* function for calculating the surface of a polygon as follows:

Definition 7.9 Polygon Surface

The function $surface(pos)$ calculates the surface of a polygon specified by a tuple pos of planar positions (i.e., $\rho.z = 0$ for all $\rho \in pos$).

Many polygons that specify areas spanned by a set of planar positions do not have any overlapping edges and are thus simple [Pag19]. For instance, both the minimum bounding rectangle and the planar convex hull of any arbitrary set of positions are simple polygons. The Surveyor's formula [Bra86] enables calculating the surface of arbitrary simple polygons. Moreover, the surface of the minimum bounding rectangle of a set of positions (i.e., *surface*(*mbr*(*P*)) where *P* is a set of positions) as well as of any other axis-aligned rectangle can be calculated even simpler by means of the following algorithm:

Algorithm 7.2 Surface of an Axis-aligned Rectangle

Input: A tuple of planar positions *pos* that specifies the corners of an axis-aligned rectangle
Output: The surface of the rectangle

```
1: $x_{min} \leftarrow \min \left\{ \rho.x \mid \rho \in pos \right\}$
2: $y_{min} \leftarrow \min \left\{ \rho.y \mid \rho \in pos \right\}$
3: $x_{max} \leftarrow \max \left\{ \rho.x \mid \rho \in pos \right\}$
4: $y_{max} \leftarrow \max \left\{ \rho.y \mid \rho \in pos \right\}$
5: return $(x_{max} - x_{min}) \cdot (y_{max} - y_{min})$
```

Part III

Implementation

8 StreamTeam

In Part II, we have defined a precise but generic model on the basis of which we have analyzed and discussed properties of worker-based data stream analysis systems. In Part III, we will convert this theory into practice by means of presenting and evaluating our implementation. All our generic and application-specific implementations that we will present in Part III are published on GitHub under the GNU Affero General Public License v3.0 (see Appendix B).

In this chapter, we will present StreamTeam, our generic real-time data stream analysis infrastructure. More precisely, we will first give an overview of StreamTeam's architecture in Section 8.1. Subsequently, Section 8.2 presents details on our data stream analysis system prototype and thus on our implementation of the model which we have presented in Part II. Finally, we will present in Section 8.3 how we achieved that external Web clients can consume all input and output stream elements and how the data stream analysis system can be debugged with our generic cluster monitor.

8.1 Infrastructure

In this section, we will give an overview of the architecture of StreamTeam, our real-time data stream analysis infrastructure, which is depicted in Figure 8.1. More precisely, we will briefly present all individual elements as well as how they interact to form the overall infrastructure. Details about the different architectural elements will be given in the subsequent sections.

The data stream analysis system at the heart of StreamTeam consumes data streams as its sole input and emits data streams as its sole output. More precisely, in compliance with our system model, StreamTeam's data stream analy-

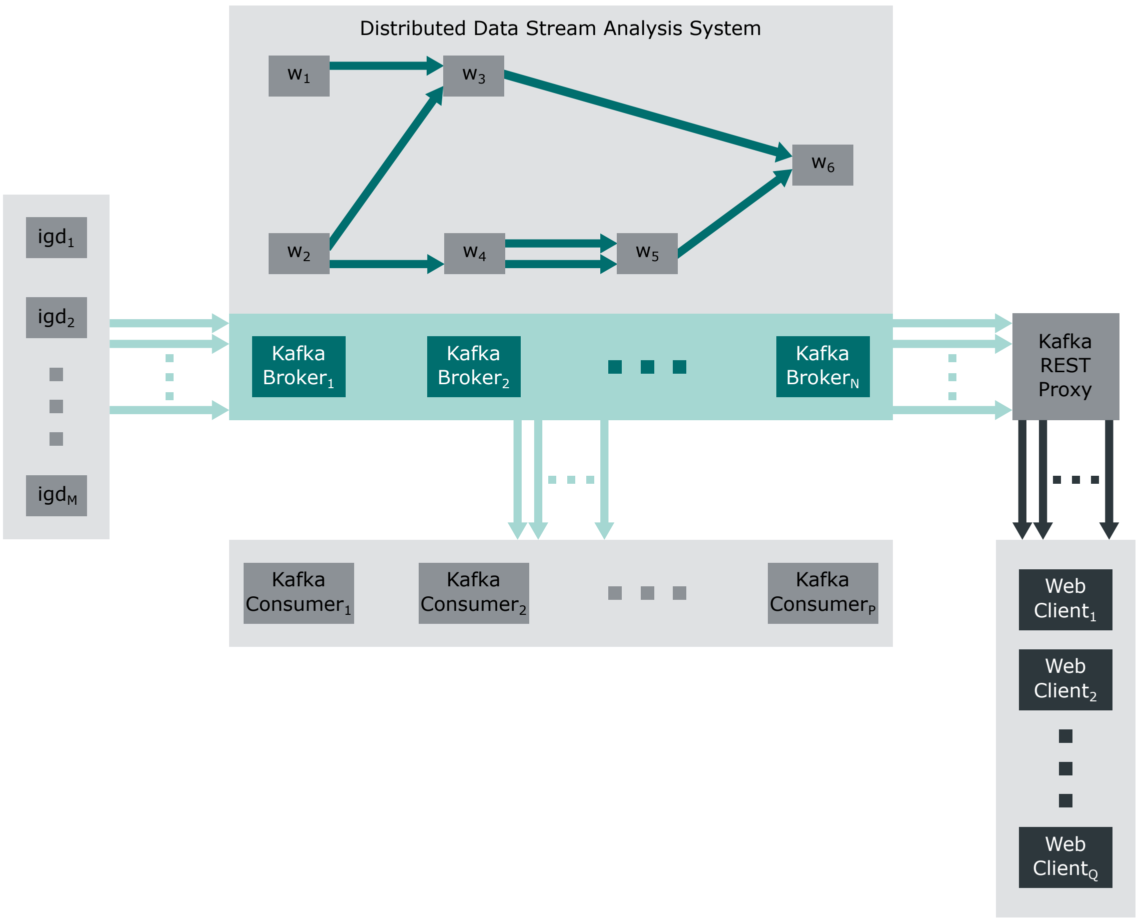


Figure 8.1 **Architecture Overview.** Except for the workers (w_1 to w_6) which are conceptual components that specify the analysis subtasks performed by Samza tasks (see Section 8.2 for more details on the data stream analysis system including its distributed deployment), each box represents a component of the StreamTeam infrastructure or a raw input stream generating device (igd_1 to igd_M). The arrows illustrate data transfer between the components. Each light-mint arrow visualizes that the elements of a data stream are pushed to or pulled from Kafka brokers. Each dark-mint arrow illustrates that elements of a data stream are transfered between the Samza tasks which perform the analysis subtasks specified by two workers with a hybrid communication model using the Kafka brokers as the communication proxies. The dark-gray arrows visualize that data stream elements are fetched from the Kafka REST proxy via its REST API. The light-gray and light-mint boxes around the raw input stream generating devices, the Kafka consumers, and the Web clients do not mean that all components in such a box are deployed on the same machine. Instead, they are used to group components in order to reduce the number of arrows. This is another simplification, besides the workers which each replace one or multiple Samza containers deployed on the same or different machines which each execute one or multiple Samza tasks and the dark-mint arrows which each replace multiple light-mint arrows, for keeping the overview figure readable.

sis system consumes only raw input stream elements which are emitted by raw input stream generating devices (see Section 5.1). There can be multiple raw input stream generating devices provided that their clocks are synchronized in order to guarantee a consistent generation time space (see Section 6.1.1). Moreover, StreamTeam supports diverse raw input stream generating devices ranging from low-level sensors (e.g., GPS or heart rate sensors) to devices with a full-fledged operating system (e.g., a Linux machine which runs a program that extracts positions from video signals). In Section 9.1, we will present the raw input stream generating devices which we use to test and evaluate our real-time football analysis application.

In StreamTeam we leverage Apache Kafka [KNR11], a widely-used publish/subscribe system, for shipping data stream elements between components and thus implement a hybrid communication model as described in Section 5.4.2. This design choice implies a single additional assumption on the raw input stream generating devices which was not present in our model, namely that we assume that every raw input stream generating device is able to interact with Kafka. More precisely, we assume that every raw input stream generating device is able to construct raw input stream elements containing the input data it measures or generates and to push these elements to a Kafka broker which provides the elements to other components and thus acts as the communication proxy. Unfortunately, there are devices which are not capable of doing so but still generate interesting input data that should be analyzed by the data stream analysis system. This is often the case for low-level sensors which neither offer interoperability with Kafka innately nor support extending their software to make use of an existing Kafka library[1] or to implement a dedicated specialized Kafka producer from scratch. To solve this problem, one can deploy an auxiliary component which receives or pulls the input data from such devices, packs them into proper raw input stream elements, and pushes these elements by using an existing Kafka library to a Kafka broker. Note that in this case we regard the auxiliary component instead of the original device as the raw input generating device and the transfer of the input data to the auxiliary component as a part of the measurement procedure in order to be in accordance with our model.

StreamTeam's data stream analysis system is a fully-parallelized worker-based data stream analysis system as described in our system model. That is, the raw input stream elements which are emitted by the raw input stream generating devices are analyzed stepwise and in parallel in the workflow of

[1] In addition to the original Java library which is developed in Kafka's main code base there are plenty of other Kafka client libraries for diverse programming languages [Apa19].

the data stream analysis system. More precisely, the data stream analysis system consists of workers which each specify the procedure to perform a subtask (e.g., pass detection) of the overall analysis (e.g., extensive analysis of a football match). Each worker specification encompasses a set of input streams (raw input, event, state, and/or statistics streams) on the basis of whose elements the subtask is performed, an optional timer period, the code which has to be performed when an input stream element is received or the timer is triggered, and a set of output streams (event, state, and/or statistics streams) in which the analysis results are published. These specifications implicitly form the overall workflow of the data stream analysis system. The actual execution of the analysis is then partitioned to components which perform the analysis subtasks of the workers separately for every key. Our prototype implementation is based on Apache Samza [NPP+17], a famous worker-based data stream analysis system, but introduces an additional modularization level inside the workers that helps structuring and reusing code, facilitate a stricter separation per key which is beneficial in many applications, establishes a well-defined data model implemented with Google Protocol Buffer [Goo19] message types, and adds support for ingestion timestamps. More details about our data stream analysis system prototype and the deviations from our model that are caused by the implementation will be given in Section 8.2.

The output stream elements of the data stream analysis system, i.e., the event, state, and statistics stream elements emitted by the components which execute the analysis subtasks specified by the workers of the data stream analysis system, can be consumed by external[2] consumers via Kafka. The same is true for all raw input stream elements emitted by raw input stream generating devices. StreamTeam supports two types of external consumers.

On the one hand there are Kafka consumers which can consume data stream elements directly from the Kafka brokers. One example for such a Kafka consumer is the MongoDB stream importer which we will present in Section 9.4.2.

On the other hand there are JavaScript-based Web clients which serve as platform-independent user interfaces. As depicted in Figure 8.1, there can be multiple Web clients which consume the same analysis results and input data. This is meaningful as the different Web clients might be targeted for different users and fulfill different purposes. They can be generic administration and debugging tools, such as our cluster monitor, or application-specific user interfaces, such as the real-time user interface of our football analysis application that

[2] To be consistent with our model, we do not use the term "external" with respect to the whole data stream analysis infrastructure but with respect to the data stream analysis system.

we will present in Section 9.3. However, despite their differences, all Web clients share the problem that they cannot directly interact with Kafka. The reason for this is that there is no Kafka library for plain[3] JavaScript which provides a Kafka consumer to pull data stream elements from the Kafka brokers. To solve this issue, we have added the Kafka REST proxy to our infrastructure. This dedicated Kafka consumer buffers the latest elements of all data streams and provides a Representational State Transfer (REST) API via which the Web clients can access all data stream elements. In consequence, not the Web clients but the Kafka REST proxy is the actual external consumer of the data streams. More details about the Kafka REST proxy and our cluster monitor will be given in Section 8.3.

8.2 Data Stream Analysis System

In this section, we will give more details on our data stream analysis system prototype implementation and thus on the implementation of our model which we have presented in Part II. Earlier versions of StreamTeam's data stream analysis system prototype have been presented in a lower level of detail in [PBS+17], [PRS+18], and [SRP+19].

Our data stream analysis system prototype implementation is based on Apache Samza [NPP+17]. More precisely, we use Samza version 0.13.1 as a foundation which we extend and improve.

Samza is a popular worker-based data stream analysis system. In Samza an analysis application is built by connecting conceptual jobs by means of their input and output streams to a dataflow graph [Sam17d]. This is equivalent to our system model in which conceptual workers form a workflow which is implicitly defined by the input and output stream sets of the workers (see Section 5.2.2). Moreover, Samza provides support for key-based data parallelism by means of splitting data streams into data stream partitions and jobs into tasks [Sam17d]. As explained in [Sam17b], Samza is deployed on YARN [VMD+13], the resource manager and scheduler of Apache Hadoop [Had19], and transfers data stream elements with Apache Kafka [KNR11].[4]

In the remainder of this section, we will present those aspects of our implementation which are not inherited from Samza but added by us. Moreover,

[3] Note that the Node.js libraries listed in [Apa19] cannot be used in plain JavaScript-based Web clients which run in the browser.

[4] Note that, as stated in [Sam17b], Samza does not enforce using Kafka and YARN. Instead, "Samza's execution and streaming layer are pluggable, and allow developers to implement alternatives if they prefer." [Sam17b]. However, both are supported innately [Sam17b].

we will highlight and discuss all deviations from our model which result from our implementation, i.e., from Samza or from our extensions. Whenever necessary, we will start with a detailed description of the concept, architecture, or procedure of Samza which causes a deviation or serves as a foundation for our extension.

8.2.1 Modular Code

As we have summarized in Section 8.1, according to our model, each worker specifies the procedure to perform a subtask of the overall analysis by means of specifying a set of input streams on the basis of whose elements the subtask is performed, an optional timer period, the code which has to be performed when an input stream element is received or the timer is triggered, and a set of output streams in which the analysis results are published. In the following, we will present how this specification has to be defined in our data stream analysis system prototype and thus how a StreamTeam worker can be implemented.

In our prototype, each worker is implemented as a Samza job using Samza's low-level API [Sam17a], including its windowing feature [Sam17l]. The set of input streams and the optional timer period are specified in the configuration file of the Samza job [Sam17a; Sam17l]. In contrast to our model, the output streams do not have to be specified explicitly (e.g., in the configuration file). Instead, the information in which output stream a new element is emitted is given in the code [Sam17a]. However, the configuration file has to contain (de-)serialization information for each input and output stream [Sam17i].

According to our system model, there is only a single code block (*co*) for each worker which is executed when an input stream element is received and when the timer is triggered. Even if we did not extend Samza but simply used its low-level API, we would deviate from our model as in Samza the code is split into two blocks. More precisely, a Samza job has two freely-programmable functions, namely `process(`*dse*`,...)` and `window(...)` (see [Sam17a] and [Sam17l]). The code of the former (co_{proc}) is executed when an input stream element is received and the code of the latter (co_{win}) is executed when the timer is triggered. However, since we have defined in our model that the code of each worker is freely-programmable, this deviation can be compensated by means of wrapping Samza's low-level API functions into a single code block with an if-statement. More precisely, the single code block of every worker can be set to be the following algorithm:

Algorithm 8.1 Samza's Low-Level API Functions Wrapped into a Single Code Block

Input: Received input stream element *dse*
Null ($dse = \lambda$) if the code execution is triggered by the timer
Output: No output

```
1: if $dse = \lambda$ then
2:     window(...)          ▷ Executes $co_{win}$ and pushes all generated output stream elements to Kafka.
3: else
4:     process($dse$,...)   ▷ Executes $co_{proc}$ for $dse$ and pushes all generated output stream elements to Kafka.
5: end if
```

Samza's simple low-level API with only two functions enables implementing workers straightforward with two blocks of code, one for each function. Experienced worker developers may enjoy this simplicity as they are free to design their own code structure on top of it. However, the drawback of this simplicity is that the API does not prevent worker developers from producing bad code. If the analysis logic of a worker is complex and the worker developer does not leverage a sophisticated code structure but implements each function with a single monolithic code block, this can result in very large, badly structured spaghetti code with many code duplicates.

In StreamTeam we aim to assist domain experts without a profound software engineering background (e.g., football match analysts) to implement their own analysis workers with readable, well-structured, and duplicate-free code. Therefore, our data stream analysis system prototype extends Samza by adding an additional code modularization layer on top of its low-level API.

In a nutshell, in StreamTeam, the code of a worker is formed by means of connecting modules to two module graphs. If implementing an analysis application by means of combining workers which perform a subtask of the overall analysis to a workflow, as done in all worker-based data stream analysis systems, is regarded as the first modularization level, these module graphs inside the worker can be regarded as a second modularization level (see Figure 8.2).

The first module graph structures the code of Samza's `process(`*dse*`,...)` function. Each module (m) in this graph implements logic to process exactly one data stream element at the same time. Therefore, we denote these modules as *single element processor modules* and this graph as the *single element processor graph* of the worker. In its logic, each module can store information in the state and/or generate output stream elements. The graph has start modules

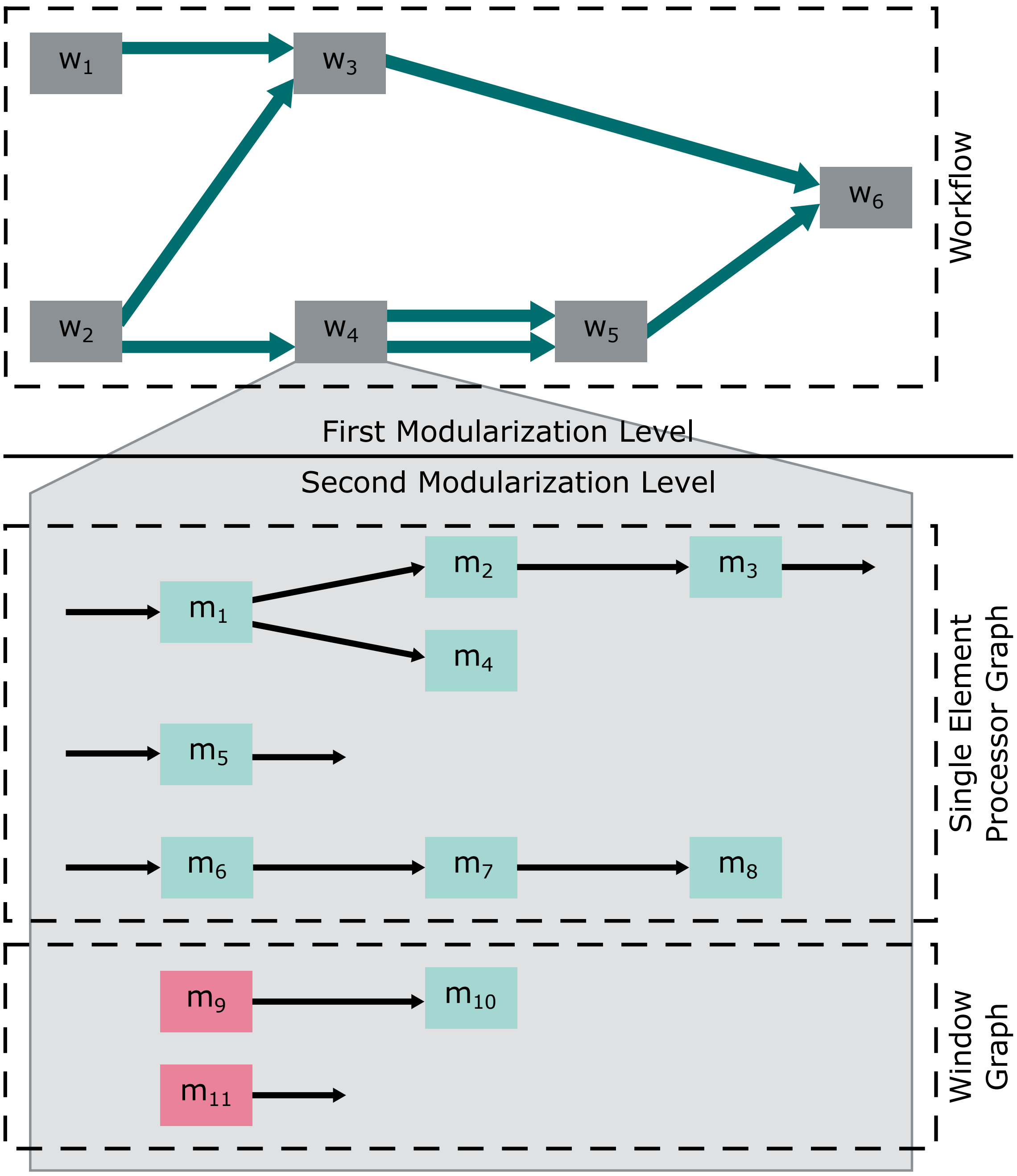

w1
w2
w3
w4
w5
w6
Workflow
First Modularization Level
Second Modularization Level
m1
m2
m3
m4
m5
m6
m7
m8
Single Element Processor Graph
m9
m10
m11
Window Graph

Figure 8.2 **StreamTeam's Two Modularization Levels.** The top shows the workflow of the data stream analysis system (as illustrated in Figure 8.1) and thus the first modularization level. The bottom depicts the second modularization level for worker w_4. The mint boxes represent single element processor modules and the red boxes represent window modules. The black arrows do not represent individual data streams but show which modules have successor modules to which they hand over the output stream elements they generate (e.g., m_1 hands the output stream elements it generates to m_2 and m_4). In addition, the black arrows illustrate which modules generate no output stream elements at all (m_4, m_8, and m_{10}), which modules generate the output stream elements of the worker (m_3, m_5, and m_{11}), and which modules process the input stream elements the worker (or more precisely a task of the Samza job) receives (m_1, m_5, and m_6).

($M_{startSEPG}$) and each module can have successor modules (M_{succ}) which further process its output stream elements.

Whenever a processor of the worker or more precisely a task of the Samza job receives a new input stream element and thus Samza's `process(`*dse*`,...)` function is called, the single element processor graph is executed in a depth-first way. That is, if the logic of a module of the single element processor graph generates at least one output stream element and if the module has at least one successor module, the first output stream element is handed to its first successor module. If the logic of the module has generated multiple output stream elements, the second output stream element is handed to the first successor module when the first successor module (and all its successors) finished its execution for the first output stream element. This is continued until all generated output stream elements are processed by the first successor module. If the module has multiple successor modules, the same procedure is done for the second successor module when the first successor module (and all its successors) finished its execution for all generated output stream elements, and so on, until the execution is finished for all generated output stream elements and all successor modules.

The depth-first execution is started when Samza's `process(`*dse*`,...)` function is called by handing the input stream element received by the Samza task to the first start module. When the first start module (and all its successors) finished the execution, the input stream element is handed to the second start module, and so on, until the whole single element processor graph has performed its work for the received input stream element. If the logic of a module generates output stream elements but the module has no successor modules, these module output stream elements are output stream elements of the worker and thus pushed by the Samza task to a Kafka broker.

In addition to the single element processor graph, there is a *window graph* which structures the code of Samza's `window(...)` function. The sole but crucial difference between the single element processor graph and the window graph is that the window graph has window modules as start modules ($M_{startWG}$). A *window module* implements logic which has to be performed when the timer of the worker is triggered. Besides storing information in the state, window modules can also generate output stream elements in their logic. To further process these output stream elements a window module can have single element processor modules (but no window modules) as its successors.

The window graph is executed when the timer of a processor of the worker or more precisely of a task of the Samza job is triggered and thus Samza's `window(...)` function is called in the same depth-first way, as described for the single element processor graph. The only difference in the execution procedure is that there is no received input stream element which is handed to the start modules.

In the following, we present the algorithms which we implemented in our data stream analysis system prototype to add the additional modularization layer on top of Samza's low-level API:

Algorithm 8.2 Implementation of Samza's Process Function in StreamTeam

```
Input:  Received input stream element $dse$
Output: No output

1: Assign additional information to $dse$ (see Section 8.2.3 and Section 8.2.4)
2: $DSE_{out} \leftarrow \emptyset$
3: for all $m_{start} \in M_{startSEPG}$ do        ▷ Iterates over all start modules of the single element processor graph.
4:     $DSE_{out} \leftarrow DSE_{out} \cup m_{start}$.process($dse$)        ▷ Executes $m_{start}$ (and its successor modules).
5: end for
6: for all $dse_{out} \in DSE_{out}$ do
7:     Push $dse_{out}$ to Kafka
8: end for
```

Algorithm 8.3 Implementation of Samza's Window Function in StreamTeam

```
Input:  No input
Output: No output

1: $DSE_{out} \leftarrow \emptyset$
2: for all $m_{start} \in M_{startWG}$ do        ▷ Iterates over all start modules of the window graph.
3:     $DSE_{out} \leftarrow DSE_{out} \cup m_{start}$.window()        ▷ Executes $m_{start}$ (and its successor modules).
4: end for
5: for all $dse_{out} \in DSE_{out}$ do
6:     Push $dse_{out}$ to Kafka
7: end for
```

Algorithm 8.4 Implementation of the Process Function of a Single Element Processor Module

Input: Received input stream element *dse*
Output: Output stream elements that have to be pushed to Kafka

```
1: $DSE_{outCur} \leftarrow \emptyset$
2: Execute the logic of the module for $dse$ (can add elements to $DSE_{ourCur}$)
3: if $M_{succ} = \emptyset$ then                         ▷ If the module has no successor modules.
4:     return $DSE_{outCur}$
5: else
6:     $DSE_{outSucc} \leftarrow \emptyset$
7:     for all $m_{succ} \in M_{succ}$ do                 ▷ Iterates over all successor modules.
8:         for all $dse_{outCur} \in DSE_{outCur}$ do      ▷ Iterates over all elements generated
                                                            in the module logic.
9:             $DSE_{outSucc} \leftarrow DSE_{outSucc}$
                       $\cup\, m_{succ}$.process($dse_{outCur}$)   ▷ Executes $m_{succ}$ (and its
                                                                      successor modules).
10:        end for
11:    end for
12:    return $DSE_{outSucc}$
13: end if
```

Algorithm 8.5 Implementation of the Window Function of a Window Module

Input: No input
Output: Output stream elements that have to be pushed to Kafka

```
1: $DSE_{outCur} \leftarrow \emptyset$
2: Execute the logic of the module (can add elements to $DSE_{ourCur}$)
3: if $M_{succ} = \emptyset$ then                         ▷ If the module has no successor modules.
4:     return $DSE_{outCur}$
5: else
6:     $DSE_{outSucc} \leftarrow \emptyset$
7:     for all $m_{succ} \in M_{succ}$ do                 ▷ Iterates over all successor modules.
8:         for all $dse_{outCur} \in DSE_{outCur}$ do      ▷ Iterates over all elements generated
                                                            in the module logic.
9:             $DSE_{outSucc} \leftarrow DSE_{outSucc}$
                       $\cup\, m_{succ}$.process($dse_{outCur}$)   ▷ Executes $m_{succ}$ (and its
                                                                      successor modules).
10:        end for
11:    end for
12:    return $DSE_{outSucc}$
13: end if
```

So far, our data stream analysis system prototype provides a generic filter module which filters input stream elements based on the information they ship (e.g., the object identifier they contain or the data stream they belong to), a generic store module which extracts information from input stream elements and stores the extracted information in the state (e.g., keep a history of the generation timestamp of the last five input stream elements or store the second position contained in the last input stream element), and two auxiliary modules for executing timer-triggered code separately for all active keys which we will present in Section 8.2.2.2. In our real-time football analysis application (see Chapter 9), we used these modules in multiple workers with different parameters to serve diverse purposes. Based on this experience, we argue that having such generic modules is a very helpful tool for reusing code and avoiding code duplication. However, we also experienced that having the additional option to further add worker-specific modules with a freely-programmable logic (e.g., a module for detecting passes based on filtered input stream elements and information stored in the state) to the module graphs is beneficial as it enables worker developers to implement workers that perform arbitrarily complex analysis subtasks without hacking around non-fitting generic modules. In consequence, we designed our module graphs in a way that they can consist of both, parameterizable generic modules and freely-programmable worker-specific modules.

As the presence of a generic store module already implies, the state is shared between all modules of both module graphs of a worker. That is, it is possible, for instance, to build a single element processor graph that contains a store module which stores the generation timestamp of the last ball hit event stream element in the state and a worker-specific pass detection module which accesses this stored timestamp. Furthermore, the pass detection module can increase a pass counter in the state which is accessed by a worker-specific module in the window graph which periodically generates pass statistics stream elements. However, the state is not shared between Samza tasks. More details on that as well as on how we facilitate ensuring the strict key-based separation that we have defined in our system model will be given in Section 8.2.2.

Moreover, as stated in [Sam17f; Sam17l], Samza is thread-safe since (at least in its default concurrency level which we use) each Samza task is single-threaded and thus never executes the code of the `process(`*dse*`,...)` function for multiple data stream elements or the code of the `process(`*dse*`,...)` function and the code of the `window(...)` function at the same time even if this requires deferring the timer.[5] This thread-safety is not harmed by our additional modularization layer. In consequence, a worker developer can safely modify and access the state in

[5] As stated in Literature Discussion 5.5, we have adopted this part of Samza's processing procedure into our model.

multiple modules without considering concurrency.

As a last point we want to highlight that our module graph approach supports also implementing workers without `process(`dse`,...)` or `window(...)` code. If one wants to implement a worker without a timer (i.e., $w.T = \lambda$), this can be done without providing any `window(...)` code by means of leaving the window graph empty, i.e., by not defining any start modules for the window graph. Similarly, a worker which does not consume any input stream (i.e., $w.DS_{in} = \emptyset$) can be implemented without providing any `process(`dse`,...)` code by means of leaving the single element processor graph empty, i.e., by not defining any start module for the single element processor graph.

Literature Discussion 8.1 Worker Code Modularization

The idea to further modularize the code of a Samza job has not only been pursued by us. In parallel to our work, the contributors of Samza introduced a new high-level API [NPP+17; Sam20].

Samza's high-level API enables worker developers to implement analysis workers by means of connecting operators to a graph [NPP+17; Sam20]. Each operator is a higher-order function which enables the worker developer to specify arbitrary worker-specific logic blocks.[6]

The higher-order function development style is often regarded to be the quicker and more comfortable option to implement a worker with a simple logic (e.g., filtering tweets [Twi20] by topic and generating statistics for each country). However, implementing a worker in Samza's high-level API requires the worker developers to have a certain understanding of the functional programming model. We doubt that this is the case for the domain experts we aim to assist with our module graph approach (e.g., match analysts who have so far only written small analysis scripts). Moreover, we argue that the higher-order function development style looses its superiority if a worker with a very complex logic should be implemented. For instance, the pass and shot detection worker of our real-time football analysis application (see Section 9.2.1.10) contains a complex logic to distinguish between the different pass and shot events which is packed on purpose into a single worker-specific module as further

[6] Note that this high-level API does not convert Samza from a worker-based to a graph-based data stream analysis system. The reason for this is that the logic of each worker is still implemented separately. Only the coding style has changed. In contrast, in graph-based data stream analysis systems not only an individual analysis subtask (i.e., the worker) but the overall analysis task (i.e., the whole workflow) is implemented by concatenating higher-order functions to a graph (see Section 3.1).

splitting it would complicate keeping the overview. Packing the logic into a single operator of Samza's high-level API would yield the same result. Moreover, Samza's high-level API removes the separation between element-triggered and time-triggered code and instead introduces a dedicated window operator [NPP+17; Sam20]. Although this again might be more comfortable to implement simple analysis logic, this can even complicate implementing complex workers. For instance, periodically executing some code for each active key as supported by StreamTeam (see Section 8.2.2.2 for more details) requires hacking around Samza's window operator.

8.2.2 Strict Key-Based Separation

In our model, we separate the data and the analysis procedure strictly per key. More precisely, according to our definitions for each data stream there is exactly one partition for every key (see Section 4.3) and for each worker there is exactly one processor for every key which performs the analysis subtask defined by the worker for the input stream elements with a certain key and thus for the input stream elements belonging to the input stream partitions for this key (see Section 5.3.1). Moreover, we have defined each processor to have its individual local state which is not and cannot be shared with other processors (see Section 5.3.1). In consequence, the whole analysis task is performed independently for every key.

As mentioned above, also Samza provides support for key-based data parallelism by means of splitting data streams into data stream partitions and jobs into tasks [Sam17d]. In fact, we have adopted the idea to partition data streams and distribute the workload by means of the keys of the data stream elements from Samza's fundamental concepts (see Literature Discussion 4.1 and Literature Discussion 5.4). However, as briefly summarized in Literature Discussion 5.4, Samza defines data stream partitions a little different than we do in our model.

While we define in our model that each data stream is split in a way that there is exactly one data stream partition for each key in the key domain, this is not the case in Samza. Instead, Samza adopts the data stream partitioning and thus also the number of partitions from the streaming layer [Sam17h]. If Apache Kafka is used, as we do in StreamTeam, the number of partitions into which each data stream is split can be configured globally by means of configuring the number of partitions per data stream in the Kafka configuration by setting `num.partitions` [Sam17h].[7] Nevertheless, also in Samza the keys of the data stream elements are regarded when splitting a data stream into partitions. More precisely, all elements which belong to the same data stream and have the same key are guaranteed to be assigned to the same data stream partition [Sam17d]. However, in Samza a partition of a data stream does not only comprise the elements of the data stream with a single key but the elements of the data stream whose key is in a certain subset of the key domain. The key domain subsets of the different partitions of the data stream are determined by Samza's hash-based partitioning function.[8] The size of each subset depends on the size of the key domain and the number of partitions.

In order to realize a data-parallel analysis procedure, Samza splits the analysis workload by means of assigning different data stream partitions to different tasks. More precisely, in Samza's default partitioning mode which we use in StreamTeam, "[e]ach task consumes data from one partition for each of the job's input streams" [Sam17d] and the number of tasks is equal to "the maxmimum number of partitions across all input streams" [Sam17h] and thus in our implementation equal to the global number of partitions (`num.partitions`). Moreover, if two elements of two input streams have the same key, they are guaranteed to be assigned to data stream partitions that are processed by the same Samza task [Sam17h]. Hence, there is only one difference to the key-based data parallelism described in our model, namely that each Samza task performs the analysis for a subset of the key domain which belongs to the same data stream partition instead of only for a single key as we have defined for the processors. This difference is caused by the different data stream partitioning approach.

As already indicated in Footnote 5 in Section 5.3.1, the size of the key domain can be very huge (e.g., $|Dom_k| = 4'294'967'295$ if each key is a 32-bit Integer) or even infinite (e.g., $|Dom_k| = \infty$ if each key is an arbitrary String). Although it is unproblematic and in our opinion much cleaner and more intuitive to define data stream partitions and processors on a single-key basis in a theoretical model, as we have done in Part II, it would not be possible in practice to deploy and run a very large not to mention infinite number of processors from

[7] Moreover, it is also possible to specify the number of partitions individually for a certain data stream [Sam17h]. However, we do not make use of this option and therefore assume that all data streams are split into the same number of partitions which is specified by `num.partitions`.

[8] More precisely, Samza hashes the key and uses the modulo function to map the hash value to the partition numbers. This information can be extraced from the code of Samza version 0.13.1, namely from line 57 in `KafkaUtils` (see https://github.com/apache/samza/blob/0.13.1/samza-kafka/src/main/scala/org/apache/samza/util/KafkaUtil.scala, last accessed at 06.03.2020).

which the most do not perform any work as there are usually not data stream elements for every key. If the key domain is not designed on purpose to be very small, configuring the number of partitions results in a much lower and thus more manageable number of components which have to be executed but still supports scalability by increasing the configuration parameters.[9] Therefore, we argue that Samza's approach is an elegant solution to this problem.

Nevertheless, we also argue that separating the data and the analysis procedure strictly per key, as described in our model, is very beneficial as doing so enables leveraging application-defined keys to separate the analysis. For instance, in our real-time football analysis application (see Chapter 9) we separate the analysis of different football matches by means of setting the key of every data stream element to be the unique identifier of the match it belongs to and thus enable analyzing multiple matches independently in parallel.

Samza does not guarantee the strict separation per key which we have defined in our model since the same Samza task is responsible to process data stream elements with different keys that belong the same data stream partition. For instance, a Samza task of a football analysis application is responsible to process data stream elements belonging to two different matches if the identifiers of both matches lead to the fact that the elements belonging to the different matches are assigned to the same partitions. Hence, the worker developers have to take care for separating the analysis per key by themselves. This is a challenging task for domain experts without a profound software engining background. As we aim to assist domain experts in developing their own analysis workers, StreamTeam adds additional extensions on top of Samza which facilitate implementing workers that perform the analysis strictly separated per key. These extensions will be presented in the remainder of this section.

8.2.2.1 State Abstractions

In our system model, we have only stated that a worker can be stateful and that each processor has its individual local state which is not shared with other processors. However, we have not defined in more detail how the state is managed but regarded this as an implementation detail. Since our data stream analysis system prototype implementation is based on Samza, StreamTeam inherits Samza's state management. In a nutshell, Samza ships a key-value store that is based on RocksDB [Fac20] and offers a key-value store API to store and access

[9] As we will describe in more detail in Section 8.2.5, Samza introduces an additional container layer which enables controlling the deployment individually for each StreamTeam worker.

state [NPP+17; KK15; Sam17j]. As stated in Section 5.4.1, failure tolerance was not a topic of our research and is thus not discussed in this thesis. Nevertheless, we want to briefly mention that Samza's state management is failure-tolerant (see [NPP+17] and [Sam17j] for more details) and that we inherit this failure-tolerance in our implementation.

In Samza each task has its individual local key-value store which is not accessible by other tasks [NPP+17; KK15; Sam17j].[10] If the state was further strictly separated per key, this would be sufficient to guarantee that the analysis is performed independently for every key since each single element processor module processes only a single data stream element at the same time and thus only accesses to the state can violate the key boundaries. However, in Samza there is no mechanism which separates the state automatically per key. Instead a Samza task can access and modify the same information in the store when processing a data stream element with k_1 and when processing another data stream element with k_2 if both element belong to the same data stream partition.

Admittedly, the potential to access and modify the state across key boundaries can be beneficial in some edge cases. We actually make use of this in our active keys modules to maintain the set of active keys (see Section 8.2.2.2). However, having this potential also leads to the fact that the worker developers have to be very careful in selecting proper key-value store keys for storing state as values in the key-value store in order to prevent corrupting the predominantly desired and meaningful strict separation per key unintentionally. This is contradictory to our goal to assist worker developers without a profound software engineering background. Therefore, StreamTeam adds two state abstractions, the `SingleValueStore` and the `HistoryStore`, on top of Samza's key-value store API which facilitate separating the state on a single-key basis but still enable breaking the key boundaries by intention if necessary.

The `SingleValueStore` class is a wrapper around Samza's `KeyValueStore` class which supports writing arbitrary values to the key-value store by means of a `put(...)` function and reading values from the key-value store by means of a `get(...)` function. In contrast to Samza's `KeyValueStore`, the `SingleValueStore` does not enable the worker developer to specify the key-value store key *kvsk* for which the currently stored value should be read from or under which a new value v should be stored in the key-value store directly when calling these functions. Instead, the key-value store key is constructed by the `SingleValueStore`

[10] In fact, Samza supports tasks to initialize and use multiple key-value store instances and even using instances of other store engines [Sam17j], but StreamTeam initializes only a single key-value store instance for each task which is used to store the whole state of this task.

instance by means of concatenating the partitioning key *k*, a single value store key *svsk*, and an inner key *ik* to a single String.[11]

The single value store key has to be specified only once together with the `KeyValueStore` instance when creating the `SingleValueStore` instance. The partitioning key and the inner key have to be specified for each get or put operation. That is, every `SingleValueStore` instance provides a `get(`*k*`,`*ik*`)` function and a `put(`*k*`,`*ik*`,`*v*`)` function which the worker developer can use in the logic of the single element processor modules and the window modules. For instance, in our real-time football analysis application, the number of successful passes for each team (which is emitted in elements of the pass statistics stream) is stored in a `SingleValueStore` instance which is created with "numSuccessfulPasses" as the single value store key. The stored number of successful passes for a certain team in a certain football match can be accessed and modified by calling the `get(`*k*`,`*ik*`)` function and the `put(`*k*`,`*ik*`,`*v*`)` function of this `SingleValueStore` instance with the match identifier as the partitioning key and the team identifier as the inner key. When called the `get(`*k*`,`*ik*`)` function and the `put(`*k*`,`*ik*`,`*v*`)` function construct the key-value store key *kvsk* and call the `get(`*kvsk*`)` function and the `put(`*kvsk*`,`*v*`)` function of the `KeyValueStore` instance, respectively.

The benefit of this abstraction is that the worker developer automatically uses only proper key-value store keys which guarantee a strict separation per key when he/she calls the functions with the correct partitioning key (*k*). This is much easier than properly selecting the whole key-value store key (*kvsk*) as doing it wrong would require explicitly specifying a wrong partitioning key.[12]

In order to further simplify the development process, the `SingleValueStore` abstraction provides a second version of both functions in which the partitioning key and the inner key are automatically generated. More precisely, each `SingleValueStore` instance provides a `get(`*dse*`)` and a `put(`*dse*`,`*v*`)` function which are intended to be called with the received input stream element as *dse*. When called these functions automatically extract the partitioning key *k* from the data stream element *dse*. Moreover, they generate the inner key *ik* by applying a schema to the data stream element. This inner key schema is specified together with the `KeyValueStore` instance and the single value store key *svsk* when creating the `SingleValueStore` instance. The schema can return always a constant

[11] In our current prototype we require that values have to be objects of Java classes which implement the `java.io.Serializable` interface and that the partitioning key, the single value store key, and the inner key are Strings.

[12] Note that our functions reduce the chance that the worker developer breaks the key boundaries unintentionally but still enable breaking the key boundaries intentionally if necessary (as for instance in our active keys modules).

String or extract arbitrary information from the data stream element (e.g., the object identifier it is assigned to, the name of the data stream it belongs to, or arbitrary data from its payload). The `get(`*dse*`)` and the `put(`*dse*`,`*v*`)` function are heavily used in the generic store modules. For instance, in our real-time football analysis application, the ball possession worker (see Section 9.2.1.6) comprises a store module which stores the latest position of each player by applying an inner key schema which extracts the player identifier to each received field object state stream element and the pass and shot detection worker (see Section 9.2.1.10) contains a store module which stores the latest kick position by applying an inner key schema which returns always a constant String (since there is only one latest kick position in each match) to each received kick event stream element. However, both functions can also be used in the logic of worker-specific modules. Moreover, it is also possible to block the `get(`*dse*`)` and the `put(`*dse*`,`*v*`)` function if the inner key is not constant and cannot be extracted from the data stream element by means of creating the `SingleValueStore` instance with a dedicated dummy schema which causes both functions to throw an exception if it is applied. This is for instance done for the `SingleValueStore` instance that is used to maintain the set of active keys in the active keys modules.

The `HistoryStore` class is another wrapper around Samza's `KeyValueStore` class which works similar as the `SingleValueStore` class. However, it facilitates storing a history of values with a certain length instead of only single values. More precisely, instances of the `HistoryStore` class do not provide a `get(`*k*`,`*ik*`)` function and a `put(`*k*`,`*ik*`,`*v*`)` function but a `getLatest(`*k*`,`*ik*`)` function which returns the latest value stored in the history, a `getList(`*k*`,`*ik*`)` function which returns the whole history as a list, and an `add(`*k*`,`*ik*`,`*v*`)` function which adds a new value to the history (with automatic history length capping).[13] Moreover, there is again a second version of all three functions in which the partitioning key and the inner key are automatically generated.

8.2.2.2 Active Keys Modules

In our system model, we have defined that the timer is triggered periodically at every processor and thus for every key. However, Samza's `window(...)` function is only called and thus the window graph of a worker is only executed for each Samza task [Sam17l]. Moreover, when a start module of the window graph is executed its window function is called without any key context (see

[13] Admittedly, it is also possible to store a value history using a `SingleValueStore` instance but doing so in a `HistoryStore` instance is much more comfortable.

Algorithm 8.3 and Algorithm 8.5).

One option would be to execute the logic of the start module for all keys which are in the subset of the key domain for which the Samza task is responsible. However, as described above, the size of the key domain and thus also the size of this subset can be huge or even infinite (e.g., if String keys are used as in our prototype). Thus, we have refrained from implementing this option.

Instead, StreamTeam ships two modules, namely an active keys processor module and an active keys window module, which together enable executing logic periodically for every active key. We define a key to be active with respect to a Samza task if this task has processed a data stream element with this key recently, or more precisely in the last $\Delta\tau$ milliseconds where $\Delta\tau$ is a configurable parameter.

The active keys processor module is a single element processor module which has to be added as a start module to the single element processor graph.[14] The active keys window module is a window module which has to be added as a start module to the window graph. Together, the active keys processor module and the active keys window module maintain a set of active keys and cause the logic of a single or multiple worker-specific single element processor modules to be executed periodically for each key in this set.

For this purpose, the active keys processor module adds the key of every data stream element for which its `process(`*dse*`)` function is called to a set which is stored in the key-value store. This is done by means of a `SingleValueStore` instance. However, in order to maintain a single set containing all active keys for the Samza task not the key of the data stream element but a constant dummy String is used as the partitioning key. This is one of the rare exceptions in which it is useful to intentionally break the predominantly desired key boundaries.[15] Moreover, the active keys processor module stores the maximum of all observed processing timestamps and the maximum of all observed generation timestamps for each key in the key-value store by means of using two additional `SingleValueStore` instances as state abstractions and the data stream element

[14] Alternatively, the active keys processor module can be added at another position in the single element processor graph as long as it is guaranteed that the `process(`*dse*`)` function of the active keys processor module is called (i) regularly with a data stream element for each of the keys for which the Samza task currently receives input stream elements and (ii) for each input stream element which might contribute to the data of a data stream element that is generated in one of the successor modules of the active keys window module.

[15] Note that the state is still not shared between Samza tasks.

key as the partitioning key.[16]

When the `window()` function of the active keys window module is executed for a Samza task it first reads the active keys set from the key-value store. Subsequently, the active keys window module iterates over all keys in the set, reads the maximum processing timestamp for each key from the key-value store, and checks if this processing timestamp is too old, i.e., if the difference between this timestamp and the current system time of the Samza task is larger than $\Delta\tau$.[17] If this is the case, the active keys window module removes this key from the set. Otherwise, it additionally reads the maximum generation timestamp for this key from the key-value store and generates a new internal active keys stream element with this key and the read maximum generation timestamp as the generation timestamp. The internal active keys stream element is a dummy element without any data which is used to cause the time-triggered execution of the worker-specific single element processor modules with key context. When the active keys window module finished iterating over all keys it writes the cleaned active keys set back to the key-value store.

The worker-specific single element processor modules which implement the actual logic that is executed periodically for every active key have to be added as successor modules of the active keys window module to the window graph. The `process(`*dse*`)` function of these single element processor modules is called periodically every time when the system time of a Samza task triggers Samza's `window(...)` function (see Section 8.2.3.2 for more details) for each active key the Samza task is responsible for with an internal active keys stream element from which the key can be easily extracted. Note that, as every other module in the single element processor graph and the window graph, each of these single element processor modules can have other single element processor modules as successor modules to further process their generated output stream elements.

In combination, the active keys processor module, the active keys window module, and the worker-specific successor modules of the active keys window module mimic that there is a separate processor for every active key whose code execution is triggered periodically by a timer. Although, we admit that this solution does not implement our system model perfectly since we do not mimic

[16] Note that the maximum of all observed processing timestamps for a certain key is always the processing timestamp of the last data stream element with this key the active keys processor module has processed since, as a matter of course, elements are processed in processing time order but that this is not the case for the maximum of all observed generation timestamps since elements might be processed out-of-order with respect to their generation timestamps (see Section 8.2.3.3).

[17] As we will present in Section 8.2.3, the processing timestamp is assigned using the system time of the Samza task and thus comparable to the current system time.

that there is a separate processor for every key but only for every active key, we argue that this solution is sufficient as, at least in the scenarios we focus on, every relevant key is active. For instance, when implementing our real-time football analysis application this approach proofed to be a perfect solution for generating time window based statistics, such as ball possession statistics or heatmaps, and emitting the statistics in statistics stream elements (see Chapter 9).

8.2.3 Timestamps and Orderings

In Chapter 6, we have defined and discussed different time notions and orderings in theory in the context of our generic model that can serve as a foundation for diverse implementations. In this section, we will give some details on our implementation prototype. More precisely, we will describe how the different timestamps and the sequence numbers are created and assigned in StreamTeam (see Section 8.2.3.1), how the different timestamps should be used by the worker developers (see Section 8.2.3.2), and which ordering consistency guarantees StreamTeam provides (see Section 8.2.3.3).

8.2.3.1 Timestamp and Sequence Number Assignment

In our model, we assume that each data stream element is assigned a globally unambiguous generation timestamp that specifies the time of generation with respect to a single globally consistent generation time space (see Section 6.1.1) and a globally unambiguous sequence number which is used to order the element with respect to all other elements of the same data stream partition (see Section 6.2.1). Moreover, we assume that data stream elements are assigned processor-specific processing timestamps which each specify the moment when a processor started processing the element (see Section 6.1.3). Furthermore, according to our model, raw input stream elements are assumed to be assigned an additional data stream analysis system specific ingestion timestamp that specifies the moment when an entry component of the data stream analysis system received the element (see Section 6.1.3). In this section, we will describe how the different timestamps and the sequence numbers are created and assigned in StreamTeam, or more precisely in our data stream analysis system prototype.[18]

Generation timestamps are generated and assigned exactly as described in Section 6.1.1. That is, StreamTeam assumes that every raw input stream gener-

[18] Note that we do not assign production and emission timestamps to data stream elements since both are interesting theoretical concepts but cannot be acquired in practice (see Section 6.1.2).

ating device assigns a generation timestamp created using its local clock to each raw input stream element it emits. Moreover, StreamTeam does not take care for the necessary clock synchronization but assumes that all raw input stream generating devices have already synchronized clocks. This is in accordance with the assumptions we pose in our generic model. The generation timestamps of event, state, and statistics stream elements are inherited from the data stream element which has the largest generation timestamp of all data stream elements which have contributed to the data of the produced element. In our generic model we have not specified how the data stream element which has contributed last to the data with respect to the generation timestamps is identified. Since the logic of each worker-specific module is freely-programmable (see Section 8.2.1) and can thus comprise the generation of output stream elements with very diverse semantics and data dependencies, we have refrained from integrating a generic mechanism into the data stream analysis system which deduces this data stream element and thus the correct generation timestamps automatically. In consequence, StreamTeam requires worker developers to specify the generation timestamp inheritance logic for each generated output stream element in the logic of the worker-specific module which generates the new output stream element. The generation timestamp inheritance logic can be very simple but also complex. Usually the generation timestamp is acquired by extracting the generation timestamp from the input stream element for which the single element processor module logic is executed and/or by means of reading a stored generation timestamp from the local state.

As described in Section 6.2.1, Samza uses Kafka's offsets to order all elements which belong to the same data stream partition. Since we have adopted the sequence number concepts of our model from Samza (see Literature Discussion 4.1, Literature Discussion 5.5, and Literature Discussion 6.4), StreamTeam does not modify how Samza uses Kafka's offsets to process the elements of a data stream partition but simply regards the offsets as the sequence numbers of the elements. In consequence, StreamTeam inherits Samza's proxy approach to assign sequence numbers with the Kafka brokers as the communication proxies. Note that, as we have required for the proxy approach in Section 6.2.1.1, Kafka guarantees that only a single Kafka broker assigns sequences numbers to the elements of each data stream partition even if the Kafka log is replicated to multiple brokers [Kaf16][19].

[19] Quote: "Each partition has one server [Kafka broker] which acts as the 'leader' and zero or more servers [Kafka brokers] which act as 'followers'. The leader handles all read and write requests for the partition while the followers passively replicate the leader." [Kaf16]

The processing timestamp creation and assignment is very straightforward. StreamTeam simply acquires the current system time of the Samza task when Samza's `process(`*dse*`, ...)` function is called and assigns this value as the processing timestamp to the input stream element *dse* before the element is handed to the start modules of the single element processor graph (see line 1 of Algorithm 8.2).

Unfortunately, Samza does neither know the concept of entry components nor the concept of ingestion time. Since we demand StreamTeam's data stream analysis system to support all time notions which we have modeled in Chapter 6, we enrich Samza with support for ingestion timestamps. As stated in Section 5.2.2, entry components can in practice be integrated in existing components of the architecture although we have modeled them as separate components. In StreamTeam, we regard the Kafka brokers which receive the raw input stream elements from the raw input stream generating devices, add the elements to their logs, and provide them for Samza tasks and external consumers (see [KNR11; KK15] for more details) as the entry components of the data stream analysis system. Doing so renders the Kafka brokers and thus the communication proxies to be a component of the data stream analysis system.[20] Since each raw input stream element is received by and appended to the log of exactly one Kafka broker which also assigns the sequence number (see above), the requirement of our system model that every raw input stream element is only received by a single entry component (see Section 5.2.2) is fulfilled. Moreover, just as an entry component in our model forwards received raw input stream elements to multiple processors, multiple Samza tasks (and external consumers) can pull raw input stream elements from each Kafka broker. According to our model, the ingestion timestamp of a raw input stream element is created by the local clock of the entry component which receives the element when it receives the element (see Section 6.1.3). In StreamTeam, we leverage Kafka's log append time, i.e., the system time of the Kafka broker when it appends the raw input stream element to the log [Guz19], as the ingestion timestamp of a raw input stream element. Since each raw input stream element is received by and appended to the log of only a single Kafka broker, the ingestion timestamp of a raw input stream

[20] Although the communication proxies are not a part of the data stream analysis system in our model as the entry components are modeled as separate components this is not a conflict between our generic model and our implementation. This is due to the fact that merging a communication proxy and an entry component and thus regarding the communication proxy as a part of the data stream analysis system is equivalent to having a network channel between the communication proxy and the entry component which provides FIFO guarantees and which does not introduce any transmission delay.

element is unambiguous in a single data stream analysis system as defined in our model (see Section 6.1.3). In order to be able to leverage Kafka's log append time as the ingestion timestamp Kafka's message timestamp type has to be set to `LogAppendTime` [Guz19]. Moreover, we had to modify some classes of Samza (version 0.13.1) in order to be able to access the log append time and assign it to the raw input stream element in line 1 of Algorithm 8.2.[21]

Literature Discussion 8.2 Ingestion Timestamps in Samza

The idea to leverage Kafka timestamps in Samza has not only been pursued by us. The latest version of Samza (version 1.4.0) makes also use of the Kafka timestamps.
However, in this version the Kafka timestamp is assigned as the event timestamp which is equivalent to the generation timestamp in our model (see event time discussion Literature Discussion 6.1).[22] We therefore suppose that the developers of Samza assume that Kafka's message timestamp type is set to `CreateTime` and thus to the default value [Guz19; Kaf16].[23]
Moreover, note that the arrival timestamp that is assigned in the latest version of Samza to each message (i.e., data stream element) is also not the ingestion timestamp but the processing timestamp since the arrival timestamp is created in every `KafkaConsumerProxy` instance and thus in every Samza task.
Hence, even the latest version of Samza does not support ingestion timestamps.

As a last point we want to highlight that StreamTeam does not assume that the clocks of the Samza tasks (which implement the processors of our model) and the clocks or the Kafka brokers (which implement the entry components of our

[21] Note that solely the ingestion timestamp assignment but not the ingestion timestamp creation is done in the `process(`*dse*`, ...)` function. If multiple Samza tasks consume the same raw input stream element, it is assigned the same log append time as the ingestion timestamp.

[22] This information can be extracted from the code of Samza version 1.4.0, namely from line 327 to 343 in `KafkaConsumerProxy` (see https://github.com/apache/samza/blob/release-1.4.0-rc1/samza-kafka/src/main/scala/org/apache/samza/system/kafka/KafkaConsumerProxy.java) and from line 76 to 92 in `IncomingMessageEnvelope` (see https://github.com/apache/samza/blob/release-1.4.0-rc1/samza-api/src/main/java/org/apache/samza/system/IncomingMessageEnvelope.java). (Both links were last accessed at 03.04.2020.)

[23] In the comments of Kafka's `ConsumerRecordTimestampExtractor` (version 0.10.1 which we use in StreamTeam) it is even stated that using `CreateTime` leads to event time semantics and that using `LogAppendTime` leads to ingestion time semantics (see https://github.com/apache/kafka/blob/0.10.1/streams/src/main/java/org/apache/kafka/streams/processor/ConsumerRecordTimestampExtractor.java, last accessed at 03.04.2020). However, if event timestamps are generated like this only the event timestamps of raw input stream elements are equivalent to our generation timestamps since the generation timestamps of event, state, and statistics stream elements have to be inherited to meet the semantics of our stream time model.

model) are synchronized – neither with the raw input stream generating devices nor among each other. Hence, as defined in our model, there is not only a single globally consistent generation time space but also one or multiple processing time spaces and one or multiple ingestion time spaces. More precisely, there is one processing time space for each machine on which at least one Samza tasks (or more precisely Samza container, see Section 8.2.5.) is deployed and one ingestion time space for each machine on which at least one Kafka broker that acts as a communication proxy for a raw input stream partition is deployed. Moreover, StreamTeam does not pose any restrictive assumptions on the time spaces. Therefore, StreamTeam supports for instance to implement a real-time football analysis application which analyzes position data streams whose generation timestamps have a different origin as the system time based processing and ingestion timestamps (see Chapter 9).

8.2.3.2 Timestamp Usage

Now that we have described how the different timestamps are created and assigned in StreamTeam, we will present for which purposes the different timestamps should be used in the logic of the modules.

Generation timestamps have two advantages compared to processing timestamps and ingestion timestamps. First, all generation timestamps are from the same globally consistent generation time space and thus comparable (see Section 6.1.1). Second, generation timestamps specify the moment when the raw input was generated, when the atomic event occurred, or to which the non-atomic event update, the state, or the statistic refers and thus the moment in time which is of interest for the most analyses. Hence, we argue that worker developers should use generation timestamps to measure time between data stream elements. For instance, in a football analysis application they should be used to check if two players entered the penalty box at approximately the same time or if enough time has passed since the last set play event stream element has been emitted before a new set play event stream element is emitted (see simultaneousness and sequentiality in Section 6.3).

Even if two raw input stream elements should be compared this should usually not be done by means of comparing their ingestion timestamps. If the two raw input stream elements belong to two different raw input stream partitions, they might be handled by two different Kafka brokers deployed on two different machines. Hence, their ingestion timestamps might specify two moments in time with respect to two different ingestion time spaces and are thus not comparable. Moreover, even if this is not the case (e.g., since both elements belong to the same raw input stream partition), the ingestion timestamps do not specify at which moment in time the input data contained in the elements were generated. Therefore, ingestion timestamps should only be used if really the moments in time at which the elements were received by the Kafka brokers should be compared. So far we have not faced an analysis task in which this semantic is needed but we would not go so far as to state that there are none.

Moreover, we argue against using the local processing timestamps for the most analyses although they are guaranteed to specify moments in time with respect to the same local processing time space if they are accessed in a module. However, we use processing timestamps in our active keys modules to maintain the set of active keys. We do so as we explicitly want to compare the system time at which the last data stream element with a certain key was processed by the Samza task with the current system time of the Samza task. This is meaningful as Samza's `window(...)` function and thus the execution of the active keys window module is triggered periodically by means of the system time of the Samza task.[24]

The consequence of our windowing approach is that all time window based analyses which are performed in modules of the window graph are driven by the system time of the Samza task instead of by the generation timestamps of the processed input stream elements. For instance, in our real-time football analysis system we consider all position stream elements which were processed since the last call of Samza's `window(...)` function and thus whose processing timestamps are in the interval spanned by the system time of the last and the current `window(...)` function call to update the heatmaps for the different time windows (see Section 9.2.1.14).[25] As discussed in Section 6.1.3, doing so does not ensure deterministic analysis results. However, as shown in Example 6.2, also opening and closing time windows on the basis of generation timestamps does not ensure deterministic analysis results since data stream elements might be processed out-of-order with respect to their generation timestamps (see Section 6.2.5 and Section 8.2.3.3). Thus, we argue that our system time based windowing approach is reasonable.

[24] This information can be extracted from the code of Samza version 0.13.1, namely from line 45 in `RunLoop` (see https://github.com/apache/samza/blob/0.13.1/samza-core/src/main/scala/org/apache/samza/container/RunLoop.scala, last accessed at 24.01.2020).

[25] Note that, if the moments in time to which the data in two data stream elements that are assigned to the same window by means of their processing timestamps refer should be compared in a single element processor module of the window graph for analysis purposes, we still argue that this should be done by means of comparing their generation timestamps.

8.2.3.3 Ordering Consistency Guarantees

Finally, we will analyze which ordering consistency guarantees result from the way in which timestamps and sequence numbers are created and assigned in StreamTeam.

As mentioned in Section 8.2.3.1, StreamTeam inherits Samza's proxy approach to assign sequence numbers with the Kafka brokers as the communication proxies. More precisely, Samza and thus also StreamTeam uses Kafka's log offsets as sequence numbers and guarantees to process data stream elements sequentially with respect to them [Sam17d]. Hence, the processing time orderings introduced by the processing timestamps assigned by all Samza tasks are guaranteed to be consistent with the corresponding sequence number ordering as we have also shown in our generic model (see Theorem 6.6).

Moreover, Kafka logs are append-only [KK15] and every Kafka consumer and thus also every Samza task pulls new elements of a data stream partition from a Kafka broker by pulling new parts of the log which it has not seen yet by handing the last consumed offset in the pull request [KNR11]. In addition, each Samza container and thus every Samza task (see Section 8.2.5) has only one Kafka consumer for each input stream partition [Sam17h] and "Kafka guarantees that messages from a single partition are delivered to a consumer in order" [KNR11]. In consequence, each Samza task is guaranteed to receive the elements of every input stream partition from Kafka in offset order and thus in sequence number order. Hence, there is no need for a Samza task to buffer data stream elements and wait for out-of-order arrivals with respect to the sequence numbers. Note that this does not create a conflict between our implementation and our model since we have only defined that every processor is able to buffer incoming elements until it is guaranteed that no more data stream elements which are late with respect to their sequence numbers arrive (see Section 5.3.2). Instead, in Samza and thus in StreamTeam the known time bound for late arrivals (see Literature Discussion 6.5) is simply zero independent of the analysis application.

In StreamTeam, the Kafka brokers are not only used as communication proxies that assign sequence numbers. Instead, they are also regarded as the entry components and Kafka's log append time is leveraged as the ingestion timestamp (Section 8.2.3.1). Hence, the sequence number and the ingestion timestamp of each raw input stream element are created and assigned by the same Kafka broker at the same time. In consequence, for each raw input stream partition the sequence number ordering is guaranteed to be consistent with the ingestion time ordering which is introduced by the ingestion timestamps assigned by the Kafka brokers and thus by the entry components of StreamTeam's data stream analysis system. Moreover, for every raw input stream partition and every Samza task the ingestion time ordering is guaranteed to be consistent with the processing time ordering introduced by the processing timestamps created and assigned by the Samza tasks, since all processing time orderings are guaranteed to be consistent with the corresponding sequence number ordering (see above). Note that these StreamTeam-specific consistency guarantees are not contradictory to the theorems given in Section 6.2.5, namely Theorem 6.10 and Theorem 6.11, since these theorems state only that in our generic model consistency is not guaranteed. However, neither Theorem 6.10 nor Theorem 6.11 state that the orderings are guaranteed to be not consistent. Hence, it is conform to our model that StreamTeam provides stronger consistency guarantees.

In contrast, the data stream elements might still be received and processed by the Samza tasks out-of-order with respect to their generation timestamps. Hence, for every data stream partition and every Samza task the generation time ordering is neither guaranteed to be consistent with the sequence number ordering, nor guaranteed to be consistent with the processing time ordering, nor guaranteed to be consistent with the ingestion time ordering. This is in accordance with the theorems of our generic model, namely Theorem 6.8, Theorem 6.9, and Theorem 6.12.

Table 8.1 summarizes the ordering consistency guarantees offered by StreamTeam and contrasts them to the consistency guarantees of our generic model (see Section 6.2.5). At this point we want to highlight that the fact that the ingestion time ordering is guaranteed to be consistent with the processing time ordering and the sequence number ordering results from StreamTeam's very specific design choices. More precisely, this results from the fact that StreamTeam uses the proxy approach to assign sequence numbers and merges the communication proxy and the entry component into a single component. Although the existence of StreamTeam proofs that this works in practice we argue that it is nevertheless reasonable and worthwhile to do not assume this in our generic model but to discuss the time notions and the ordering consistency guarantees for more diverse settings.

8.2.4 Data Stream Model

In our data stream model, we have defined that the information which is shipped in data stream elements is separated into data that can be encoded in generic at-

Ordering 1	Ordering 2	Guaranteed Consistency in Generic Model	Guaranteed Consistency in StreamTeam
Processing time ordering	Sequence number ordering	Yes	Yes
Processing time ordering	Generation time ordering	No	No
Processing time ordering	Ingestion time ordering	No	Yes
Sequence number ordering	Generation time ordering	No	No
Sequence number ordering	Ingestion time ordering	No	Yes
Generation time ordering	Ingestion time ordering	No	No

Table 8.1 Ordering Consistencies in StreamTeam. The table summarizes the consistency guarantees between the different orderings offered by StreamTeam and contrast them to the consistency guarantees of our generic model.

tributes and data stream specific data which do not fit into the generic attributes but are instead encoded in a payload attribute with a data stream specific schema (see Chapter 4). In a nutshell, StreamTeam implements this model by means of providing a nested Google Protocol Buffer [Goo19] message structure to encode data stream elements and Java wrapper classes to create new data stream elements and access the information shipped in a data stream element. In this section, we will give more details on this approach.

To encode the content of a data stream element StreamTeam defines a generic Protocol Buffer message type (see Figure 8.3(a)). This message type has a field for each of the generic attributes of our data stream element model which is not already covered by the Kafka message metadata. That is, the message type has a field for encoding the generation timestamp (*ts*), a field for encoding the positions tuple (*pos*), a field for encoding the object identifiers tuple (*oids*), a field for encoding the group identifiers tuple (*gids*), a field for encoding the phase (φ), and a field for encoding the event identifier (*eid*). Moreover, to ship the information if a data stream element is atomic or not directly in the Protocol Buffer message that is transferred between components of our infrastructure we have decided to add an additional field for encoding the atomicity flag of the data stream to which the element belongs (*ato*).

In addition to the fields for encoding data that fit into the generic attributes, the generic message type has a dedicated field for encoding the payload (*pd*) of

```
message ImmutableDataStreamElementContent {
      bool atomic = 1;
      string eventIdentifier = 2;
      Phase phase = 3;
      int64 generationTimestamp = 4;
      repeated Position positions = 5;
      repeated string objectIdentifiers = 6;
      repeated string groupIdentifiers = 7;
      google.protobuf.Any payload = 8;

      message Position {
            double x = 1;
            double y = 2;
            double z = 3;
      }

      enum Phase {
            NULL = 0;
            START = 1;
            ACTIVE = 2;
            END = 3;
      }
}
```

(a) Generic Data Stream Element Message Type

```
message SuccessfulPassEventStreamElementPayload {
      double length = 1;
      double velocity = 2;
      double angle = 3;
      string direction = 4;
      int32 packing = 5;
}
```

(b) Successful Pass Event Stream Element Payload Message Type

Figure 8.3 Nested Google Protocol Buffer Message Structure in StreamTeam. (a) shows the definition of StreamTeam's generic data stream element message type and (b) shows the definition of the payload message type for the successful pass event stream.

a data stream element. Since this field is of type `Any`, it can be filled with an arbitrary Protocol Buffer message. Each data stream has its own individual payload message type. This type defines the data stream specific payload schema (*sch*) of the data stream element. For instance, in our real-time football analysis application we specify a payload message type for the successful pass event stream which defines that the payload contains a field for encoding the length of the pass, a field for encoding the velocity of the pass, a field for encoding the angle of the pass, a field for encoding the direction category, and a field for encoding the packing value (see Figure 8.3(b)).

In StreamTeam the information which is shipped in input stream elements can be accessed by means of data stream specific wrapper classes. Each wrapper class instance contains the received Protocol Buffer message and the missing attributes which are part of the Kafka message metadata, namely the key (k) and the sequence number (ξ). Moreover, it contains the name (*name*) and the category (*cat*) of the data stream to which the data stream element belongs. In addition, the instance contains the locally valid processing timestamp (τ), and if the data stream element is a raw input stream element also the ingestion timestamp (T).

When an input stream element is received by a Samza task and thus Samza's `process(`*dse*`,...)` function is called for a new input stream element an instance of the data stream specific wrapper class is created. For doing so, StreamTeam contains an abstract generic wrapper class which provides a function that generates an instance of the correct data stream specific wrapper class when it is called with the received Protocol Buffer message, the key, the sequence number, the processing timestamp, and the ingestion timestamp. This is done in line 1 of Algorithm 8.2 after creating the processing timestamp and extracting the sequence number, the key, and the ingestion timestamp from the Kafka message metadata (see Section 8.2.3.1). Note that this requires that there is an implementation of this abstract generic wrapper class for each data stream.

Moreover, the abstract generic wrapper class contains functions to access data from the generic attributes and the payload. Among others, there is a function to get the data stream name, a function to get the key, a function to get the processing timestamp, a function to get the generation timestamp, a function to get the position at a given index from the positions tuple, and a function to get a field value from the payload. These functions can be used to access all data from a data stream specific wrapper class instance.

However, using these functions in the module logic is cumbersome, especially if data from the payload should be extracted. For instance, extracting the start position of a pass requires specifying the index in the positions tuple and extracting the velocity of a pass from the payload requires specifying the field name as a String and casting the result.[26] To improve the comfort for accessing the data in the module logic, the data stream specific wrapper classes which implement the abstract generic wrapper class can provide additional getter functions with semantically-rich names. For instance, in our real-time football analysis application we implemented a successful pass event stream element wrapper

[26] Specifying the field name as a String and casting the result is necessary since the abstract generic wrapper class does not know the schema of the data stream specific payload.

class which provides, among others, a `getKickPosition()` function which returns the position of the kicking player and thus the position where the pass started from the positions tuple and a `getVelocity()` function which returns the velocity from the payload.

To facilitate generating output stream elements in the module logic, each data stream specific wrapper class should further provide a function to create a new instance that wraps around a generated output stream element. For instance, our successful pass event stream element wrapper class provides a function which has to be called with the match identifier, the generation timestamp, information about the kicking player, information about the receiving player, the length, the velocity, the angle, the direction category, and the packing value as parameters to generate an instance which wraps a new successful pass event stream element. As a matter of course, wrapper class instances for output stream elements do neither contain a sequence number (since the Kafka brokers assign them), nor a processing timestamp, nor an ingestion timestamp.

We want to highlight that StreamTeam's data stream model implementation supports adding new data streams with very little effort. More precisely, to add a new data stream it is sufficient to add a new Protocol Buffer message type for the payload and a data stream specific wrapper class. None of the existing classes or types has to be modified.[27]

As a last point we want to mention that the wrapper classes cannot only be used in the logic of the modules. Instead, they can be used in arbitrary Java programs. For instance, they are used in our MongoDB stream importer (see Section 9.4.2). Moreover, to facilitate accessing the data shipped in a data stream element in the Web clients of the infrastructure (see Section 8.1), StreamTeam provides an automatically generated JavaScript API which enables accessing all data from a data stream element when it is loaded together with protobuf.js [Wir20]. This API is used by our generic cluster monitor (see Section 8.3.2) and by our football-specific user interface (see Section 9.3).

8.2.5 Deployment

In our model, we have defined that all entry components and processors have to be deployed on a set of machines to execute the data stream analysis system (see Section 5.4.1). In StreamTeam there are no separate entry components (see Section 8.2.3.1). Instead, a Kafka broker is deployed on every machine of the

[27] Avoiding the need to modify existing classes or types was the driving force behind our decision to define the payload field to be of type `Any` instead of `oneof`.

cluster that is used to execute the data stream analysis system. Moreover, the components which perform the analysis subtasks defined by the workers are deployed onto the cluster machines.

As described above, each StreamTeam worker is implemented as a Samza job (with our module graph extension) and each Samza job is separated into multiple Samza tasks. However, according to Samza's design, these Samza tasks are not deployed directly [Sam17d]. Instead, multiple Samza tasks are grouped to a Samza container which executes all Samza tasks that are assigned to it [Sam17d; Sam17h] and which is deployed as a YARN container [VMD⁺13] on a machine of the cluster which runs a YARN node manager.

Remember that in StreamTeam the number of partitions is defined globally for the streaming layer and that thus the number of Samza tasks is the same for every worker (see Section 8.2.2). Instead, the number of containers can be configured individually for each Samza job by setting `job.container.count` in its configuration file and thus for each worker [Sam17h; Sam17e]. Hence, the additional container layer enables deploying different StreamTeam workers onto a different numbers of machines.[28] This is a useful feature since some workers specify only simple analysis subtasks that do not require much computational resources while other workers specify compute-intensive analysis subtasks and thus require that their Samza tasks are distributed onto multiple machines. Moreover, Samza supports changing the number of containers while the number of partitions and thus the number of tasks is fixed [Sam17d]. Thus, the additional container layer enables adapting the parallelism degree flexibly to the current workload [Sam17h]. Although we have not used this feature in StreamTeam yet, we argue that it is very useful for applications, such as fake news analysis, in which the volume of streamed data which has to be analyzed varies over time (e.g., peaks at elections).

We want to highlight that, although multiple Samza tasks are executed in the context of the same Samza container, they perform their analysis independently. For instance, the state is not shared between Samza tasks which are executed by the same Samza container [Sam17h]. In consequence, the additional container layer is only a deployment detail which does not have to be regarded when developing workers or more precisely when developing the logic of the modules.

[28] Although Kafka would support specifying the number of partitions individually for each data stream (see Footnote 7 in Section 8.2.2), using this to control the number of Samza tasks for the workers individually is not always possible since the same data stream might be an input stream of multiple workers and the number of Samza tasks for a worker is determined by "the maximum number of partitions across all input streams" [Sam17h] of the worker.

Example 8.1 Deployment of StreamTeam's Data Stream Analysis System

In order to illustrate the deployment of StreamTeam's data stream analysis system assume that there is an analysis workflow consisting of three workers (w_1, w_2, and w_3). Under the assumption that the global number of partitions per data stream (`num.partitions`) is set to six, there are six Samza tasks for each worker, namely $Task_{1A}$ to $Task_{1F}$ for w_1, $Task_{2A}$ to $Task_{2F}$ for w_2, and $Task_{3A}$ to $Task_{3F}$ for w_3.
Assume that w_1 is configured to have one container ($Container_{1X}$), w_2 is configured to have three containers ($Container_{2X}$, $Container_{2Y}$, and $Container_{2Z}$), and w_3 is configured to have two containers ($Container_{3X}$ and $Container_{3Y}$). All Samza tasks for w_1 (i.e., $Task_{1A}$ to $Task_{1F}$) are assigned to $Container_{1X}$, $Task_{2A}$ and $Task_{2B}$ are assigned to $Container_{2X}$, $Task_{2C}$ and $Task_{2D}$ are assigned to $Container_{2Y}$, $Task_{2E}$ and $Task_{2F}$ are assigned to $Container_{2Z}$, $Task_{3A}$, $Task_{3C}$, and $Task_{3E}$ are assigned to $Container_{3X}$, and $Task_{3B}$, $Task_{3D}$, and $Task_{3F}$ are assigned to $Container_{3Y}$.
Figure 8.4 shows a sample deployment of the Samza containers and the Kafka brokers onto a cluster consisting of four (homogeneous or heterogeneous) machines, one master node which runs a YARN resource manager and three processing nodes which run a YARN node manager. The first Kafka broker is deployed on the master node. $Container_{3Y}$ and a second Kafka broker are deployed on the first processing node. $Container_{1X}$, $Container_{2X}$, $Container_{2Y}$, and a third Kafka broker are deployed on the second processing node. The third processing node hosts $Container_{2Z}$, $Container_{3X}$, and another Kafka broker.

8.3 Kafka REST Proxy and Cluster Monitor

As mentioned in Section 8.1, StreamTeam supports multiple diverse generic and application-specific Web clients which implement user interfaces that serve different purposes. In this section, we will present how StreamTeam's Kafka REST proxy enables Web clients to consume data stream elements via its REST API. Moreover, we will present the cluster monitor, a generic Web client which is shipped as a part of the StreamTeam infrastructure.

8.3.1 Kafka REST Proxy

The Kafka REST proxy is a component of the StreamTeam infrastructure that serves as a proxy between the Kafka brokers and Web clients, which are not able

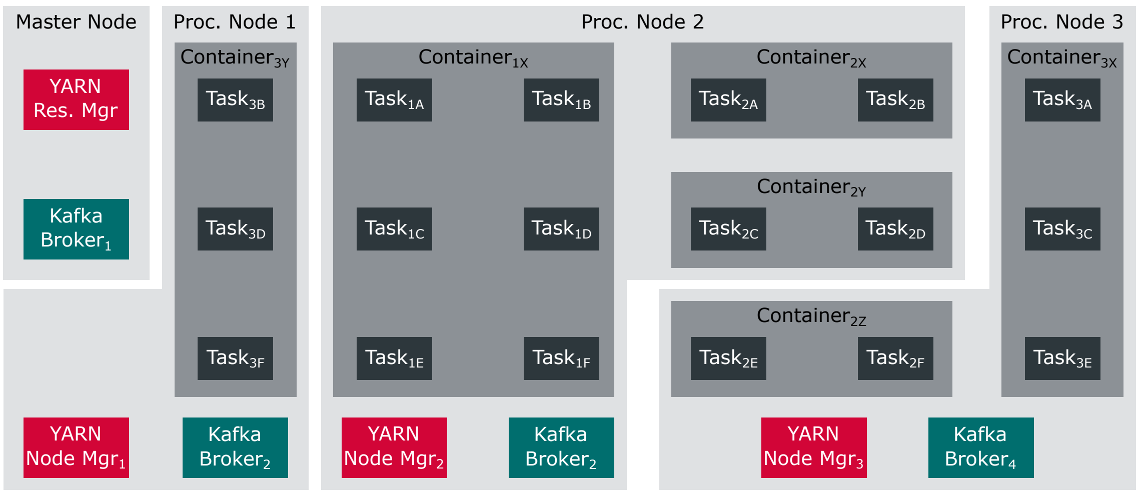


Figure 8.4 **Deployment of StreamTeam's Data Stream Analysis System.** The dark-gray boxes illustrate the Samza tasks and the gray boxes illustrate the containers to which they are grouped. The mint boxes illustrate Kafka brokers. The red boxes illustrate the YARN resource manager and the YARN node managers. The light-gray boxes visualize the cluster machines on which the Samza containers and the Kafka brokers are deployed.

to pull data stream elements directly from the Kafka brokers (see Section 8.1). Earlier versions of the Kafka REST proxy have been briefly presented together with earlier versions of StreamTeam's data stream analysis system prototype in [PBS+17] and [PRS+18].

The Kafka REST proxy is a Kafka consumer that uses the Consumer API of Kafka's original Java library [Kaf16] to pull all new raw input, event, state, and statistics stream elements periodically from the Kafka brokers. Moreover, it buffers the latest elements of each data stream partition in its local memory. The pull interval and the number of elements which are buffered for each data stream partition are configurable. In contrast, the list of data streams for which the Kafka REST proxy pulls new elements does not have to be provided in the configuration. Instead, the Kafka REST proxy builds and updates this list automatically at runtime. More precisely, the Kafka REST proxy checks periodically in a configurable interval if the Kafka brokers provide elements of a new raw input, event, state, or statistics stream whose elements the Kafka REST proxy does not consume yet. If new data streams are found, it starts consuming the elements of these data streams. The huge benefit of this is that the Kafka REST proxy does not have to be reconfigured or restarted if a new worker with new output streams is added to the data stream analysis system.

All buffered data stream elements are provided via a REST API to Web clients. For this purpose, the Kafka REST proxy runs a Jetty [Ecl20] Web server

Target	Parameter	Description
consume	t, k, l	Retrieves the latest l buffered elements of the data stream partition $\langle ds, k \rangle$ with $ds.name = t$
consume	t, l	Retrieves the latest l buffered elements of the data stream ds with $ds.name = t$
listKeys	t	Retrieves a list containing all keys for which at least one element belonging to the data stream ds with $ds.name = t$ is buffered
listTopics*	–	Retrieves a list containing the names of all data streams for which at least one element is buffered

*The target is called listTopics instead of listDataStreamNames since we adopted Kafka's terminology in the REST API.

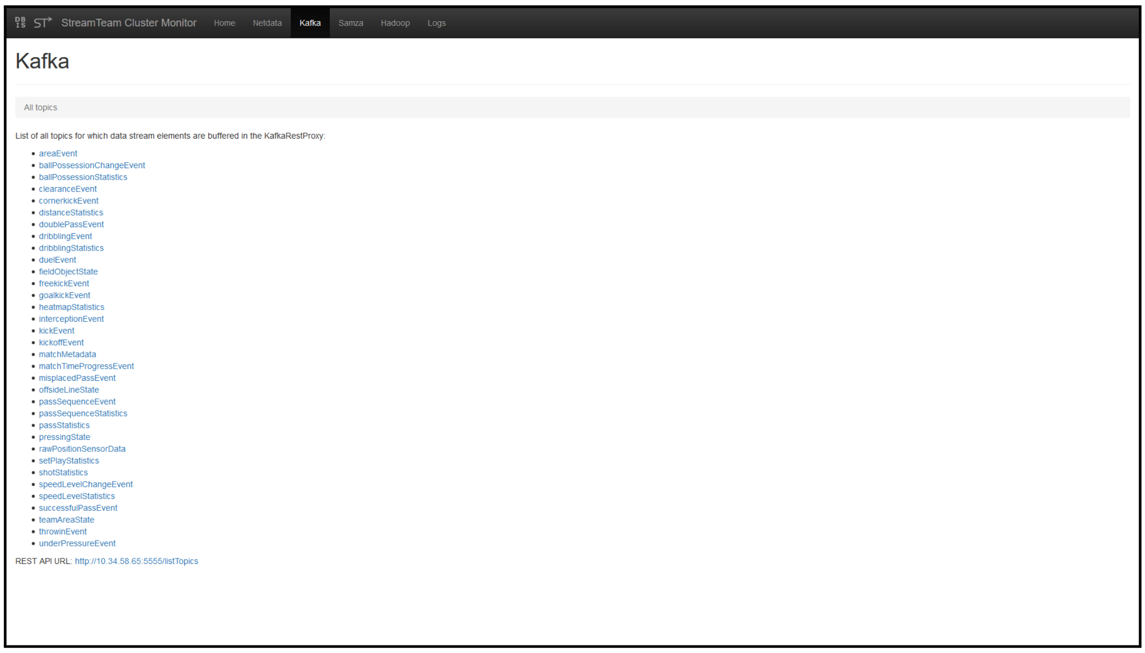
Table 8.2 Kafka REST Proxy REST API. The table lists and describes the queries which the REST API of the Kafka REST proxy supports.

that handles Hypertext Transfer Protocol (HTTP) requests which are structured according to our REST API (see Table 8.2). Our REST API supports retrieving elements of a data stream partition or a whole data stream, retrieving a list containing all keys for which at least one element belonging to a certain data stream is buffered, and retrieving a list containing the names of all data streams for which at least one element is buffered. The result which the Kafka REST proxy returns when an HTTP request was handled is an easy-to-parse JSON document. If a result contains data stream elements, the nested Protocol Buffer message of each data stream element (see Section 8.2.4) is Base64 encoded [Jos06] using the Base64 encoder of Apache Commons Codec [Com20]. This is done to prevent conflicts with the JSON syntax. Note that Web clients can access all information from the Base64 encoded nested Protocol Buffer message using the JavaScript API which StreamTeam provides.

8.3.2 Cluster Monitor

The cluster monitor is an application-independent Web client which serves as a generic administration and debugging tool. It is implemented in HTML, CSS, PHP, and JavaScript with the assistance of Bootstrap [Boo20] and jQuery [jQu20a].

Among other features, the cluster monitor enables inspecting the results of queries that are issued to the REST API of the Kafka REST proxy. More precisely, the cluster monitor lists the names of all data streams for which at least one element is buffered (see Figure 8.5), all keys for which at least one element belonging to a certain data stream is buffered (see Figure 8.6), and the latest



Kafka

All topics

List of all topics for which data stream elements are buffered in the KafkaRestProxy:

- areaEvent
- ballPossessionChangeEvent
- ballPossessionStatistics
- clearanceEvent
- cornerkickEvent
- distanceStatistics
- doublePassEvent
- dribblingEvent
- dribblingStatistics
- duelEvent
- fieldObjectState
- freekickEvent
- goalkickEvent
- heatmapStatistics
- interceptionEvent
- kickEvent
- kickoffEvent
- matchMetadata
- matchTimeProgressEvent
- misplacedPassEvent
- offsideLineState
- passSequenceEvent
- passSequenceStatistics
- passStatistics
- pressingState
- rawPositionSensorData
- setPlayStatistics
- shotStatistics
- speedLevelChangeEvent
- speedLevelStatistics
- successfulPassEvent
- teamAreaState
- throwinEvent
- underPressureEvent

REST API URL: http://10.34.58.65:5555/listTopics

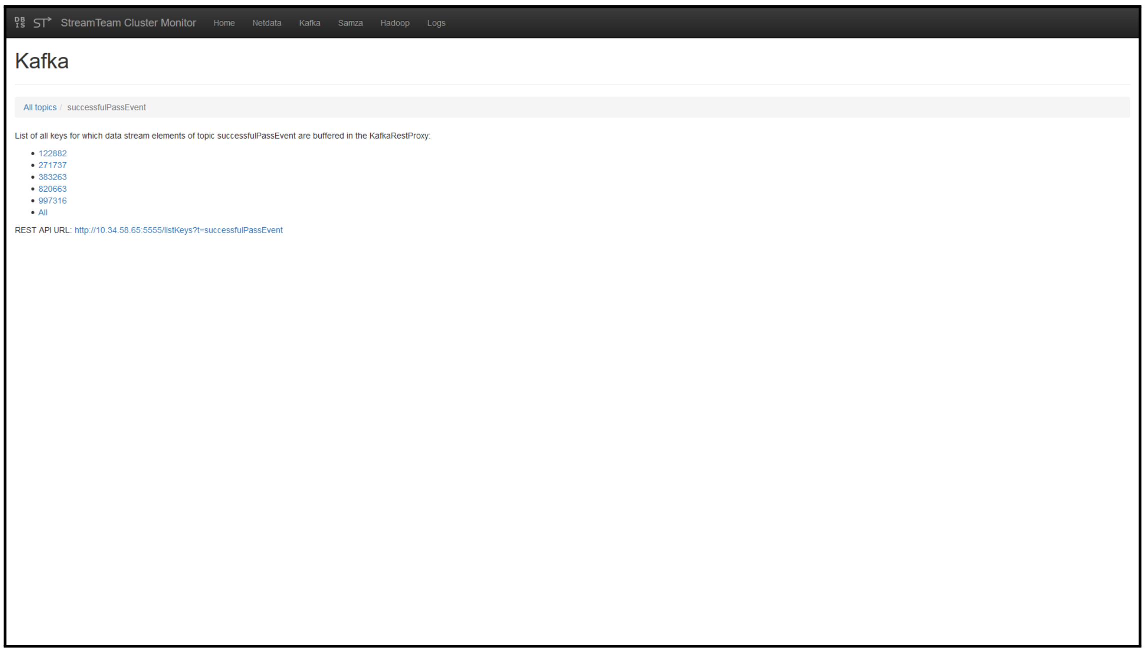
Figure 8.5 Data Stream Name List in Cluster Monitor. The screenshot shows the page of the cluster monitor which lists the names of all data streams for which at least one element is buffered by the Kafka REST proxy. The screenshot was taken when running StreamTeam-Football (see Chapter 9).

200 buffered elements of a certain data stream partition (see Figure 8.7). In the data stream element list the content of each data stream element is decoded with protobuf.js [Wir20] and StreamTeam's JavaScript API (see Section 8.2.4). Since the Kafka REST proxy buffers the elements of all raw input, event, state, and statistics streams automatically, these lists are a great tool to debug the implementation of worker modules.

Moreover, the cluster monitor provides a list of all Samza jobs and thus StreamTeam workers which are deployed on YARN (see Figure 8.8). This list enables checking if all StreamTeam workers started properly and are still running and accessing the corresponding page in the YARN Web interfaces and thus also the Samza logs. In addition, there are "kill"-buttons to stop single workers or the complete data stream analysis system.

Furthermore, the cluster monitor provides selected performance metrics captured and visualized with Netdata [Net20] (see Figure 8.9) as well as lists with links to the log pages, the YARN Web interfaces, and the HDFS[29] Web interfaces of all cluster machines.

[29] HDFS [Had19] is only used to store the JAR file that contains the specifications of the StreamTeam workers which form the analysis workflow but not to store or read any other data.



Kafka

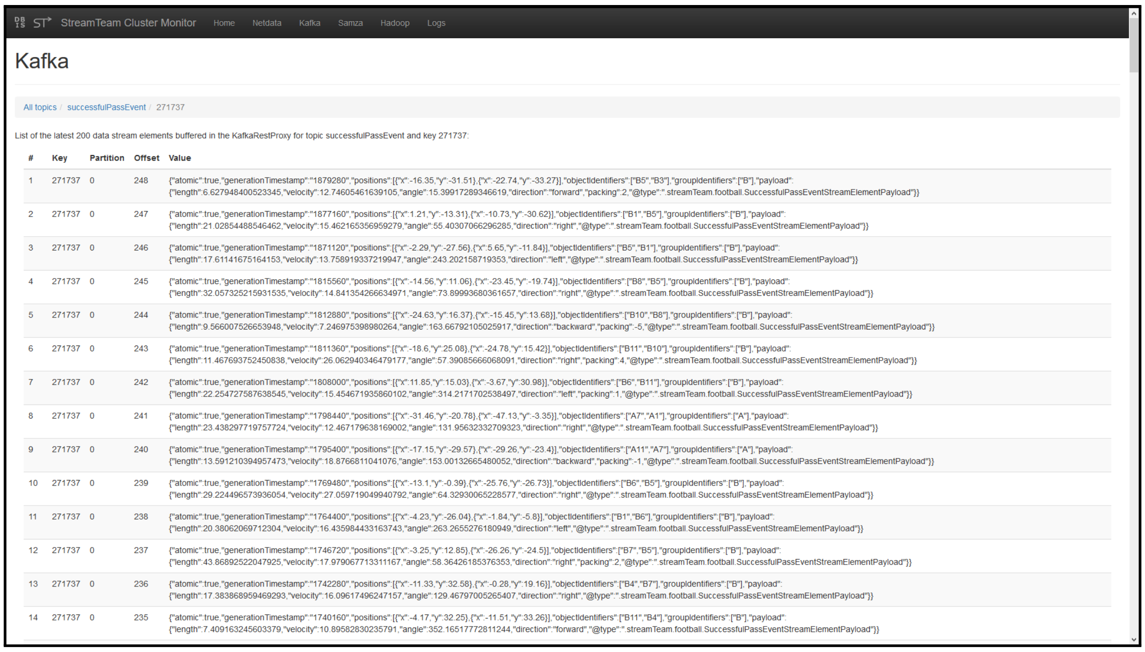
All topics / successfulPassEvent

List of all keys for which data stream elements of topic successfulPassEvent are buffered in the KafkaRestProxy:

- 122882
- 271737
- 383263
- 820663
- 997316
- All

REST API URL: http://10.34.58.65:5555/listKeys?t=successfulPassEvent

Figure 8.6 Key List in Cluster Monitor. The screenshot shows the page of the cluster monitor which lists all keys for which at least one element belonging to the successful pass event stream is buffered by the Kafka REST proxy. The screenshot was taken when running StreamTeam-Football (see Chapter 9). "All" links to the data stream element list for the whole data stream.



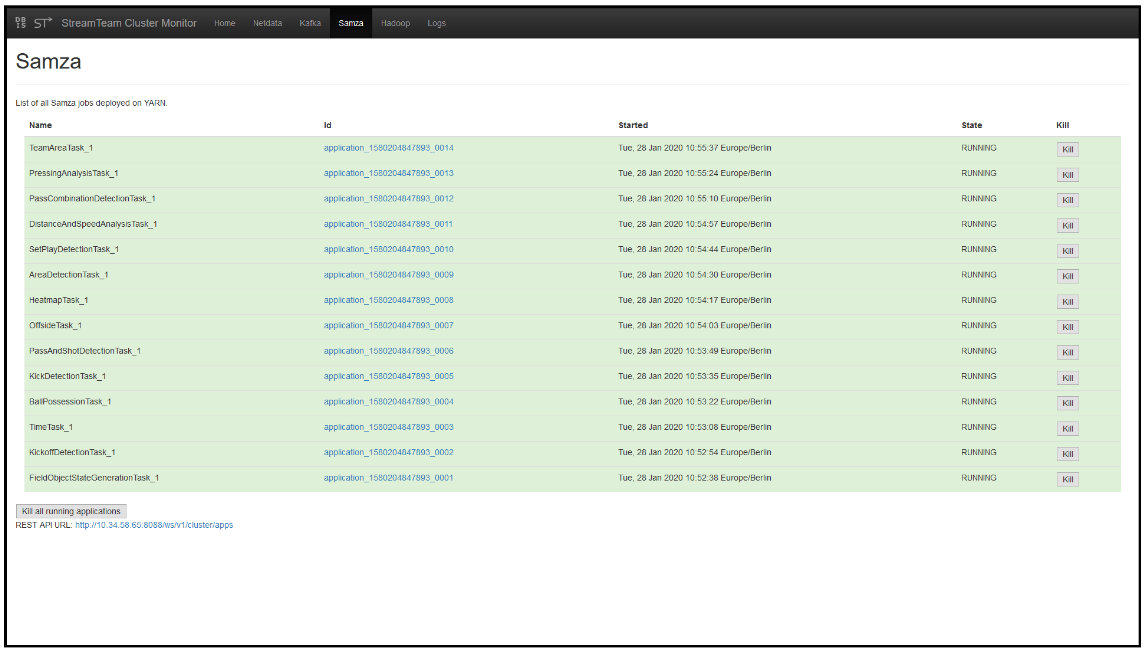
Kafka

All topics / successfulPassEvent / 271737

List of the latest 200 data stream elements buffered in the KafkaRestProxy for topic successfulPassEvent and key 271737:

#	Key	Partition	Offset	Value
1	271737	0	248	{"atomic":true,"generationTimestamp":"1879280","positions":[{"x":-16.35,"y":-31.51},{"x":-22.74,"y":-33.27}],"objectIdentifiers":["B5","B3"],"groupIdentifiers":["B"],"payload": {"length":6.627948400523345,"velocity":12.74605461639105,"angle":15.39917289346619,"direction":"forward","packing":2,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
2	271737	0	247	{"atomic":true,"generationTimestamp":"1877160","positions":[{"x":1.21,"y":-13.31},{"x":-10.73,"y":-30.62}],"objectIdentifiers":["B1","B5"],"groupIdentifiers":["B"],"payload": {"length":21.02854488546462,"velocity":15.462165356959279,"angle":55.40307066296285,"direction":"right","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
3	271737	0	246	{"atomic":true,"generationTimestamp":"1871120","positions":[{"x":-2.29,"y":-27.56},{"x":5.65,"y":-11.84}],"objectIdentifiers":["B5","B1"],"groupIdentifiers":["B"],"payload": {"length":17.61141675164153,"velocity":13.758919337219947,"angle":243.202158719353,"direction":"left","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
4	271737	0	245	{"atomic":true,"generationTimestamp":"1815560","positions":[{"x":-14.56,"y":11.06},{"x":-23.45,"y":-19.74}],"objectIdentifiers":["B8","B5"],"groupIdentifiers":["B"],"payload": {"length":32.057325215931535,"velocity":14.841354266634971,"angle":73.89993680361657,"direction":"right","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
5	271737	0	244	{"atomic":true,"generationTimestamp":"1812880","positions":[{"x":-24.63,"y":16.37},{"x":-15.45,"y":13.68}],"objectIdentifiers":["B10","B8"],"groupIdentifiers":["B"],"payload": {"length":9.566007526653948,"velocity":7.246975398980264,"angle":163.66792105025917,"direction":"backward","packing":-5,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
6	271737	0	243	{"atomic":true,"generationTimestamp":"1811360","positions":[{"x":-18.6,"y":25.08},{"x":-24.78,"y":15.42}],"objectIdentifiers":["B11","B10"],"groupIdentifiers":["B"],"payload": {"length":11.467693752450838,"velocity":26.062940346479177,"angle":57.39085666068091,"direction":"right","packing":4,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
7	271737	0	242	{"atomic":true,"generationTimestamp":"1808000","positions":[{"x":11.85,"y":15.03},{"x":-3.67,"y":30.98}],"objectIdentifiers":["B6","B11"],"groupIdentifiers":["B"],"payload": {"length":22.254727587638545,"velocity":15.454671935860102,"angle":314.2171702538497,"direction":"left","packing":1,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
8	271737	0	241	{"atomic":true,"generationTimestamp":"1798440","positions":[{"x":-31.46,"y":-20.78},{"x":-47.13,"y":-3.35}],"objectIdentifiers":["A7","A1"],"groupIdentifiers":["A"],"payload": {"length":23.438297719757724,"velocity":12.467179638169002,"angle":131.95632332709323,"direction":"right","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
9	271737	0	240	{"atomic":true,"generationTimestamp":"1795400","positions":[{"x":-17.15,"y":-29.57},{"x":-29.26,"y":-23.4}],"objectIdentifiers":["A11","A7"],"groupIdentifiers":["A"],"payload": {"length":13.591210394957473,"velocity":18.8766811041076,"angle":153.00132665480052,"direction":"backward","packing":-1,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
10	271737	0	239	{"atomic":true,"generationTimestamp":"1769480","positions":[{"x":-13.1,"y":-0.39},{"x":-25.76,"y":-26.73}],"objectIdentifiers":["B6","B5"],"groupIdentifiers":["B"],"payload": {"length":29.224496573936054,"velocity":27.059719049940792,"angle":64.32930065228577,"direction":"right","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
11	271737	0	238	{"atomic":true,"generationTimestamp":"1764400","positions":[{"x":-4.23,"y":-26.04},{"x":-1.84,"y":-5.8}],"objectIdentifiers":["B1","B6"],"groupIdentifiers":["B"],"payload": {"length":20.38062069712304,"velocity":16.435984433163743,"angle":263.2655276180949,"direction":"left","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
12	271737	0	237	{"atomic":true,"generationTimestamp":"1746720","positions":[{"x":-3.25,"y":12.85},{"x":-26.26,"y":-24.5}],"objectIdentifiers":["B7","B5"],"groupIdentifiers":["B"],"payload": {"length":43.86892522047925,"velocity":17.979067713311167,"angle":58.36426185376353,"direction":"right","packing":2,"@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
13	271737	0	236	{"atomic":true,"generationTimestamp":"1742280","positions":[{"x":-11.33,"y":32.58},{"x":-0.28,"y":19.16}],"objectIdentifiers":["B4","B7"],"groupIdentifiers":["B"],"payload": {"length":17.383868959469293,"velocity":16.09617496247157,"angle":129.46797005265407,"direction":"right","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
14	271737	0	235	{"atomic":true,"generationTimestamp":"1740160","positions":[{"x":-4.17,"y":32.25},{"x":-11.51,"y":33.26}],"objectIdentifiers":["B11","B4"],"groupIdentifiers":["B"],"payload": {"length":7.409163245603379,"velocity":10.89582830235791,"angle":352.16517772811244,"direction":"forward","@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}

Figure 8.7 Data Stream Element List in Cluster Monitor. The screenshot shows the page of the cluster monitor which lists the latest 200 elements belonging to the successful pass event stream and having key 271737 that are buffered by the Kafka REST proxy. The screenshot was taken when running StreamTeam-Football (see Chapter 9).



Samza

List of all Samza jobs deployed on YARN.

Name	Id	Started	State	Kill
TeamAreaTask_1	application_1580204847893_0014	Tue, 28 Jan 2020 10:55:37 Europe/Berlin	RUNNING	Kill
PressingAnalysisTask_1	application_1580204847893_0013	Tue, 28 Jan 2020 10:55:24 Europe/Berlin	RUNNING	Kill
PassCombinationDetectionTask_1	application_1580204847893_0012	Tue, 28 Jan 2020 10:55:10 Europe/Berlin	RUNNING	Kill
DistanceAndSpeedAnalysisTask_1	application_1580204847893_0011	Tue, 28 Jan 2020 10:54:57 Europe/Berlin	RUNNING	Kill
SetPlayDetectionTask_1	application_1580204847893_0010	Tue, 28 Jan 2020 10:54:44 Europe/Berlin	RUNNING	Kill
AreaDetectionTask_1	application_1580204847893_0009	Tue, 28 Jan 2020 10:54:30 Europe/Berlin	RUNNING	Kill
HeatmapTask_1	application_1580204847893_0008	Tue, 28 Jan 2020 10:54:17 Europe/Berlin	RUNNING	Kill
OffsideTask_1	application_1580204847893_0007	Tue, 28 Jan 2020 10:54:03 Europe/Berlin	RUNNING	Kill
PassAndShotDetectionTask_1	application_1580204847893_0006	Tue, 28 Jan 2020 10:53:49 Europe/Berlin	RUNNING	Kill
KickDetectionTask_1	application_1580204847893_0005	Tue, 28 Jan 2020 10:53:35 Europe/Berlin	RUNNING	Kill
BallPossessionTask_1	application_1580204847893_0004	Tue, 28 Jan 2020 10:53:22 Europe/Berlin	RUNNING	Kill
TimeTask_1	application_1580204847893_0003	Tue, 28 Jan 2020 10:53:08 Europe/Berlin	RUNNING	Kill
KickoffDetectionTask_1	application_1580204847893_0002	Tue, 28 Jan 2020 10:52:54 Europe/Berlin	RUNNING	Kill
FieldObjectStateGenerationTask_1	application_1580204847893_0001	Tue, 28 Jan 2020 10:52:38 Europe/Berlin	RUNNING	Kill

Kill all running applications

REST API URL: http://10.34.58.65:8088/ws/v1/cluster/apps

Figure 8.8 Samza Job List in Cluster Monitor. The screenshot shows the page of the cluster monitor which lists all Samza jobs and thus StreamTeam workers which are deployed on YARN. The screenshot was taken when running StreamTeam-Football (see Chapter 9).

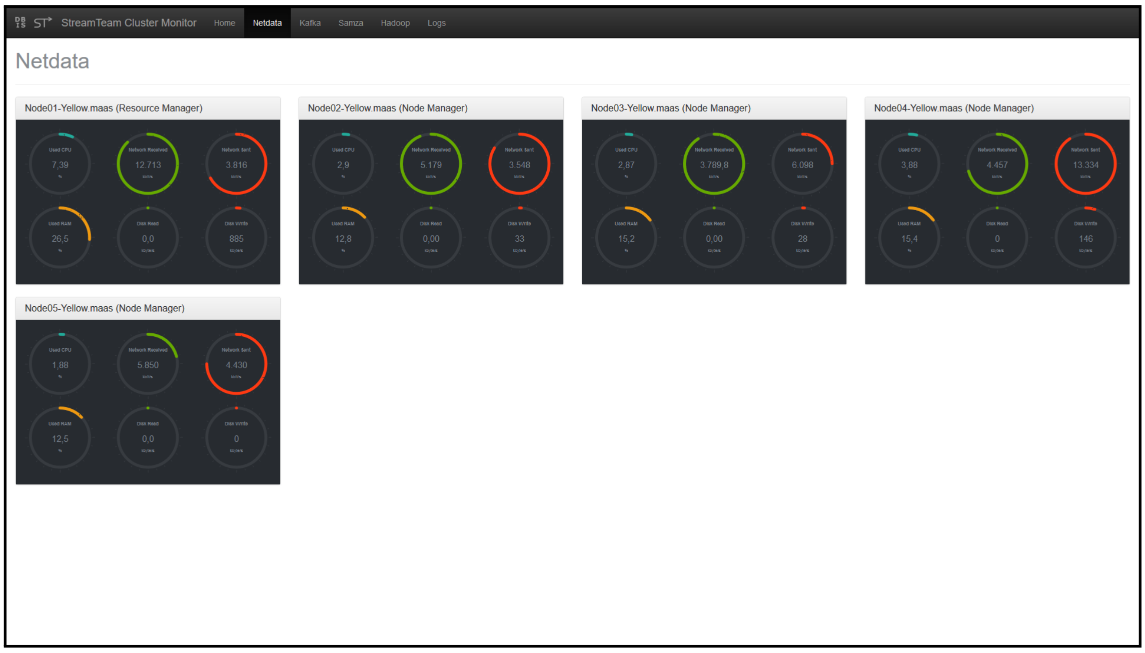

StreamTeam Cluster Monitor
Home
Netdata
Kafka
Samza
Hadoop
Logs
Netdata
Node01-Yellow.maas (Resource Manager)
Used CPU 7,39 %
Network Received 12.713 kbit/s
Network Sent 3.816 kbit/s
Used RAM 26,5 %
Disk Read 0,0 kbyte/s
Disk Write 885 kbyte/s
Node02-Yellow.maas (Node Manager)
Used CPU 2,9 %
Network Received 5.179 kbit/s
Network Sent 3.548 kbit/s
Used RAM 12,8 %
Disk Read 0,00 kbyte/s
Disk Write 33 kbyte/s
Node03-Yellow.maas (Node Manager)
Used CPU 2,87 %
Network Received 3.789,8 kbit/s
Network Sent 6.098 kbit/s
Used RAM 15,2 %
Disk Read 0,00 kbyte/s
Disk Write 28 kbyte/s
Node04-Yellow.maas (Node Manager)
Used CPU 3,88 %
Network Received 4.457 kbit/s
Network Sent 13.334 kbit/s
Used RAM 15,4 %
Disk Read 0 kbyte/s
Disk Write 146 kbyte/s
Node05-Yellow.maas (Node Manager)
Used CPU 1,88 %
Network Received 5.850 kbit/s
Network Sent 4.430 kbit/s
Used RAM 12,5 %
Disk Read 0,0 kbyte/s
Disk Write 0 kbyte/s

Figure 8.9 Machine Performance Metrics in Cluster Monitor. The screenshot shows the page of the cluster monitor which presents selected performance metrics captured and visualized with Netdata. The screenshot was taken when running StreamTeam-Football (see Chapter 9).

9

StreamTeam-Football

In Chapter 8, we have only presented the generic StreamTeam infrastructure and thus the generic part of our implementation. In this chapter, we will present StreamTeam-Football, the real-time football analysis application we have implemented on top of our generic StreamTeam infrastructure. More precisely, we will first describe in Section 9.1 how we replay football matches to generate input stream elements for StreamTeam-Football's analysis workflow which we will present in Section 9.2. Subsequently, Section 9.3 presents how StreamTeam-Football's analysis results are visualized in a real-time user interface. Finally, we will present in Section 9.4 how the analysis results can be stored persistently for offline activities.

9.1 Sensor Simulator

Ideally, we could deploy a sensor-based and/or video-based tracking system as described in Section 2.1 and use the tracked player and ball positions of a real ongoing football match to test, debug, and evaluate our analysis infrastructure. If the components of the tracking system were able to create raw input stream elements containing the positions and to push these elements to Kafka brokers, these components would be the raw input stream generating devices. Otherwise, we could add an auxiliary component which receives or pulls the positions from the tracking system, creates proper raw input stream elements, and pushes them to the Kafka brokers as the raw input stream generating device (see Section 8.1).

Although we have a cooperation with the Swiss Federal Institute of Sport

Magglingen (SFISM)[1] which has equipped a football field with a combination of the LPM system [SPF04] and the corresponding ball tracking system [Inm20], and although we plan to use StreamTeam-Football to analyze live matches on this field in the future, we argue that using this deployment is not a viable input data generating approach for developing and evaluating StreamTeam-Football. This is due to two reasons. First, the tracked positions in general and the tracked ball positions in particular contain too many errors due to misdetections and require thus a manual post-match cleaning process.[2] Second, there is not always a currently ongoing match which can be tracked.

To eliminate these issues we have implemented sensor simulators which pretend to be sensor devices that measure the player and ball positions of a currently ongoing football match by means of replaying a complete and potentially manually corrected position dataset. Originally, we have presented the sensor simulator concept in [Pro14]. The first version of our sensor simulator implementation which did not push raw input stream elements to Kafka brokers yet was used to test, debug, and evaluate PAN [Pro14; PGS16a; PGS16b; Bri16], an earlier worker-based data stream analysis system developed in our research group. Later versions have been used to continuously test and debug improvements of our generic real-time data stream analysis infrastructure (StreamTeam) and our football-specific analysis application (StreamTeam-Football), to demonstrate an earlier version of StreamTeam-Football at an international conference [PBS+17], to generate data for SportSense [PRS+18], an offline team sports video retrieval system developed in our research group (see Section 9.4.3), and to generate data for an elementary StreamTeam workflow that analyzes Defense of the Ancients 2 (DotA2) matches [Zum19]. The current version which is published on GitHub (see Appendix B) and used in the evaluation that will be presented in Chapter 10 is able to emit raw position sensor data stream elements, i.e., elements of a dedicated raw input stream which are structured according to StreamTeam's data stream model implementation (see Section 8.2.4), for given ball-enriched LPM datasets [SPF04; Inm20] from the SFISM and arbitrary TRACAB Optical Tracking datasets [Chy20c] and we plan to extend it to support datasets from other vendors in the future.

To replay a position dataset, we do not run only one sensor simulator but a

[1] SFISM: https://www.ehsm.admin.ch/en/home.html (Last accessed: 29.01.2020)

[2] Note that position-based analyses as those performed in StreamTeam-Football can only generate meaningful results if the positions have a sufficient quality. Moreover, note that improving the quality of tracking data is orthogonal to our research. Therefore, StreamTeam-Football assumes that the positions contained in the raw input stream elements have a high quality.

separate sensor simulator for each player and the ball. In consequence, there are multiple sensor simulators, each emitting the raw position sensor data stream elements of one player or the ball. The easiest setup which we also use in our evaluation (see Section 10.1.1) is to deploy and execute all sensor simulators on the same dedicated[3] machine. In this case, this machine is the only raw input stream generating device which emits the raw position sensor data stream elements emitted by all sensor simulators. This is equivalent to having a centralized video-based tracking system. However, as shown in [Pro14], it is also possible to deploy the sensor simulators on different machines and thus having multiple raw input stream generating devices if the clocks of these machines are synchronized.[4] Doing so is equivalent to equipping every player and the ball with its own clock-synchronized tracking sensor.

We want to highlight that the clocks of the machines which host sensor simulators are not required to be synchronized to generate proper generation timestamps. In fact, the sensor simulators do not create the generation timestamps themselves but read them from the position datasets. Hence, StreamTeam-Football assumes that the timestamps in these datasets are generated using synchronized clocks. This is in accordance with our model (see Section 6.1.1). However, the synchronized sensor simulator clocks are required to emit each raw position sensor data stream element at the correct time and thus to create the impression that the raw position sensor data stream elements were generated and emitted by a sensor-based and/or video-based tracking system which tracks the positions of an ongoing football match.

For properly simulating a whole match it is important that all sensor simulators start replaying their part of the position dataset and thus enter their simulation loop at approximately the same time. This is achieved by setting a desired UNIX timestamp which is far enough in the future when starting all sensor simulators and waiting in each sensor simulator until the local system time is greater than or equal to this timestamp before entering the simulation loop. In its simulation loop each sensor simulator regularly (in a configurable interval)

[3] According to Section 5.4.1, each raw input stream generating device is a dedicated machine itself which does not host any component of the data stream analysis system.

[4] Earlier versions of the sensor simulator have shipped WeakTrueTime [Pro14] to create timestamps using a virtual synchronized clock. We have removed this feature as it was not used since years. However, the clocks of the machines can be synchronized using different approaches (see Section 6.1.1) and WeakTrueTime can be easily re-added if necessary.

converts the current system timestamp into the generation time space[5], checks if there are not yet emitted positions in the position dataset for which the sensor simulator is responsible and whose timestamp is smaller than or equal to the current generation timestamp, creates a proper raw position sensor data stream element for each of these positions, and pushes these elements to the Kafka brokers by means of the Producer API of Kafka's original Java library [Kaf16]. In order to ensure that raw input stream elements with a certain key are processed by the same Samza tasks as the event, state, and statistics stream elements with the same key the sensor simulators adopt Samza's hash-based partitioning function (see Footnote 8 in Section 8.2.2) to map keys to partition numbers when pushing data stream elements to the Kafka brokers.

In addition to the raw position sensor data stream elements, one arbitrarily chosen sensor simulator further generates and emits a dedicated match metadata stream element before waiting for entering the simulation loop.[6] This raw input stream element contains some metadata about the match which are required for the analysis (see Section 9.2), for the real-time visualization (see Section 9.3), and for the offline video retrieval (see Section 9.4.3). For instance, properly detecting misplaced passes requires that the payload of the match metadata stream element contains the unstandardized length and width of the football field. Moreover, the payload of the match metadata stream element should contain the team colors for visualization purposes and a path to a video of the match for playing scenes.

[5] Under the assumption that the generation timestamps and the system timestamps have the same resolution, the current system timestamp can be converted into the generation time space by means of adding the difference between the current system timestamp and the system timestamp at which the simulation loop was entered to the generation timestamp at which the match started.

[6] The generation timestamp of this match metadata stream element is set to be the smallest timestamp in the positions dataset and thus equal to the generation timestamp of the first raw position sensor data stream element although it is sent some seconds earlier than the first raw position sensor data stream element. Note that this is not conflicting with our model since the same would result from having a very small transmission delay between the raw input stream generating device and the entry components when the match metadata stream element is sent and a much higher transmission delay when the raw position sensor data stream elements are sent. Moreover, doing so guarantees a consistent generation time space since the generation timestamps of all raw input stream elements are taken from the position datasets which we assume to be created by tracking system devices with synchronized clocks (see above).

9.2 Analysis Workflow

StreamTeam-Football comprises StreamTeam workers (i.e., configuration files, module graph specifications, and the logic of worker-specific modules) which form a workflow that can be used to analyze a football match stepwise in real-time on the basis of some match metadata and a continuous stream of player and ball positions.

Earlier versions of StreamTeam-Football which consisted of less workers and performed less analyses than the current version have been presented in [PBS+17], [PRS+18], and [SRP+19]. However, [PBS+17] and [PRS+18] have presented the analysis workflow only very briefly and [SRP+19] has only presented a small subset of the workflow in more detail. Moreover, before we have implemented the first version of StreamTeam we have already started analyzing football matches in real-time in PAN [Pro14; PGS16a; PGS16b; Bri16].[7] When we implemented the first version of StreamTeam-Football we adopted concepts and algorithms from the PAN worker implementations. Although the initial workers have been modified, improved, and extended over and over again some concepts and algorithms can still be found in the current version of StreamTeam-Football.

The current version of StreamTeam-Football which is published on GitHub (see Appendix B) consists of 14 workers (listed in Table 9.1) which together consume elements of two raw input streams – elements of the match metadata stream and the raw position sensor data stream which are emitted by our sensor simulators – and emit elements of 19 atomic event streams, three non-atomic event streams, four state streams, and nine statistics streams. The key of every data stream element is set to be the unique identifier of the match it belongs to in order to enable analyzing multiple matches separately and independently in parallel. The real-time football analysis workflow formed by all 14 workers is depicted in Figure 9.1.

In a nutshell, StreamTeam-Football's analysis workflow detects diverse atomic events, such as kickoffs, ball possession changes, set plays, shots, passes, and even pass sequences. Moreover, also non-atomic dribblings, duels and under pressure situations are detected. In addition, StreamTeam-Football generates many statistics, such as heatmaps, ball possession statistics, and pass statistics, and calculates states, such as a virtual offside line and information about

[7] Note that even the most extensive PAN-based football analysis workflow which has been presented in [Bri16] is much simpler than the analysis workflow which belongs to the current version of StreamTeam-Football.

Worker	Analysis subtask	Details
Field object state generation worker	Transforms raw position sensor data stream elements into unified field object state stream elements with additional information	Section 9.2.1.1
Kickoff detection worker	Detects kickoffs and informs which team plays in which direction	Section 9.2.1.2
Time worker	Informs about the current match time in seconds	Section 9.2.1.3
Area detection worker	Detects if a field object (i.e., a player or the ball) enters or leaves an area	Section 9.2.1.4
Set play detection worker	Detects freekicks, cornerkicks, goalkicks, penalties, and throwins and generates set play statistics	Section 9.2.1.5
Ball possession worker	Detects ball possession changes as well as duels and generates ball possession statistics	Section 9.2.1.6
Offside worker	Generates a virtual offside line	Section 9.2.1.7
Pressing analysis worker	Calculates a pressing metric and detects under pressure situations	Section 9.2.1.8
Kick detection worker	Detects kicks (i.e., if the ball has moved away from the player in ball possession)	Section 9.2.1.9
Pass and shot detection worker	Detects successful passes, interceptions, misplaced passes, clearances, goals, and shots off target and generates pass statistics and shot statistics	Section 9.2.1.10
Pass combination detection worker	Detects pass sequences as well as double passes and generates pass sequence statistics	Section 9.2.1.11
Distance and speed analysis worker	Detects speed level changes as well as dribblings and generates distance statistics, speed level statistics, and dribbling statistics	Section 9.2.1.12
Team area worker	Generates information about the areas which are spanned by the players of the teams	Section 9.2.1.13
Heatmap worker	Generates heatmaps for individual players and the teams	Section 9.2.1.14

Table 9.1 StreamTeam-Football's Workers. List of the StreamTeam workers which form StreamTeam-Football's analysis workflow.

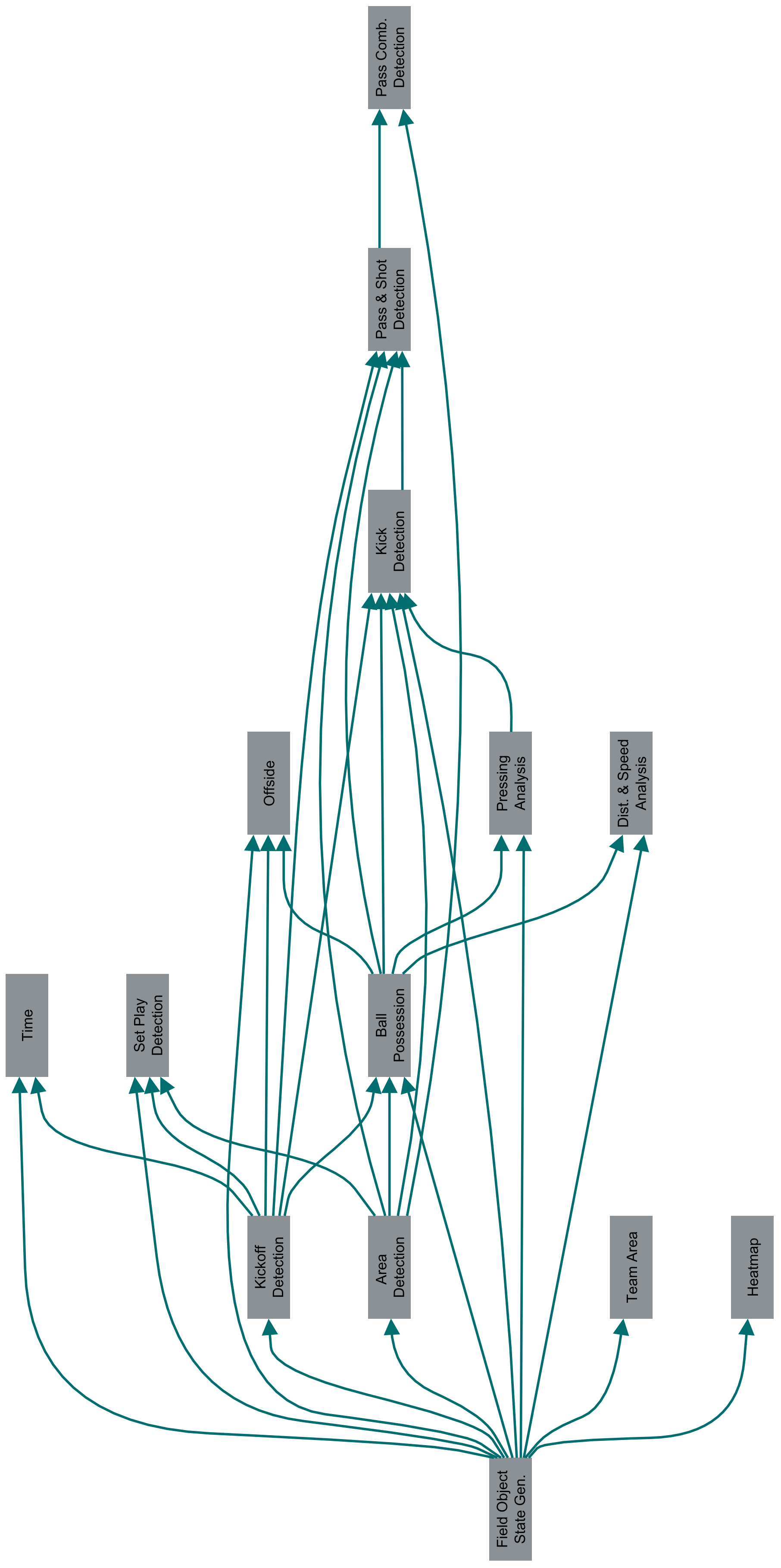


Figure 9.1 **StreamTeam-Football's Analysis Workflow.** The gray boxes represent the StreamTeam workers. Each dark-mint arrow visualizes that the worker (or more precisely the Samza tasks which perform the analysis subtask specified by this worker) at the end of the arrow consumes elements of one or multiple data streams which the worker (or more precisely the Samza tasks which perform the analysis subtask specified by this worker) at the start of the arrow emits. Raw input streams in general and output streams which are not consumed by any worker are not depicted.

Category	Alphabetical list
Raw inputs	Match metadata, raw player/ball positions
Atomic events	Area entry/leave actions, ball possession changes, clearances, cornerkicks, double passes, freekicks, goals, goalkicks, interceptions, kicks, kickoffs, match time progresses, misplaced passes, pass sequences, penalties, shots off target, speed level changes, successful passes, throwins
Non-atomic events	Dribblings, duels, under pressure situations
States	Field object information, offside line, pressing metric, team area surface
Statistics	Ball possession, distance, dribbling, heatmap, pass, pass sequence, set play, shot, speed level

Table 9.2 StreamTeam-Football's Inputs and Outputs. Alphabetical lists of all raw input data which StreamTeam-Football consumes, all atomic and non-atomic events which StreamTeam-Football detects, all states which StreamTeam-Football calculates, and all statistics which StreamTeam-Football generates.

the areas which the teams span. A full list of all consumed raw input data, all detected atomic and non-atomic events, all calculated states, and all generated statistics is given in Table 9.2.

9.2.1 Worker

In this section we will describe StreamTeam-Football's analysis workflow in more detail. More precisely, we will present for each worker which analysis subtask it performs, on the basis of which input streams it performs this analysis subtask, how it performs the analysis subtask, and in which output streams it emits the analysis results. The module graphs of three workers are depicted in Appendix C.

However, before we present the individual workers in detail we want to highlight that StreamTeam-Football's analysis workflow is not defined explicitly but only implicitly by means of the input and output streams of the workers (see Section 5.2.2). In consequence, each worker requires only that the elements of each of its input streams are emitted as output stream elements by a preceding worker (or even multiple preceding workers) but not that they are emitted by a certain predefined worker. For instance, the kick detection worker requires that pressing state stream elements are emitted by one or multiple preceding workers but not that they are emitted by the pressing analysis worker which the current version of StreamTeam-Football's analysis workflow comprises. Hence, the workers of StreamTeam-Football's analysis workflow can be replaced, merged, or split and new workers can be added without modifying a workflow specification file or the other workers.

9.2.1.1 Field Object State Generation Worker

The field object state generation worker transforms raw position sensor data stream elements into unified field object state stream elements with additional information. The idea to unify the raw input data in the first worker is inspired by Herakles' data abstraction approach [MBC+15; BBC+15].

Input Streams Match metadata stream, raw position sensor data stream

Process Whenever a raw position sensor data stream element which ships the current position of a field object (i.e., a player or the ball) is processed, all data are extracted from this element and enriched with the velocity of the field object that is calculated by leveraging the latest two positions and generation timestamps of the field object which have been stored in the local state.[8] Subsequently, the position and the velocity are scaled to SI units and the field axes are mirrored if necessary. Moreover, the object identifiers and the group identifiers are renamed using rename maps contained in the match metadata stream element for the match. Finally, a field object state stream element which ships all information about the current state of a field object is generated and emitted.

Output Streams Field object state stream

9.2.1.2 Kickoff Detection Worker

The kickoff detection worker detects kickoffs and informs which team plays in which direction.

Input Streams Field object state stream

[8] Note that all workers of StreamTeam-Football's analysis workflow access the state only via our state abstractions, i.e., via `SingleValueStores` and `HistoryStores` (see Section 8.2.2.1). Moreover, they make heavy use of generic filter, store, and active keys modules (see Section 8.2.1 and Section 8.2.2.2). We have decided to do not describe the usage of our generic modules and abstractions in the "Process" paragraphs in order to keep them abstract and concise.

Process Whenever a new field object state stream element for the ball is processed, the stored positions of all players and the ball are used to check if the spatial arrangement of the players and the ball qualifies for a kickoff. This is done by checking the distance between the ball and the midpoint (see Section 7.1) as well as the number of players per team on the left side, the right side, and in the mid circle (see point containment in Section 7.2). Doing so enables also identifying which team is positioned on which side of the field when a kickoff is detected and thus determining the playing directions of the teams. If this spatial check is passed, it is further checked if enough time has passed since the last kickoff to prevent duplicates (see sequentiality in Section 6.3). If also this temporal check is passed, the occurrence of a kickoff and the playing directions are announced by emitting a new element of the kickoff event stream.

Output Streams Kickoff event stream

9.2.1.3 Time Worker

The time worker facilitates displaying a game clock in StreamTeam-Football's real-time user interface (see Section 9.3) by means of informing about the current match time in seconds.

Input Streams Field object state stream, kickoff event stream

Process Whenever a new field object state stream element for the ball is processed, the current match time in seconds is calculated by comparing the generation timestamp of this element with the stored generation timestamp of the first kickoff event stream element belonging to this match. If the current match time in seconds has increased since the last match time progress event stream element was emitted, a new match time progress event stream element which contains the current match time in seconds is generated and emitted.

Output Streams Match time progress event stream

9.2.1.4 Area Detection Worker

The area detection worker detects if a field object (i.e., a player or the ball) enters or leaves an area.

Input Streams Field object state stream, match metadata stream

Process Whenever a new field object state stream element is processed, the position contained in this element is used to check if the field object is positioned in an area of the field of not (see point containment in Section 7.2). If the fact whether the field object is in an area has changed since the last field object state stream element for this field object was processed, a new area event which announces that the field object entered of left the area is generated and emitted. The fact if the field object entered or left an area is encoded in a boolean field in the payload of the area event stream element. This is done for all areas that are encoded in the area information field of the match metadata stream element for the match.

Output Streams Area event stream

9.2.1.5 Set Play Detection Worker

The set play detection worker detects freekicks, cornerkicks, goalkicks, penalties, and throwins. Moreover, it generates set play statistics.

Input Streams Area event stream, field object state stream, kickoff event stream

Process Whenever a new field object state stream element for the ball is processed, it is first checked if enough time has passed since the last set play event to prevent duplicates (see sequentiality in Section 6.3). If this check is passed, the ball velocity history which is stored in the local state is checked to determine if the ball was static for some time and started just moving again with the last field object state stream element. If this is the case, either a freekick, a goal kick, a penalty, or a cornerkick occurred. Which of these set play events occurred is determined by inspecting the area in which the ball is located and the playing direction of the player who is the nearest neighbor of the ball (see Chapter 7). If this is not the case, i.e., if the ball was not static (or at least not long enough), but the ball just entered the field, a throwin occured. If any set play event (e.g., a throwin) was detected, a corresponding set play event stream element (e.g., a throwin event stream element) is generated and emitted. Moreover, the set play statistics are updated and emitted in new set play statistics stream elements.

Output Streams Cornerkick event stream, freekick event stream, goalkick event stream, penalty event stream, set play statistics stream, throwin event stream

9.2.1.6 Ball Possession Worker

The ball possession worker detects ball possession changes and duels. Moreover, it generates ball possession statistics.

Input Streams Area event stream, field object state stream, kickoff event stream, match metadata stream

Process Whenever a new field object state stream element for the ball is processed, the history of the ball positions and velocities which is stored in the local state is checked to determine if the ball was hit by a player. This is the case if the absolute velocity or the moving direction of the ball changed too much. If the ball was hit, the distance between the nearest player of the ball and the ball is small enough (see Section 7.1), and the nearest player was not in possession of the ball before, the ball possession changed. This is announced by generating and emitting a ball possession change event stream element which contains in its payload the number of players which are closer to the goal than the player in ball possession – a value which is calculated by leveraging the stored positions of all players and the playing direction of the team in ball possession. Moreover, also if the ball left the field (see point containment in Section 7.2), a ball possession change event stream element which announces that no player is in possession of the ball is emitted. In addition, also a duel might be detected when processing a field object state stream element for the ball. More precisely, the start and the end of a duel is detected by inspecting the two nearest players of the ball. If they are close enough to the ball, they belong to different teams, one of them (the defending player) is in possession of the ball, and there is no active duel yet, a new duel started and the first new duel event stream element for this duel event is emitted. If there is already an active duel, but the defending player is not in ball possession anymore, the attacking player changed, their distance to the ball is too large, the two closest players belong to the same team, or the ball left the field, the duel ended and the last duel event stream element for this duel event is emitted. Moreover, whenever a new field object state stream element for the ball is processed while a duel is active, a new duel event stream element which contains update information is generated and emitted. In contrast, the ball possession statistics are updated and emitted periodically (triggered by a timer) in ball possession statistics stream elements for all players and the teams.

Output Streams Ball possession change event stream, ball possession statistics stream, duel event stream

9.2.1.7 Offside Worker

The offside worker generates a virtual offside line.

Input Streams Ball possession change event stream, field object state stream, kickoff event stream

Process Whenever a new field object state stream element for the player in ball possession is processed, the playing direction of the team in ball possession and the stored positions of all players are used to calculate a virtual offside line and to construct a list of those players who would be in offside position if the player in ball possession would pass the ball to them. The position of the virtual offside line and the list of players who would be in offside position are emitted in an offside line state stream element. Moreover, a special offside line state stream element is emitted when the first field object state stream element for a player is processed after the ball left the field and thus no player is in possession of the ball anymore.

Output Streams Offside line state stream

9.2.1.8 Pressing Analysis Worker

The pressing analysis worker calculates the pressing index, a pressing measure developed by the SFISM [Rum20], and detects under pressure situations.

Input Streams Ball possession change event stream, field object state stream

Process Whenever a new field object state stream element is processed, the stored positions and velocities of all players and the ball are used to calculate the current pressing index. If no player is in ball possession since the ball recently left the field, the pressing index is simply zero. Otherwise the pressing index is the player-ball-distance weighted sum of the velocities with which the players of the team that is not in ball possession approach the ball. However, since generating and emitting a new pressing state stream element after every calculation would result in unnecessary many pressing state stream elements with only negligible updates, the pressing state stream element generation and emission is performed periodically (triggered by a timer). In addition, also the detection of under pressure situations is time-triggered. Whenever a new pressing state stream element is generated and emitted, the current pressing index is used to detect under pressure situations. The start of such a situation is detected and the first under pressure event stream element is emitted when the current pressing index steps over a parameterizable threshold. As long as the current pressing index is greater than or equal to this threshold and the player in ball possession does not change, the under pressure situation is active and a new under pressure event stream element which contains update information is generated and emitted whenever the timer triggers. When the current pressing index falls below the threshold or the player in ball possession changes, the under pressure situation ended and the last under pressure event stream element for this under pressure situation is emitted.

Output Streams Pressing state stream, under pressure event stream

9.2.1.9 Kick Detection Worker

The kick detection worker detects if the ball has moved away from the player in ball possession, an event which we denote as a kick.

Input Streams Area event stream, ball possession change event stream, duel event stream, field object state stream, kickoff event stream, match metadata stream, under pressure event stream

Process Whenever a new field object state stream element for the ball is processed, it is checked if there is a player in ball possession. If this check is passed, the position of the ball and the stored position of the player in ball possession are used to check if the distance between the ball and the player in ball possession steps over a parameterizable threshold (see Section 7.1). In this case, the occurrence of a kick is announced by emitting a kick event stream element. New kicks are only detected if the ball was again close enough to the (potentially new) player in ball possession. Note that the kick event stream element does not only contain the information that, where, and when a kick took place. Instead, it further ships if the player who kicked the ball was attacked during the kick, i.e., if there was an active duel or an active under pressure event, and the field zone in which the ball was kicked. This contextual information is acquired by consuming elements of other data streams and storing contained data in the local store. Moreover, the kick event stream element contains the number of players which have been closer to the goal than the player who kicked the ball – a value which is calculated by leveraging the stored positions of all players and the playing direction of the team whose player kicked the ball.

Output Streams Kick event stream

9.2.1.10 Pass and Shot Detection Worker

The pass and shot detection worker detects successful passes, interceptions, misplaced passes, clearances, goals, and shots off target. Moreover, it generates pass statistics and shot statistics.

Input Streams Area event stream, ball possession change event stream, kick event stream, kickoff event stream

Process Whenever a new area event which ships the information that the ball left the field at a certain region or a new ball possession change event stream element which does not ship the information that no player is in possession of the ball is processed, the contained information as well as the information stored for the last kick event are used to check if a pass or shot occurred. To qualify for a pass or shot, the last kick event must not be already the start of the last detected pass or shot and the generation time difference between the kick event and the currently processed area or ball possession change event must be positive (i.e., the kick event must have happened first) but not be too large. If these temporal checks are passed, a pass or shot occurred. In this case, the information who kicked the ball, the playing direction of the kicking team, the field zone in which the ball was kicked, the fact if the kicking player was attacked, and the information where the ball left the field (if an area event is processed) or where and by whom (same or different team as the kicking player) the ball was received (if a ball possession change event stream element is processed) are used to determine which pass or shot event occurred. For instance, a successful pass is detected if the ball was kicked and received by players of the same team. In contrast, a shot off target is detected, if the ball leaves the field close to the goal of the opposing team and the kicking player was not attacked in the defense third of the field. In any case, i.e., if any pass or shot event (e.g., a successful pass) was detected, a corresponding pass or shot event stream element (e.g., a successful pass event stream element) is generated and emitted. Moreover, the pass or shot statistics for the kicking player and his/her team are updated and emitted in new pass or shot statistics stream elements.

Output Streams Clearance event stream, goal event stream, interception event stream, misplaced pass event stream, pass statistics stream, shot off target event stream, shot statistics stream, successful pass event stream

9.2.1.11 Pass Combination Detection Worker

The pass combination detection worker detects pass sequences and double passes. Moreover, it generates pass sequence statistics.

Input Streams Area event stream, clearance event stream, interception event stream, misplaced pass event stream, successful pass event stream

Process Whenever a new successful pass event stream element is processed, the pass sequence to which this successful pass belongs is built. This is done by iterating over the history of the successful pass data – the generation timestamp, the team identifier, the position and identifier of the kicking player, and the position and identifier of the receive player of the last successful passes – which is stored in the local state. More precisely, the history is iterated in reverse order starting with the successful pass event which is currently processed. The iteration is continued and thus the pass sequence is extended into the past as long as the receive player and the kick player match, the generation time difference between two consecutive passes does not exceed a parameterizable threshold, and the pass sequence is not interrupted by a clearance, an interception, a misplaced pass, or the fact that the ball left the field. The latter is checked by comparing the generation timestamps of the successful passes with the generation timestamps of the last clearance event stream element, the last interception event stream element, the last misplaced pass event stream element, and the last area event stream element which ships the information that the ball left the field. If the resulting pass sequence contains at least two passes, it is announced by emitting a pass sequence stream element. If the resulting pass sequence reflects a double pass, i.e., if it contains only a pass from player X to player Y and a pass from player Y to player X, an additional double pass event stream element is generated and emitted. Moreover, whenever a pass sequence is detected, the pass sequence statistics are updated and emitted in new pass sequence statistics stream elements.

Output Streams Double pass event stream, pass sequence event stream, pass sequence statistics stream

9.2.1.12 Distance and Speed Analysis Worker

The distance and speed analysis worker detects speed level changes and dribblings. Moreover, it generates distance statistics, speed level statistics, and dribbling statistics.

Input Streams Ball possession change event stream, field object state stream

Process Whenever a new field object state stream element for a player is processed, the speed level of the player is determined by comparing the absolute velocity contained in the element with the thresholds of the parameterizable speed levels. When the speed level of a player changes this is announced by emitting a speed level change event stream element. Moreover, the speed level statistics for this player and his/her team, i.e., the amount of time the player has or the players of the team have spent in each speed level, is emitted in a new speed level statistics stream element. In addition, also a dribbling might be detected when processing a field object state stream element for a player. When the very same player was in possession of the ball and faster than a parameterizable dribbling speed threshold for long enough, a new dribbling started and the first dribbling event stream element for this dribbling event is emitted. As long as the player stays in ball possession and remains fast enough, a new dribbling event stream element which contains update information is generated and emitted whenever a new field object state stream element which belongs to the player in ball possession is processed. However, as soon as the player in ball possession becomes too slow or the ball possession changes, the dribbling ended and the last dribbling event stream element for the dribbling event is emitted. Moreover, the dribbling statistics for the dribbling player and his/her team are updated and emitted in new dribbling statistics stream elements. In contrast, the distance statistics are updated and emitted periodically (triggered by a timer) in distance statistics stream elements for all players and the teams. More precisely, the distance (see Section 7.1) between the current position and the position where the player was located when the timer triggered last time is calculated for each player and added to the total distance of the players and their teams.

Output Streams Distance statistics stream, dribbling event stream, dribbling statistics stream, speed level change event stream, speed level statistics stream

9.2.1.13 Team Area Worker

The team area worker generates information about the areas which are spanned by the players of the teams.

Input Streams Field object state stream

Process Whenever a new field object state stream element for a player is processed, the stored positions of all players are used to calculate the surface of the minimum bounding rectangle (see Definition 7.7) and the planar convex hull (see Definition 7.8) spanned by all players of the team to which the player whose latest state is shipped in the field object state stream element belongs. If the surface of the minimum bounding rectangle and/or the surface of the planar convex hull has changed, i.e., if one of the players at the boundary has moved, a new team area state stream element is generated and emitted. While the computation of the minimum bounding rectangle surface is straightforward (see Algorithm 7.2), the planar convex hull is computed using the Tektosyne library [Nah20] and its surface is calculated by means of our implementation of the Surveyor's formula [Bra86].

Output Streams Team area state stream

9.2.1.14 Heatmap Worker

The heatmap worker generates heatmaps for individual players and the teams.

Input Streams Field object state stream, match metadata stream

Process Whenever a new field object state stream element for a player is processed, the heatmap for the current one-second-window is updated for the player whose latest state is shipped in the field object state stream element and his/her team. This is done by mapping the position contained in the field object state stream element to the correct heatmap cell and incrementing the counter in the one-second-window heatmap of the player and team by one. The actual heatmap generation and emission is triggered periodically once a second by a timer. Whenever the timer triggers, the current one-second-window heatmaps of all players and teams are used to update the full game heatmaps which are stored in the local state, added to the heatmap diff history stored in the local state, and cleared for the next one-second-window. Moreover, for each interval in the parameterizable interval list and for each player and team a heatmap stream element containing the heatmap for the interval and the player or team is generated and emitted. If the interval list contains the special full game interval, the full game heatmap is simply read from the store. The heatmaps for all other intervals are built jointly on a per player/team basis by iterating over the heatmap diff history. Since heatmaps are quite large, the fact that especially the player heatmaps for short intervals are very sparse is leveraged when storing the one-second-window heatmaps in the heatmap diff history and when encoding a heatmap in a heatmap statistics stream element.

Output Streams Heatmap statistics stream

9.2.2 Limitations

As the workers of StreamTeam-Football's analysis workflow perform nontrivial analysis subtasks the logic of some worker-specific modules is inevitably complicated. In order to keep the logic of StreamTeam-Football's worker-specific modules as concise and understandable as possible we have decided to rely on two facts which are neither covered by the assumptions that we have defined in our model (see Part II) nor offered automatically by the implementation of StreamTeam's data stream analysis system prototype (see Section 8.2).

First, StreamTeam does not guarantee that data stream elements are processed in generation time order (see Section 8.2.3.3). Therefore, all generic modules that are shipped with StreamTeam are either able to or do not have to handle elements that are processed out-of-order with respect to their generation timestamps. The active keys processing module stores the maximum generation timestamp instead of the last observed generation timestamp and the active keys window module uses the correct generation timestamp when generating inner active keys stream elements (see Section 8.2.2.2). Moreover, the store and the filter module access (and store) only information contained in single input stream elements and potentially forward the input stream elements to its successor modules but do not generate new output stream elements.[9] In con-

[9] Note that the store module stores always only the last observed value (for each inner key and partitioning key) into the `SingleValueStore` and that it builds histories by always appending the last observed value to the `HistoryStore` (see Section 8.2.2.1). We argue that this behavior is reasonable since the fact if the ordering with respect to the generation timestamps matters more than the sequence number ordering is application-specific. Hence, it is the duty of the worker-specific modules which access the `SingleValueStore` or the `HistoryStore` to select the correct values.

trast, the most worker-specific modules of StreamTeam-Football as well as the placement of the active keys processor module in the single element processor graphs of some football analysis workers rely on the fact that all data stream elements are processed in correct order with respect to their generation timestamps (even across data stream partitions). That is, StreamTeam-Football does not handle (or at least not entirely) input stream elements which are processed out-of-order with respect to their generation timestamps – an issue which can lead to rare misdetections and slightly too small generation timestamps. For instance, a corrupt ball possession change event stream element which announces that a certain player is in possession of the ball might be generated if the last observed field object state stream element for this player has a much lower generation timestamp and is thus much older than the last field object state stream elements for the other players. Moreover, a ball possession change event stream element might have a too small generation timestamp if the last processed field object state stream element for the player who obtained the ball possession has a greater generation timestamp than the generation timestamp of the field object state stream element for the ball which triggered the detection of the ball possession change event.

Second, StreamTeam adopts only Samza's guarantee that each input stream element is processed at least once [Sam17c; Sam17j; NPP+17] as this is consistent with the network assumptions we have posed in our system model (see Section 5.4.2). Moreover, if StreamTeam's generic filter and store modules process an input stream element multiple times they simply forward the element multiple times to their successor modules after filtering it or storing its content multiple times. Hence, the responsibility to handle duplicate input stream elements rests with the worker-specific modules. However, the most worker-specific modules of StreamTeam-Football rely on the fact that each input stream element is received and processed exactly once. That is, the logic of these modules does not handle if input stream elements are processed multiple times – an issue which can lead for instance to corrupted statistics. For instance, heatmaps might be corrupted by increasing the counters twice if some field object state stream elements are processed a second time.

We admit that relying on these two facts can be regarded as a limitation of StreamTeam-Football's analysis workflow as doing so can result in incorrect analysis results. Hence, although we argue that rare corrupt results do not have severe consequences in football analysis (in contrast to health telemonitoring [Bre08]), we plan to eliminate this limitation by improving the logic of all worker-specific modules in our future work (see Section 13.3). However, note that fully correct and deterministic analysis results cannot be guaranteed even if data stream elements which are duplicates or processed out-of-order with respect to their generation timestamps are handled by all modules since the modules compare environmental condition dependent generation timestamps and since window calls are triggered by the system time of the Samza tasks. Therefore, we suppose that the negligible non-determinism effects which are observable in StreamTeam-Football's analysis results – for instance the number of detected successful passes and interceptions varies slightly between multiple analysis attempts – will be still observable even if we improve all worker-specific modules.

In addition to these system-related limitations there are also some application-specific limitations. First, many worker-specific modules of StreamTeam-Football rely on the fact that new raw position sensor data stream elements are generated and emitted in a high frequency. Although this is no problem for the datasets which we replay (see Section 9.1), we admit that this can be regarded as a limitation. Moreover, since there is no reliable input which announces the start and the end of the halftime break or player substitutions, some analyses (e.g., the match time calculation and the ball possession statistics generation) work only during the first halftime or until the first player is substituted yet.[10] As a last point we want to highlight that we are no sports scientists but computer scientists. In consequence, fine tuning the workers in order to optimize the football analysis results was not the objective of our research. Nevertheless, we argue that StreamTeam-Football's analysis workflow definitely serves as a convincing proof-of-concept for algorithmically analyzing football matches in real-time. In fact, as shown in [SRP+19], StreamTeam-Football fulfills already some analysis demands which are extracted from interviews with football coaches and can be easily extended to perform additional analyses requested by the coaches. In the future, we will continue our collaboration with the SFISM in order to improve and extend the analyses which StreamTeam-Football performs (see Section 13.3).

[10] Note that, the sensor simulators would support simulating both halftimes if there is a single file containing the positions for both half times with the match start as the origin.

9.3 Real-Time User Interface

In addition to the StreamTeam workers which form the football analysis workflow, StreamTeam-Football comprises a Web client which visualizes the emitted analysis results in real-time. This Web client is the platform-independent application-specific user interface of StreamTeam-Football. It is intended to provide coaches and match analysts with live results during the match (see Section 2.1) but proved further to be a great tool for football analysis worker developers to test if the workers produce the expected analysis results (e.g., if successful passes are detected correctly).

The first Web client of our group which served as a considerable user interface for visualizing football analysis results in real-time was developed in a student project [Bri16] to visualize the analysis results of the football analysis workflow which the student has implemented in PAN. When we implemented the first version of StreamTeam-Football, we have adopted this Web client as the real-time user interface of StreamTeam-Football. Afterwards, we have modified, improved, and extended the Web client over and over again and presented later versions in [PBS+17] and [PRS+18]. The current version which visualizes almost all analysis results emitted by StreamTeam-Football's current analysis workflow that we have presented in Section 9.2 is published on GitHub (see Appendix B).

The current version of StreamTeam-Football's Web client is implemented in HTML, CSS, and JavaScript with the assistance of Chart.js [Cha20], graham_scan_js [Bar20], heatmap.js [Wie20], jQuery [jQu20a], jQuery selectBox [jQu20b], protobuf.js [Wir20], rgb-color [Tre20], Tooltipster [JA20], tooltipster-follower [Ame20], Video.js [Bri20], and StreamTeam's JavaScript API (see Section 8.2.4). In order to visualize the analysis results in real-time the Web client pulls the latest elements of all but four data streams for the currently selected match periodically in data stream specific intervals by means of issuing consume queries with the match identifier as the key and data stream specific limits to the Kafka REST proxy (see Section 8.3.1). The elements of the area event stream, the speed level change event stream, and the kick event stream are not pulled as these events are not visualized in the user interface. Moreover, also the raw position sensor data stream is omitted as all data contained in the elements of this data stream are also contained in the elements of the unified field object state stream (see Section 9.2.1.1).

StreamTeam-Football's Web client provides many live visualizations which

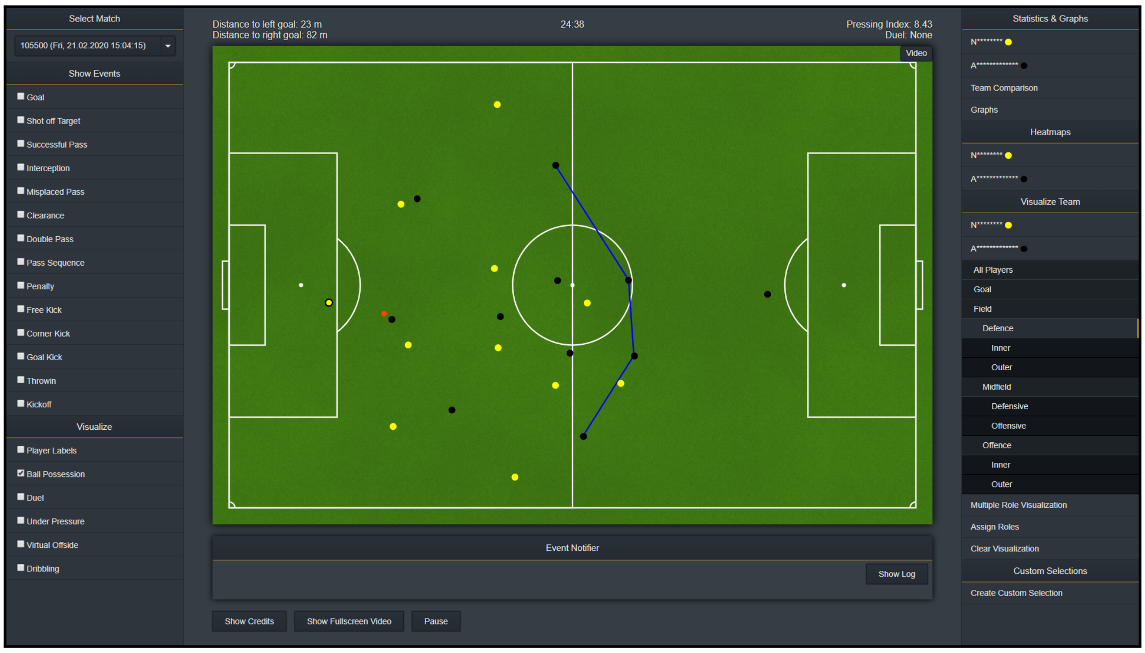
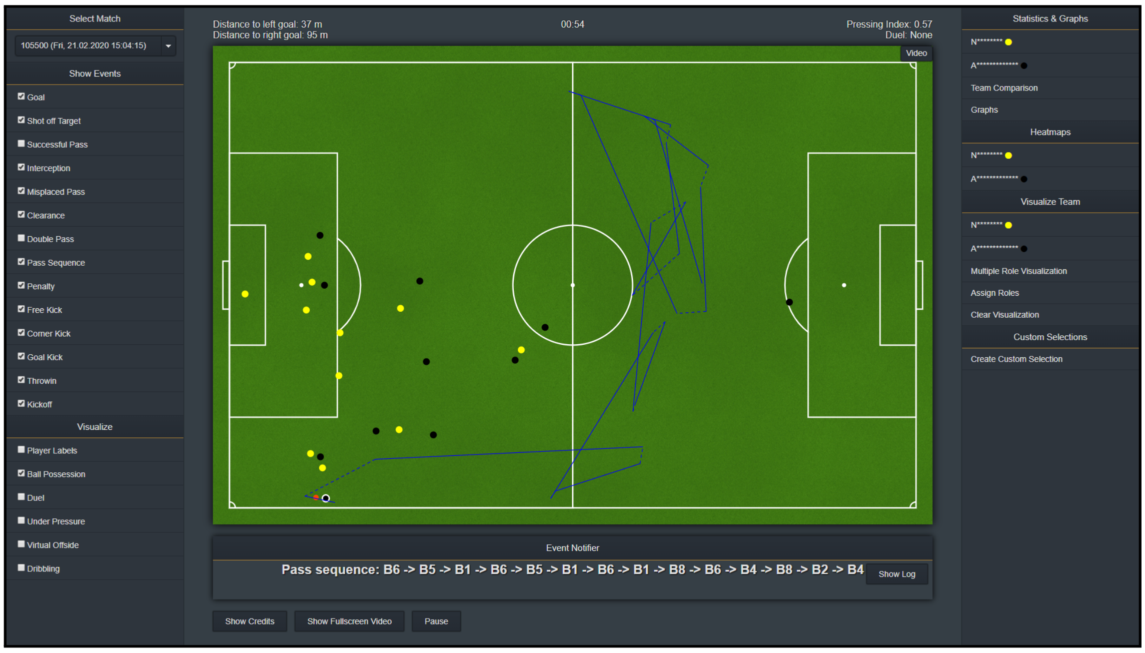

Distance to left goal: 23 m
Distance to right goal: 82 m
24:38
Pressing Index: 8.43
Duel: None
105500 (Fri, 21.02.2020 15:04:15)

Figure 9.2 Defense Line in StreamTeam-Football's Web Client. The screenshot shows the visualization of the spatial arrangement of the defense players of the black team. The blue lines are used to highlight the spatial arrangement. Also minimum bounding rectangles and planar convex hulls are supported. The player and team names are anonymized for privacy reasons.

support coaches (and their match analysts) in making the right decisions during the match. Among others, the Web client helps coaches to inspect the spatial arrangement of a set of players by means of drawing the line, minimum bounding rectangle, or planar convex hull which is spanned by them (see Figure 9.2). Moreover, the Web client visualizes detected atomic events (see Figure 9.3) and active non-atomic events (see Figure 9.4) on the field. In addition, also the virtual offside line (see Figure 9.5) and the heatmaps (see Figure 9.6) are visualized directly on the field. The other states and statistics are visualized as bar charts (see Figure 9.7) and graphs (see Figure 9.8), respectively.


Select Match

105500 (Fri, 21.02.2020 15:04:15)

Show Events

- ☑ Goal
- ☑ Shot off Target
- ☐ Successful Pass
- ☑ Interception
- ☑ Misplaced Pass
- ☑ Clearance
- ☐ Double Pass
- ☑ Pass Sequence
- ☑ Penalty
- ☑ Free Kick
- ☑ Corner Kick
- ☑ Goal Kick
- ☑ Throwin
- ☑ Kickoff

Visualize

- ☐ Player Labels
- ☑ Ball Possession
- ☐ Duel
- ☐ Under Pressure
- ☐ Virtual Offside
- ☐ Dribbling

Distance to left goal: 37 m
Distance to right goal: 95 m

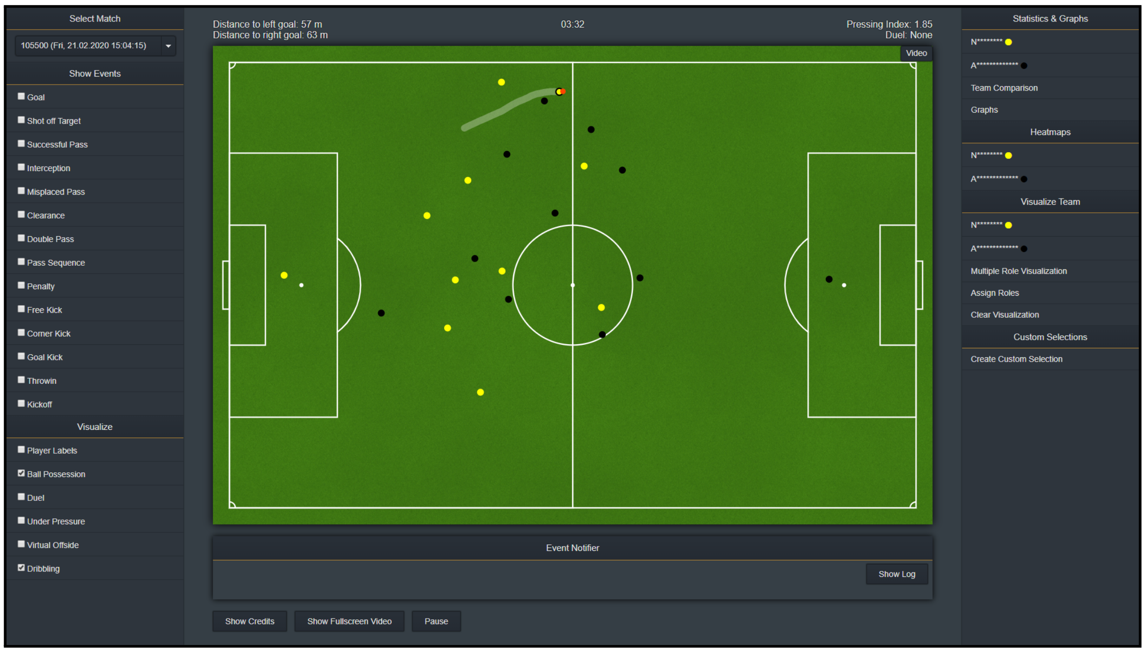
00:54

Pressing Index: 0.57
Duel: None

Event Notifier

Pass sequence: B6 -> B5 -> B1 -> B6 -> B5 -> B1 -> B6 -> B1 -> B8 -> B6 -> B4 -> B8 -> B2 -> B4

Figure 9.3 Pass Sequence in StreamTeam-Football's Web Client. The screenshot shows the visualization of a detected pass sequence event. The solid blue lines illustrate successful passes and the dotted blue lines visualize the walked path between two passes of the sequence. The player and team names are anonymized for privacy reasons.


Select Match

105500 (Fri, 21.02.2020 15:04:15)

Show Events

- ☐ Goal
- ☐ Shot off Target
- ☐ Successful Pass
- ☐ Interception
- ☐ Misplaced Pass
- ☐ Clearance
- ☐ Double Pass
- ☐ Pass Sequence
- ☐ Penalty
- ☐ Free Kick
- ☐ Corner Kick
- ☐ Goal Kick
- ☐ Throwin
- ☐ Kickoff

Visualize

- ☐ Player Labels
- ☑ Ball Possession
- ☐ Duel
- ☐ Under Pressure
- ☐ Virtual Offside
- ☑ Dribbling

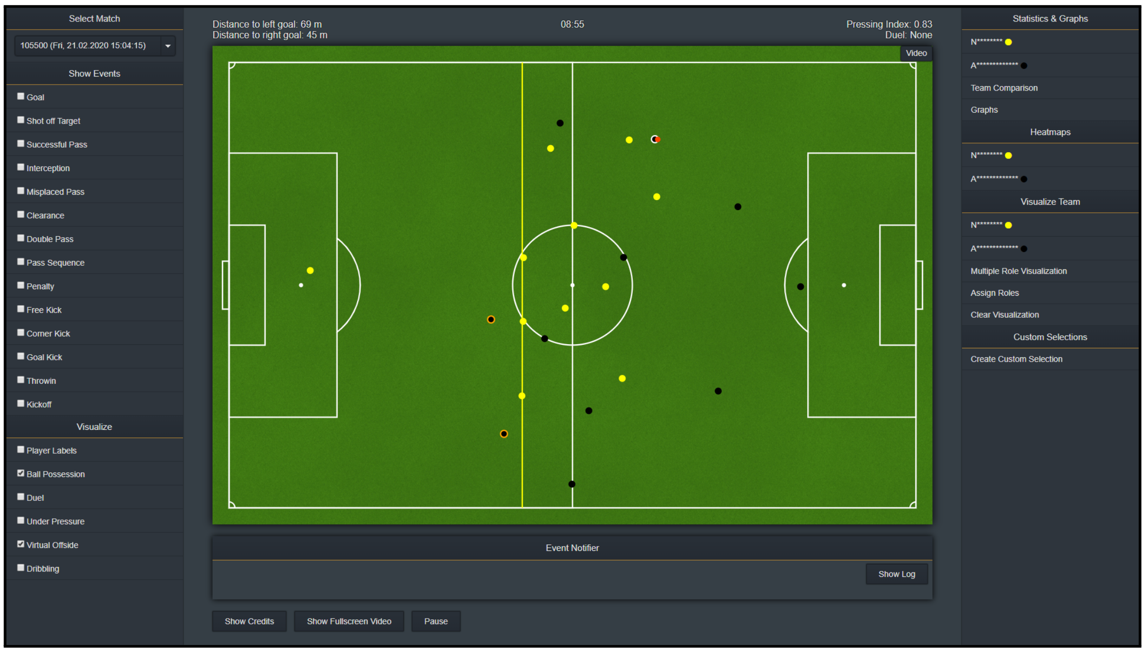
Distance to left goal: 57 m
Distance to right goal: 63 m

03:32

Pressing Index: 1.85
Duel: None

Event Notifier

Figure 9.4 Dribbling in StreamTeam-Football's Web Client. The screenshot shows the visualization of an active dribbling event. The white trace visualizes the path along which the player in ball possession has dribbled so far. The player and team names are anonymized for privacy reasons.



105500 (Fri, 21.02.2020 15:04:15)

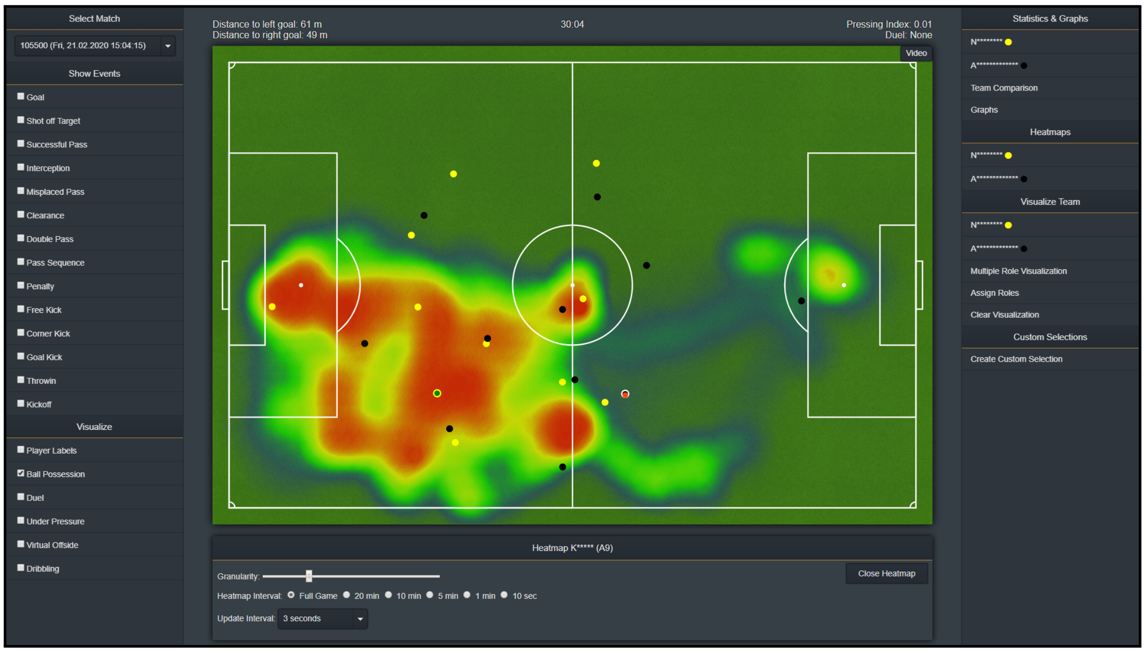
[✓ Ball Possession] [✓ Virtual Offside]

Distance to left goal: 69 m
Distance to right goal: 45 m

08:55

Pressing Index: 0.83
Duel: None

Figure 9.5 Virtual Offside Line in StreamTeam-Football's Web Client. The screenshot shows the visualization of the virtual offside line. The players who would be in offside position are highlighted with an orange border and the virtual offside line is illustrated with a yellow line. The player and team names are anonymized for privacy reasons.



105500 (Fri, 21.02.2020 15:04:15)

[✓ Ball Possession]

Distance to left goal: 61 m
Distance to right goal: 49 m

30:04

Pressing Index: 0.01
Duel: None

Heatmap K***** (A9)

Granularity:

Heatmap Interval: [✓ Full Game] 20 min 10 min 5 min 1 min 10 sec

Update Interval: 3 seconds

Figure 9.6 Heatmap in StreamTeam-Football's Web Client. The screenshot shows the visualization of the full game heatmap for player A9 who is highlighted in green. The redder an area is the more often player A9 was positioned in this area. The player and team names are anonymized for privacy reasons.

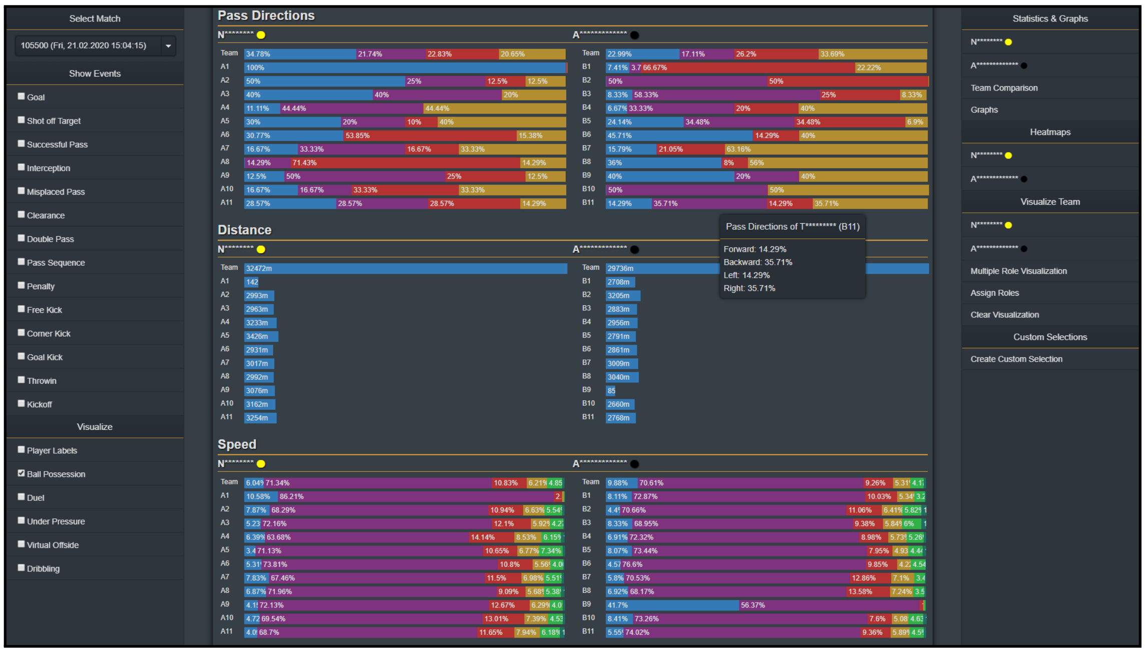


Figure 9.7 Statistics in StreamTeam-Football's Web Client. The screenshot shows bar charts for some statistics. All bar charts are updated in real-time. Moreover, each bar chart shows a tool tip when mouse hovered. The player and team names are anonymized for privacy reasons.

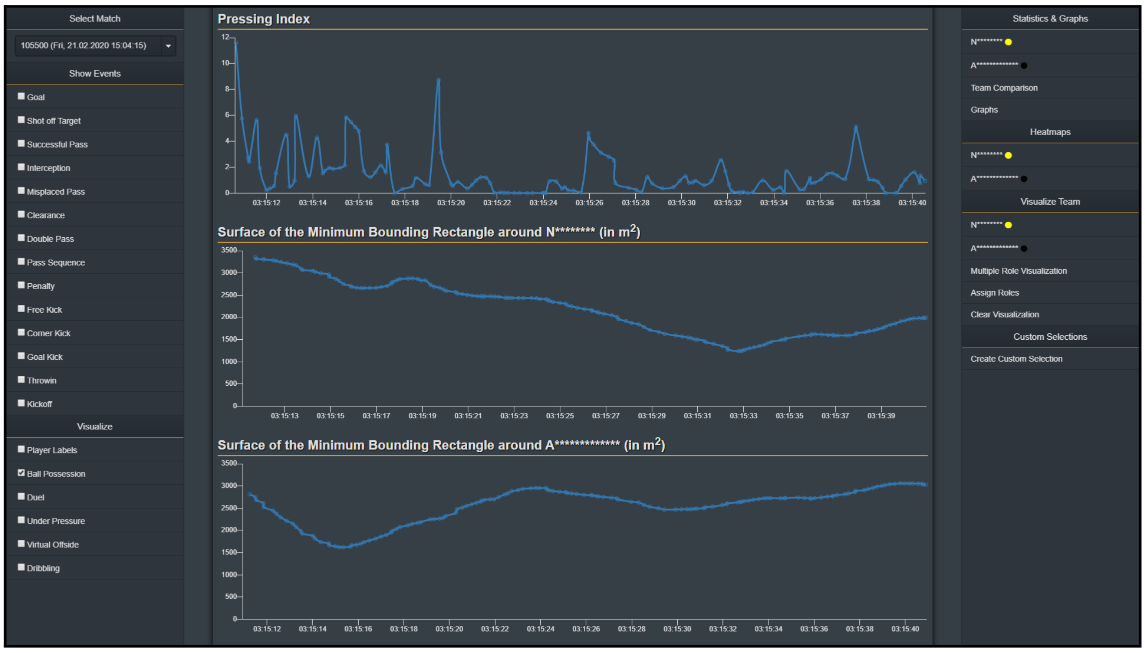

Select Match
105500 (Fri, 21.02.2020 15:04:15)
Show Events
Visualize
Pressing Index
Surface of the Minimum Bounding Rectangle around N******** (in m²)
Surface of the Minimum Bounding Rectangle around A************* (in m²)
Statistics & Graphs
Team Comparison
Graphs
Heatmaps
Visualize Team
Multiple Role Visualization
Assign Roles
Clear Visualization
Custom Selections
Create Custom Selection

Figure 9.8 State Graphs in StreamTeam-Football's Web Client. The screenshot shows graphs for some states. Each graph is updated in real-time and visualizes the values of a state for the last 30 seconds. The player and team names are anonymized for privacy reasons.

9.4 Persistent Storage for Offline Activities

According to StreamTeam's infrastructure (see Section 8.1) all data stream elements can be consumed directly from the Kafka brokers or from our Kafka REST proxy. This is great for all online activities such as real-time visualizations (see Section 9.3). However, we argue that it makes sense to store StreamTeam-Football's analysis results persistently in a database for offline activities as doing so enables for instance querying all successful pass events which fulfill some properties (e.g., kicked by player X) instead of consuming all successful pass event stream elements in sequence number order.

In this section, we will present how StreamTeam-Football stores the analysis results it generates in its workflow persistently in a MongoDB instance (see Section 9.4.1 and Section 9.4.2). Moreover, we will show that the stored analysis results can be used as video tags in SportSense, our offline team sports video retrieval system (see Section 9.4.3).

9.4.1 Database

In StreamTeam-Football, we leverage a MongoDB [Mon20] instance to store the analysis results produced in StreamTeam-Football's analysis workflow (see Section 9.2) and the match metadata persistently for offline activities. MongoDB is a document database which stores data as binary JSON documents [BSO20] in collections [Mon20]. We have decided to store the data in a MongoDB instance since MongoDB is a spatial database [Güt94] which supports all spatial features SportSense requires (e.g., point containment queries with arbitrary polygons and 2D indexes) and exhibits nice performance and scalability characteristics [SGR15; AR15].

The foundation for storing analysis results in a MongoDB instance, i.e., the selection of MongoDB as the optimal database system as well as the design of the first schemata[11] with spatial indexes, has been laid in a student project [Lob17] in which the first Web-based version of SportSense was implemented (Section 9.4.3). Later versions of the schemata have been presented in [PAL+18]. In the following we will present how, i.e., in which collections with which schemata and which indexes, the analysis results and the match metadata are stored in the current version of StreamTeam-Football which is published on GitHub (see Appendix B).

[11] These first schemata have been considered when designing the final database schemata (see Figure 9.9, 9.10, and 9.11) and the data stream model (see Chapter 4).

Version 3.6 of MongoDB which we use in StreamTeam-Football supports specifying the schema of each collection, or more precisely the schema of the binary JSON documents that are stored in a collection. For doing so, a JSON schema object which defines the schema according to a draft of the JSON Schema standard [GZC13] has to be specified as the `$jsonSchema` operator when creating the collection [Mon17d]. If a schema is specified, it is enforced by means of validating new documents before they are added to the collection [Mon17d]. Moreover, MongoDB supports creating single field indexes and spatial 2D indexes to speed up queries [Mon17b].

In StreamTeam-Football, we make use of MongoDB's schema and index features. More precisely, we specify five collections with well-defined JSON schemata and indexes for storing the analysis results and the match metadata.

The metadata of every match are stored in the *matches* collection. More precisely, each match metadata stream element is stored as a data item, i.e., as a binary JSON document which is structured in accordance with the correct schema, in the matches collection. Since there is only a single match metadata stream element for every match, there is also only a single data item in the matches collection for every match which contains all information about the match. Each match metadata item has 17 properties for storing information such as the match identifier, the field size, and the player names. The full schema of the matches collection and a concrete sample match metadata item are given in Figure 9.9 and Figure 9.12(b). To speed up queries there is a single field index for each property.

Detected atomic events, calculated states, and generated statistics are stored in the *events*, *states*, and *statistics* collection, respectively. More precisely, each atomic event stream element (e.g., successful pass event stream element) is stored as a data item in the events collection (see Figure 9.13(b) for a concrete sample successful pass event data item), each state stream element (e.g., field object state stream element) is stored as a data item in the states collection, and each statistics stream element (e.g., pass statistics stream element) is stored as a data item in the statistics collection. All three collections have the same schema which is given in Figure 9.10. There are some strictly structured properties for storing the information which is encoded in the generic attributes of a data stream element (see Chapter 4). This includes amongst others the match identifier, the timestamp, the positions, and the involved players. In addition, there is the `additionalInfo` property which can be filled with an arbitrary JSON object for storing the data stream specific information which is encoded in the payload of a data stream element. In consequence, the schema of the events, states, and statistics collection is not fully fixed as the matches collection schema but only semi-fixed. We argue that storing all events in the same collection with the same semi-fixed schema instead of having event-specific collections for each event stream with different fully fixed event-specific schemata is beneficial as it enables for instance to query all events which involved a certain player or all events which occurred in a certain area of the field with a single simple query on the events collection. The same is true for states and statistics. Moreover, having a consistent semi-fixed schema enables leveraging consistent indexes – a 2D index for the `xyCoords` property[12] and a single field index for all other properties except for the `additionalInfo` property – to speed up such queries.

Each non-atomic event stream element (e.g., duel event stream element) that ships updates of a non-atomic event is stored as a data item in the *nonatomicEvents* collection which has a slightly extended schema given in Figure 9.11. More precisely, each data item in this collection has three additional properties, namely `eventId`, `phase`, and `seqNo`. These additional properties are required to group the data items which belong to the same non-atomic event, to identify the data items which inform about the start and the end of a non-atomic event, and to order all data items with belong to the same non-atomic event (cf. *eid*, φ, and ξ of our data stream model).[13] In order to speed up queries which include conditions on these properties, this collection does not only have the same indexes as the events, states, and statistics collection but additional single field indexes for each of the three additional properties.

The reason why we have designed StreamTeam-Football's database to have no collection for storing arbitrary raw input stream elements but a dedicated matches collection is that StreamTeam-Football's analysis workflow consumes only elements of two raw input streams, namely the match metadata stream and the raw position sensor data stream, and that there is no benefit of storing a data item for each raw position sensor data stream element. This is due to the fact that the information which is shipped in a raw position sensor data stream element is also contained in the enriched and unified field object state stream element which the field object state generation worker (see Section 9.2.1.1) emits and thus stored already in the states collection.

As a last point we want to highlight that StreamTeam-Football's database

[12] The z-coordinate of the positions is separated to the `zCoords` property since MongoDB supports only 2D but no 3D indexes [Mon17b].

[13] Note that data items for atomic event, state, and statistics stream elements do not require these properties as their event identifier and phase is always λ and since there is always only a single data stream element for each atomic event, state, and statistics (see Section 4.4).

```
properties: {
     matchId: {
          description: "Match identifier",   bsonType: "string" },
     sport: {
          description: "Sport discipline",   bsonType: "string" },
     fieldSize: {
          description: "Field size (width, height)",
          bsonType: "array",
          items: { bsonType: "double" } },
     date: {
          description: "Date in ISO 8601 format",   bsonType: "string" },
     competition: {
          description: "Context/Competition",   bsonType: "string" },
     venue: {
          description: "Venue",  bsonType: "string" },
     homeTeamId: {
          description: "Identifier of the home team",
          bsonType: "string" },
     awayTeamId: {
          description: "Identifier of the away team",
          bsonType: "string" },
     homePlayerIds: {
          description: "Identifiers of the players of the home team",
          bsonType: "array",
          items: { bsonType: "string" } },
     awayPlayerIds: {
          description: "Identifiers of the players of the away team",
          bsonType: "array",
          items: { bsonType: "string" } },
     homeTeamName: {
          description: "Name of the home team",   bsonType: "string" },
     awayTeamName: {
          description: "Name of the away team",   bsonType: "string" },
     homePlayerNames: {
          description: "Names of the players of the home team",
          bsonType: "array",
          items: { bsonType: "string" } },
     awayPlayerNames: {
          description: "Names of the players of the away team",
          bsonType: "array",
          items: { bsonType: "string" } },
     videoPath: {
          description: "Path where the video of the match can be found",
          bsonType: "string" },
     homeTeamColor: {
          description: "Color of the home team",   bsonType: "string" },
     awayTeamColor: {
          description: "Color of the away team",   bsonType: "string" }
},
required: ["matchId", "sport", "fieldSize", "date", "competition", "venue",
           "homeTeamId", "awayTeamId", "homePlayerIds", "awayPlayerIds",
           "homeTeamName", "awayTeamName", "homePlayerNames", "awayPlayerNames",
           "videoPath", "homeTeamColor", "awayTeamColor"]
```

Figure 9.9 Schema of the Matches Collection. The JSON schema object which is specified as the $jsonSchema operator when creating the matches collection.

```
properties: {
     type: {
          description: "Type of the data item (stream name of the data stream
                         element)",
          bsonType: "string" },
     matchId: {
          description: "Identifier of the match the data item belongs to",
          bsonType: "string" },
     ts: {
          description: "Time in ms since the start of the match",
          bsonType: "int" },
     videoTs: {
          description: "Video offset (in s)",
          bsonType: "int" },
     xyCoords: {
          description: "Array containing the planar position(s) (x and y
                         coordinates of the positions tuple of the data stream
                         element)",
          bsonType: "array",
          items: { bsonType: "array",
               items: { bsonType: "double" } } },
     zCoords: {
          description: "Array containing the z coordinates of the position(s)
                         (z coordinates of the positions tuple of the data
                         stream element)",
          bsonType: "array",
          items: { bsonType: "double" } },
     playerIds: {
          description: "Array containing the involved players (object
                         identifiers tuple of the data stream element)",
          bsonType: "array",
          items: { bsonType: "string" } },
     teamIds: {
          description: "Array containing the involved teams (group identifiers
                         tuple of the data stream element)",
          bsonType: "array",
          items: { bsonType: "string" } },
     additionalInfo: {
          description: "Additional information (payload fields of the data
                         stream element)",
          bsonType: "object" }
},
required: ["type", "matchId", "ts", "videoTs", "xyCoords", "zCoords",
           "playerIds", "teamIds", "additionalInfo"]
```

Figure 9.10 Schema of the Events, States, and Statistics Collection. The JSON schema object which is specified as the $jsonSchema operator when creating the events, states, and statistics collection.

```
properties: {
     type: {
          description: "Type of the data item (stream name of the data stream
                          element)",
          bsonType: "string" },
     matchId: {
          description: "Identifier of the match the data item belongs to",
          bsonType: "string" },
     ts: {
          description: "Time in ms since the start of the match",
          bsonType: "int" },
     videoTs: {
          description: "Video offset (in s)",
          bsonType: "int" },
     xyCoords: {
          description: "Array containing the planar position(s) (x and y
                          coordinates of the positions tuple of the data stream
                          element)",
          bsonType: "array",
          items: { bsonType: "array",
               items: { bsonType: "double" } } },
     zCoords: {
          description: "Array containing the z coordinates of the position(s)
                          (z coordinates of the positions tuple of the data
                          stream element)",
          bsonType: "array",
          items: { bsonType: "double" } },
     playerIds: {
          description: "Array containing the involved players (object
                          identifiers tuple of the data stream element)",
          bsonType: "array",
          items: { bsonType: "string" } },
     teamIds: {
          description: "Array containing the involved teams (group identifiers
                          tuple of the data stream element)",
          bsonType: "array",
          items: { bsonType: "string" } },
     additionalInfo: {
          description: "Additional information (payload fields of the data
                          stream element)",
          bsonType: "object" },
     eventId: {
          description: "Identifier of the event the data item belongs to",
          bsonType: "string" },
     phase: {
          description: "Phase",
          bsonType: "string" },
     seqNo: {
          description: "Sequence number",
          bsonType: "long" }
},
required: ["type", "matchId", "ts", "videoTs", "xyCoords", "zCoords",
           "playerIds", "teamIds", "additionalInfo", "eventId", "phase",
           "seqNo"]
```

Figure 9.11 Schema of the NonatomicEvents Collection. The JSON schema object which is specified as the $jsonSchema operator when creating the nonatomicEvents collection.

design, i.e., the five collections with their well-defined JSON schemata and indexes, is not football-specific but can be used to store the analysis results and match metadata for abitrary teams sports and even eSports. In fact, earlier version of our database design have been used already in student projects to store automatically detected DotA2 events [Zum19] and manually annotated ice hockey events [Rau17; PAL+18].

9.4.2 MongoDB Stream Importer

In order to fill the MongoDB instance automatically we have implemented the MongoDB stream importer. The MongoDB stream importer is a dedicated Kafka consumer which consumes all event, state, and statistics stream elements as well as the match metadata stream elements, transforms them into data items which are structured in accordance with the schemata presented in Section 9.4.1, and stores those data items into the correct collections.

The first approach towards filling StreamTeam-Football's analysis results automatically into the MongoDB instance has been made in the student project [Lob17] which laid also the foundation for storing analysis results in the MongoDB instance at all. However, in this project only few analysis results have been consumed from an earlier version of StreamTeam-Football's analysis workflow and the elements of each data stream have been handled differently. A later approach to consume more analysis results and to handle the data stream elements more generically has been presented in [PRS+18] and used in another student project [Zum19] to store DotA2 events detected in an elementary StreamTeam workflow for analyzing DotA2 matches. The current version of StreamTeam-Football's MongoDB stream importer which transforms and stores all analysis results produced in StreamTeam-Football's analysis workflow (see Section 9.2) fully automatically and generically is published on GitHub (see Appendix B). In the following, we will present how the current version of StreamTeam-Football's MongoDB stream importer works.

As the Kafka REST proxy (see Section 8.3.1) the MongoDB stream importer uses the Consumer API of Kafka's original Java library [Kaf16] to pull all match metadata stream elements, event stream elements, state stream elements, and statistics stream elements periodically from the Kafka brokers. The list of data streams for which the MongoDB stream importer pulls new elements is maintained automatically at runtime using the same mechanism as in the Kafka REST proxy.[14] All data stream elements which are consumed in a pull are transformed automatically into new data items.

Match metadata stream elements are filtered by means of the data stream name (*name*) and handled separately. The values of all properties can be extracted from the match metadata stream element, albeit doing so might require converting the data format (e.g., the match start UNIX timestamp contained in the match metadata stream element is converted into the ISO 8601 date format [ISO88]) or parsing (e.g., the player identifiers and names are parsed from the object rename map contained in the match metadata stream element). The transformation of a sample match metadata stream element into a match metadata item is illustrated in Figure 9.12.

All (atomic and non-atomic) event, state, and statistics stream elements are handled in the same generic way. The category (*cat*) and the atomicity flag (*ato*) are used to select the collection in which the data item to which an event, state, and statistics stream element is transformed has to be inserted. The values of all properties except for `ts` and `videoTs` are simply extracted from the data stream element. For instance, the value of the `playerIds` property is a one-to-one copy of the object identifiers tuple (*oids*), the value of the `type` property is the data stream name (*name*), and the value of the `additionalInfo` property is a JSON encoding of the payload (*pd*). Moreover, the value of the `xyCoords` property and the `zCoords` property is obtained by extracting the coordinates from the positions in the position tuple (*pos*).[15] Also the values of the additional properties of the non-atomic event data items are simply extracted. For instance, the `seqNo` property is the sequence number (ξ) and thus the Kafka offset (see Section 8.2.3.1). In contrast, the value of the `ts` property and the `videoTs` property are calculated. More precisely, the value of the `ts` property is the generation timestamp (*ts*) of the data stream element minus the generation timestamp of the first data stream element of the match that is contained in the match metadata stream element. The value of the `videoTs` property is the sum of the match start video offset that is contained in the match metadata stream element and the value of the `ts` property converted to seconds (i.e., divided by 1000). The transformation of a sample successful pass event stream element into a successful pass event data

[14] The sole difference is that the MongoDB stream importer excludes the raw position sensor data stream.

[15] Note that data stream elements which contain a position in their position tuple whose x-coordinate or y-coordinate is not in the supported $[-180, 180)$ interval [Mon17a] are simply discarded and thus not stored in the MongoDB instance. We argue that this is acceptable since such positions are far away from the football field – the origin of our coordinate system is the midpoint of the football field and all coordinates are given in meters – and thus most probably based on corrupt input data.

```
Key:  751690
Offset:  1
Value:  {"atomic":true, "generationTimestamp":"3232387", "payload":{ "generation
        TimestampFirstDataStreamElementOfTheMatch":"3232387", "sport":"football",
        "fieldLength":99.89, "fieldWidth":63.9, "mirroredX":true, "areaInfos":
        "{field:-49.945@49.945@-31.9500@31.9500}, {leftThird:-49.945@-16.648@
        -31.9500@31.9500}, ...  MANYMORE ..., {rightTopCorner:46.945@52.945@
        -34.9500@-28.9500}", "matchStartUnixTs":"1581504658059", "competition":
        "testGame", "venue":"Magglingen", "objectRenameMap":"{950:BALL:Ball}%
        {23:A1:L***********}%{29:A2:Z*****}%...  MANYMORE ...%{72:B11:D*****}",
        "teamRenameMap":"{ball:BALL:Ball}%{home:A:F************}%{away:B:S*******
        **********}", "videoPath":"*************.avi", "matchStartVideoOffset":
        "3231", "teamColorMap":"{A:white}%{B:red}", "@type":".streamTeam.football.
        MatchMetadataStreamElementPayload"}}
```

(a) Match Metadata Stream Element as Listed in the Cluster Monitor (cf. Figure 8.7)

```
{
     "_id" :  ObjectId("5e43d892e6b91f0fbef8322c"),
     "matchId" :  "751690",
     "sport" :  "football",
     "fieldSize" :  [99.89, 63.9],
     "date" :  "2020-02-12T10:50:58Z",
     "competition" :  "testGame",
     "venue" :  "Magglingen",
     "homeTeamId" :  "A",
     "awayTeamId" :  "B",
     "homePlayerIds" :  ["A1", "A2", "A3", "A4", "A5", "A6", "A7", "A8", "A9",
                         "A10", "A11"],
     "awayPlayerIds" :  ["B1", "B2", "B3", "B4", "B5", "B6", "B7", "B8", "B9",
                         "B10", "B11"],
     "homeTeamName" :  "F************",
     "awayTeamName" :  "S*****************",
     "homePlayerNames" :  ["L***********", "Z*****", "C*********", "E********",
                           "A*****", "B*****", "P*****", "K*****", "L******",
                           "R*******", "C******"],
     "awayPlayerNames" :  ["Z******", "S********", "F********", "C******",
                           "I********", "K*****", "K******", "C*****",
                           "M*****", "G*******", "D*****"],
     "videoPath" :  "*************.avi",
     "homeTeamColor" :  "white",
     "awayTeamColor" :  "red"
}
```

(b) Match Metadata Item Stored in the Matches Collection of the MongoDB Instance

Figure 9.12 Sample Match Metadata Transformation. The MongoDB stream importer transforms the match metadata stream element given in (a) to the match metadata item given in (b). The player and team names are anonymized for privacy reasons.

```
Key:  751690
Offset:  54
Value:  {"atomic":true, "generationTimestamp":"3425349", "positions":
        [{"x":-23.849, "y":-23.994},{"x":-9.173,"y":-22.046}],
        "objectIdentifiers":["B8","B4"], "groupIdentifiers":["B"],
        "payload":{ "length":14.804718166854782, "velocity":14.317909252277351,
        "angle":7.560885213560285, "direction":"forward", "packing":3,
        "@type":".streamTeam.football.SuccessfulPassEventStreamElementPayload"}}
```

(a) Successful Pass Event Stream Element as Listed in the Cluster Monitor (cf. Figure 8.7)

```
{
     "_id" :  ObjectId("5e43d977e6b91f0fbefd9e61"),
     "type" :  "successfulPassEvent",
     "matchId" :  "751690",
     "ts" :  192962,
     "videoTs" :  3423,
     "xyCoords" :  [ [-23.849,-23.994], [-9.173,-22.046] ],
     "zCoords" :  [0.0, 0.0],
     "playerIds" :  ["B8", "B4"],
     "teamIds" :  ["B"],
     "additionalInfo" :  {
          "length" :  14.8047181668548,
          "velocity" :  14.3179092522774,
          "angle" :  7.56088521356028,
          "direction" :  "forward",
          "packing" :  3
     }
}
```

(b) Successful Pass Event Data Item Stored in the Events Collection of the MongoDB Instance

Figure 9.13 Sample Successful Pass Event Transformation. The MongoDB stream importer transforms the successful pass event stream element given in (a) to the successful pass event data item given in (b).

item is illustrated in Figure 9.13.

Since both calculations require information which is shipped in the match metadata stream element, the match metadata stream element of a match has to be processed by the MongoDB stream importer before event, state, and statistics stream elements can be transformed into data items. If an element of an event, state, or statistics stream which belong to a match whose match metadata stream element has not been processed yet is next to be processed – this happens usually only when iterating through the elements retrieved in the first pull the MongoDB stream importer performs after a new match started – this event, state, or statistics stream element is added to a waiting list whose content is iterated and processed (if the corresponding match metadata stream element has been processed in the meantime) every time when all data stream elements of a pull request have been processed.

After all data stream elements which have been retrieved in the last pull request are processed or added to the waiting list and the waiting list is iterated the resulting data items are inserted into the corresponding collections. This is done batch-wisely using the `insertMany(...)` function of the MongoDB's Java driver [Mon17c]. In consequence, all analysis results produced in StreamTeam-Football's analysis workflow and the match metadata are inserted efficiently but still almost immediately after they were generated into the MongoDB instance and are thus already available during the match for offline activities such as video retrieval (see Section 9.4.3).

We want to highlight that the current version of StreamTeam-Football's MongoDB stream importer has to be modified only if StreamTeam-Football's database design (see Section 9.4.1) is modified. If new events, states, or statistics are detected, calculated, and generated, respectively, in StreamTeam-Football's analysis workflow and even if completely new analysis workflows (e.g., an ice hockey analysis workflow) are implemented and executed the MongoDB stream importer does not have to be modified or reconfigured.

9.4.3 SportSense

In order to given an idea of what type of offline activities can be performed on the basis of the persistently stored analysis results in the MongoDB instance we will briefly present SportSense. In a nutshell, SportSense is an offline team sports video retrieval system which is intended to be used by coaches or match analysts to find characteristic video scenes in matches by means of sketch-based queries.

The original monolithic version of SportSense which did not use a MongoDB instance yet was presented in [AS13a; AS13b; AS14a; AS14b]. Later, a new football version and a new ice hockey version of SportSense have been implemented in two student projects [Lob17; Rau17] and published in [PAL+18]. Both versions shared the same new architecture consisting of a Web client, a MongoDB instance, and a MongoDB REST proxy.

As presented in [PRS+18], this new architecture enables connecting SportSense with StreamTeam to form an integrated analysis infrastructure. More precisely, StreamTeam-Football and SportSense-Football can be combined to an integrated football analysis infrastructure. Figure 9.14 depicts the architecture of this infrastructure. SportSense-Football uses the data items which StreamTeam-Football's MongoDB stream importer stored in the collections of the MongoDB instance as video tags. Especially the atomic events which are

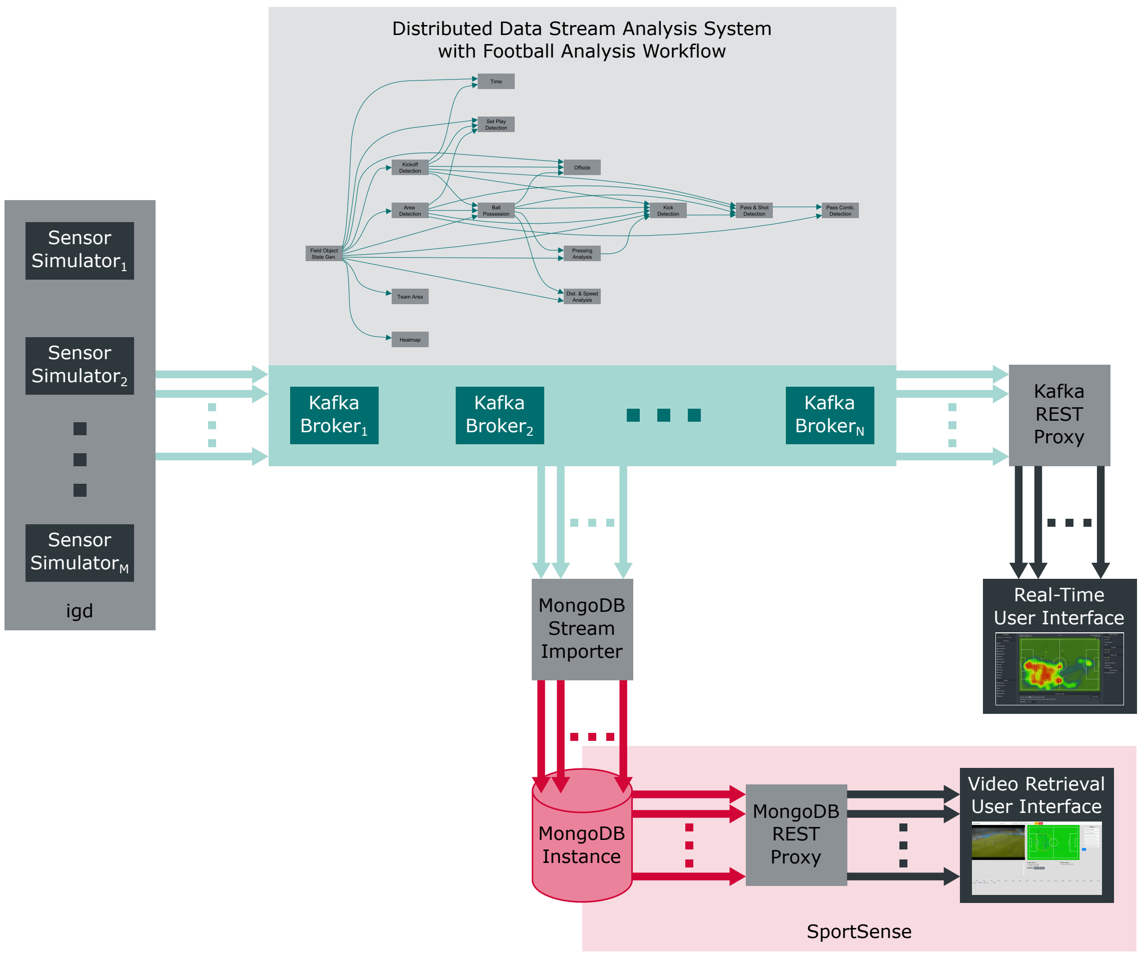
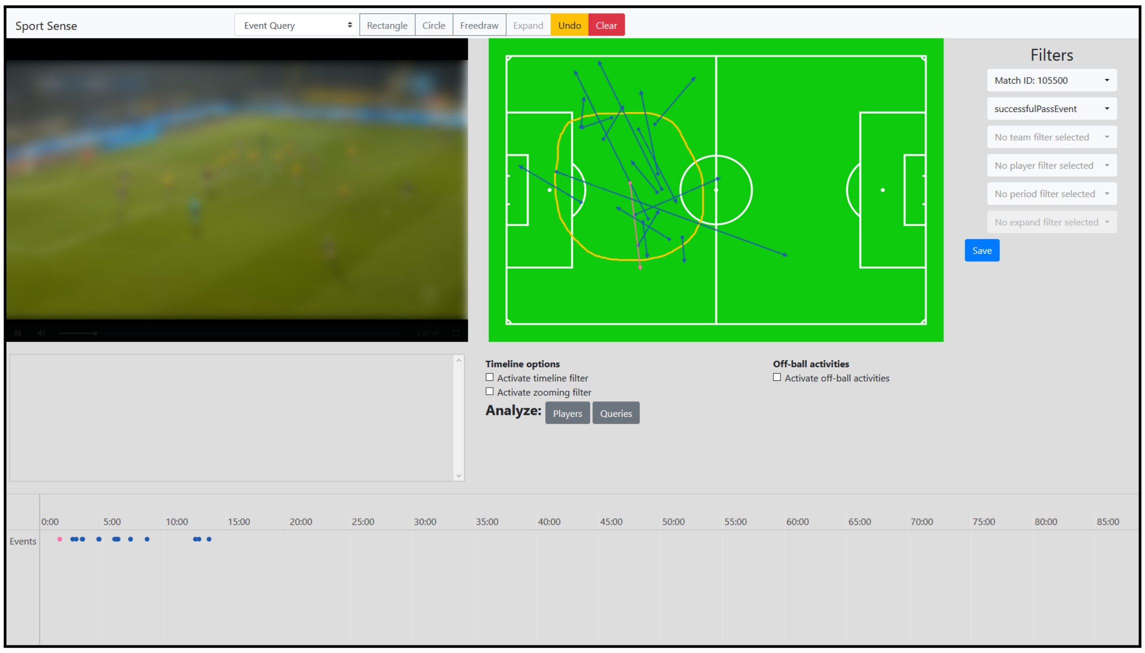


Figure 9.14 Architecture of the Integrated Football Analysis Infrastructure. Except for the workers of StreamTeam-Football's analysis workflow which are conceptual components that specify the analysis subtasks performed by Samza tasks, each box represents a component of the integrated football analysis infrastructure. The arrows illustrate data transfer between the components. Each light-mint arrow visualizes that the elements of a data stream are pushed to or pulled from Kafka brokers. Each dark-mint arrow illustrates that elements of a data stream are transfered between the Samza tasks which perform the analysis subtasks specified by two workers with a hybrid communication model using the Kafka brokers as the communication proxies. The dark-gray arrows visualize that data is fetched from the Kafka REST proxy or the MongoDB REST proxy via their REST APIs. The red arrows illustrate that data items are stored in or retrieved from the MongoDB instance. The light-red box is used to group SportSense's components and thus highlights that the MongoDB instance is shared by StreamTeam-Football and SportSense.



Event Query
Match ID: 105500
successfulPassEvent
No team filter selected
No player filter selected
No period filter selected
No expand filter selected
☐ Activate timeline filter
☐ Activate zooming filter
☐ Activate off-ball activities

Figure 9.15 SportSense's Web Client. The screenshot shows the result of a query for all successful passes which are kicked in the orange area. Each successful pass is visualized as an arrow on the field and listed in the time line. When a line or a point in the time line is clicked the video scene which shows the pass is played. The query has been performed while the match replay was still in progress (approximately after the first 15 minutes). The video area is blurred for privacy reasons.

stored as data items in the events collection and the player and ball trajectories which can be deduced from the field object state data items stored in the states collection are helpful for retrieving characteristic video scenes. The video retrieval user interface (see Figure 9.15) is a Web client which enables the coach to retrieve characteristic video scenes by means of drawing sketches and setting filters. The MongoDB REST proxy translates query-specific HTTP requests sent by the Web client to MongoDB queries and converts the MongoDB results to the final results which it sends back to the Web client (see [PRS$^+$18] for more details).

We want to highlight that SportSense has been improved since it was first connected with StreamTeam-Football. The football and the ice hockey version of SportSense have been combined to a unified version of SportSense which supports multiple team sports. Besides football and ice hockey, this unified SportSense version has been also used to retrieve characteristic video scenes in DotA2 matches [Zum19]. Moreover, new features have been added [Rau19; SJR$^+$19; SJP$^+$20] without loosing SportSense's compatibility with StreamTeam-Football.

10 Evaluation

In this chapter, we will present the evaluation of our implementation which we have presented in Chapter 8 and Chapter 9.

The abstractions and extensions which StreamTeam's generic data stream analysis system adds to Apache Samza [NPP+17] are valuable since they assist worker developers without a profound software engineering background (e.g, football match analysts) and introduce support for ingestion time (see Section 8.2). However, the abstractions and extensions do not alter the performance and scalability of the data stream analysis system. Hence, we argue that measuring the performance and analyzing the scalability of StreamTeam's generic data stream analysis system prototype with dedicated artificial workflows which are designed to simulate different workflow characteristics as usually done when evaluating a system would only evaluate Samza [NPP+17] and Kafka [KNR11] and is thus out of the scope of this thesis. In our opinion, it is more enlightening to investigate the performance while a non-trivial real-time team collaboration analysis application (as envisioned in Part I) which is implemented on top of our generic StreamTeam infrastructure performs its analysis workload in order to show that we solved the performance-related challenges posed in Section 1.1 sufficiently. Therefore, we decided to evaluate StreamTeam's generic data stream analysis system by means of performing measurements while StreamTeam-Football analyzes football matches.

Our quantitative evaluations have two objectives. First, we aim to show that the theoretical statements about the (un)ambiguity of the timestamps which we have made in our generic stream time model are correct. For this purpose, we will investigate the differences between the timestamps which components of StreamTeam-Football assign to the same data stream elements (see Section 10.3.1). Second, we aim to confirm that StreamTeam-Football is able to analyze not only a single but even multiple concurrent matches in real-time and that StreamTeam's data stream analysis system scales with respect to the number of processed and emitted data stream elements. To meet this objective we will measure the performance of StreamTeam-Football for different configurations and investigate how StreamTeam-Football's performance changes when the number of concurrent matches is increased (see Section 10.3.2).

However, before we present these quantitative evaluations we will first show that StreamTeam-Football is really a non-trivial real-time team collaboration analysis application which fulfills some actual real-world analysis demands and not just another toy application with an artificial workflow. For this purpose, we evaluate the quality of StreamTeam-Football's analysis results by comparing them to the Opta F24 data feed [Opt20a; Opt20b], the football event dataset which is prevalent in industry (see Section 10.2).

10.1 General Setup

In this section, we will describe the general evaluation setup which is the same for all qualitative and quantitative evaluations.

10.1.1 Deployment

For evaluating StreamTeam(-Football) we deploy the integrated football analysis infrastructure depicted in Figure 9.14 on a cluster consisting of six homogeneous machines (see Figure 10.1). Each machine is equipped with an Intel Core i7-4770 CPU, 32 GB RAM, and two SSDs[1]. All machines run Ubuntu 16.04.3[2] as the operating system.

One machine is the master node. The master node runs the primary and secondary HDFS namenode [Had19][3], the YARN resource manager [VMD+13], a Kafka broker [KNR11], a Zookeeper server [HKJ+10][4], the Kafka REST proxy (see Section 8.3.1), a MongoDB instance (see Section 9.4.1), the MongoDB stream importer (see Section 9.4.2), SportSense's MongoDB REST proxy (see Section 9.4.3), a Prometheus instance [Pro20], and an instance of the Samza

[1] SSD model: ADATA XPG SX900, 256GB

[2] Kernel version: Linux 4.4.0-174-generic

[3] Note that HDFS is only used to store the JAR file that contains the specifications of the StreamTeam workers which form StreamTeam-Football's analysis workflow but not to store or read any other data.

[4] Note that Zookeeper is required by YARN and Kafka but not used to assign sequence numbers using the shared counter approach which we have presented in Section 6.2.1.1.

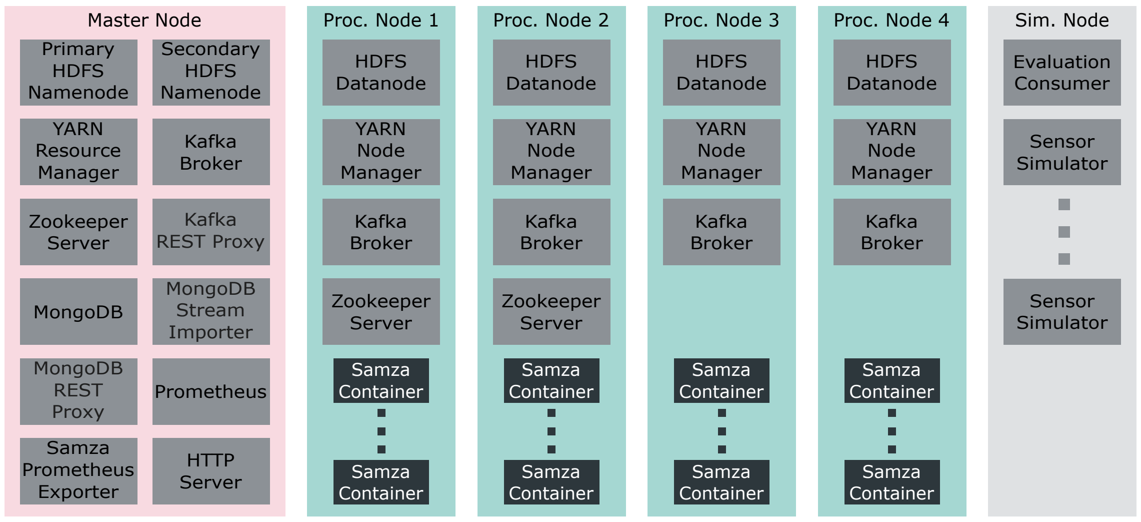


Figure 10.1 **Evaluation Deployment.** The red, mint, and light-gray boxes represent the different cluster nodes. Each gray box represents a component which is executed directly by a cluster node. The dark-gray boxes represent Samza containers which are deployed and executed as YARN containers.

Prometheus Exporter [Mov20][5]. In addition, the master node runs an Apache HTTP server [Apa20] and hosts the code for the Web clients. However, note that the major workload introduced by StreamTeam-Football's Web client (see Section 9.3) is performed by the machine which accesses this Web client and thus not by the cluster machines since this Web client is implemented solely with client-side programming languages. Moreover, note that neither StreamTeam-Football's nor SportSense's Web client is accessed during our evaluations. Hence, the Kafka REST proxy buffers data stream elements but is never queried by StreamTeam-Football's Web client and the MongoDB instance is filled by the MongoDB stream importer but never queried by SportSense's MongoDB REST proxy.

Four machines are processing nodes which execute the Samza containers. For this purpose, each processing node runs a YARN node manager [VMD+13]. Moreover, each processing node runs a Kafka broker [KNR11] and a HDFS datanode [Had19]. Two of the processing nodes further run a Zookeeper server [HKJ+10].

The sixth machine is a dedicated simulation node. The simulation node executes all instances of the sensor simulator (see Section 9.1). Moreover, this machines runs the evaluation consumer which is not depicted in Figure 9.14 in order to calculate latencies for the performance evaluation (see Section 10.3.2).

[5] The Prometheus instance and the instance of the Samza Prometheus Exporter are used to acquire Samza metrics for our performance evaluation (see Section 10.3.2.1)

10.1.2 Input

In order to generate the raw input streams for our evaluations we replay a TRACAB Optical Tracking dataset [Chy20c] of a regular European top league match which took place in autumn 2018. This dataset contains a new position for each player and the ball every 40 milliseconds. As discussed in more detail in [PGR+19] the positions are not completely free from errors but exhibit a high quality.

Since one limitation of StreamTeam-Football is that some analyses work only during the first halftime or until the first player is substituted (see Section 9.2.2), we replay only the first halftime of the match in which no substitution took place. More precisely, we stop all sensor simulators 46 minutes after they were started. Due to the artificial waiting time at the beginning which is introduced to guarantee a synchronous replay start (see Section 9.1) approximately the first 45 minutes and 20 seconds of the match and thus the complete regular first halftime and the first seconds of its overtime are replayed.

In order to investigate how the performance changes if the number of concurrently analyzed matches varies (see Section 10.3.2) we simply replay the same match multiple times in parallel with different match identifiers. We decided to do so since we have only one high-quality TRACAB Optical Tracking dataset and a few ball-enriched LPM datasets [SPF04; Inm20] from the SFISM which exhibit an occasionally unusable ball position quality (even after a manual post-match cleaning process) and since optimizing the quality of the tracking data is orthogonal to our research. Nevertheless, we want to highlight that StreamTeam-Football supports also to analyze multiple different matches in parallel, even if the positions were tracked with different technologies. We have successfully tested this by concurrently replaying and analyzing the above mentioned match and a different match which was tracked at the SFISM with the LPM system and the corresponding ball tracking system.

10.2 Qualitative Evaluation

As presented in Section 9.2, StreamTeam-Football detects, calculates, and generates a plethora of events, states, and statistics, respectively. The meaningfulness of some analyses is obvious. For instance, it is clear that it is meaningful to detect ball possession changes, passes, shots, and set plays and to generate the corresponding statistics when analyzing a football match. In addition, StreamTeam-Football has been developed not only by computer scientists but in close collaboration with sports scientists from the SFISM. That is, the sports scientists from the SFISM have provided us with definitions of events, states, and statistics which we used to develop algorithms to detect, calculate, and generate them. Because of this, many analyses which StreamTeam-Football performs can be mapped to concepts which are extracted from interviews with football coaches [SRP+19]. Moreover, the algorithm to calculate the pressing index in the pressing analysis worker (see Section 9.2.1.8) is even a direct implementation of a formula developed by the SFISM [Rum20].

However, this confirms only that the analysis results which StreamTeam-Football aims for are meaningful. To show that StreamTeam-Football is really a non-trivial real-time team collaboration analysis application which fulfills the analysis demands of football coaches, match analysts, and sports scientists we have to further show that StreamTeam-Football's analysis results are correct. For this purpose, we evaluate the quality of its analysis results.

10.2.1 Method

In a nutshell, we evaluate the quality of StreamTeam-Football's analysis results by comparing events which StreamTeam-Football detects with the corresponding events in the Opta F24 data feed [Opt20a; Opt20b] of the same match.

Note that Opta F24 data feeds are not perfect and can thus not be regarded as a ground truth. At least the data feed we have for our evaluation match contains some events with slightly shifted timestamps and some events with wrong positions. Moreover, there are some events in the data feed which are controversial as some coaches and/or match analysts might disagree with the event categorization – a fact which cannot be prevented as there are no clear, universally accepted definitions in football which cover all corner cases. Nevertheless, Opta F24 data feeds are accepted and prevalent in industry. For instance, as mentioned in [Opt20b], Opta F24 data feeds are used by Sky Sports, one of the most famous sports broadcasters. Therefore, and since there is no ground truth, we argue that it is the best option to use the non-perfect Opta F24 data feed in order to evaluate the quality of StreamTeam-Football's analysis results.

Unfortunately, not every analysis result is suited for a qualitative evaluation. Instead, only those analysis results of StreamTeam-Football which are also contained in the Opta F24 data feed in a comparable form are candidates for a qualitative evaluation. For the qualitative evaluation in this thesis, we have identified six atomic events, namely successful passes, interceptions, freekicks, cornerkicks, throwins, and goalkicks (see Table 10.1). These events are detected by StreamTeam-Football and contained in the Opta F24 data feed with similar temporal and spatial information.

10.2.1.1 Event Extraction

In StreamTeam-Football all data about an atomic event are published in an element of an event type specific data stream (see Section 4.4). For instance, the data of each successful pass event are published in an element of the successful pass event stream (see StreamTeam-Football column in Table 10.1). Moreover, each atomic event is stored as a single data item in the events collection of a MongoDB instance with the type set to the data stream name (see Section 9.4).

In order to access all relevant events which StreamTeam-Football detects we make use of this persistent storage. We analyze the first halftime of our match in StreamTeam-Football as described in Section 10.1.[6] Subsequently, we issue for each event type a query to the MongoDB instance using MongoDB's Java driver [Mon17e] and store some selected temporal and spatial information for each event in a line of an event type specific file. More precisely, we store the generation timestamp of the event as well as the start and end position if the event is a successful pass event, the end position if the event is an interception event[7], and the position of the ball (when it was kicked/thrown) if the event is a freekick, cornerkick, throwin, or goalkick event.

The Opta F24 data feed is provided as an XML document. In this document, each Opta event (*oe*) is encoded as an event element with a type identifier, a timestamp, a period identifier, an x-coordinate, a y-coordinate, an outcome, and an arbitrary number of qualifier child elements ($oe.Q$). Each qualifier element (q) has a qualifier identifier and optionally a value.

For comparing the events which StreamTeam-Football detects with the events contained in the Opta F24 data feed we extract all successful passes, interceptions, freekicks, cornerkicks, throwins, and goalkicks from the XML document. More precisely, we store the same temporal and spatial information as we do for the events detected by StreamTeam-Football in event type specific

[6] Note that, as discussed in Section 9.2.2, StreamTeam-Football does not produce deterministic analysis results. Instead, the analysis results vary for different analysis attempts. However, we decided to perform the qualitative evaluation only for the events that are detected in a single analysis attempt as the analysis results vary only very little. Moreover, we accept that stopping the match replay before the overtime of the first halftime ended might slightly worsen our qualitative evaluation results since less than a minute is missing.

[7] The start position of an interception event is omitted as this information is not available in the Opta F24 data feed.

Event	StreamTeam-Football	Opta F24 data feed
Successful pass	$dse.ds.name =$ "successfulPassEvent"	$oe.typeId \in \{1,2\} \wedge oe.outcome = 1 \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 140) \wedge (\exists q_2 \in oe.Q : q_2.qualifierId = 141) \wedge (\nexists q_3 \in oe.Q : q_3.qualifierId \in \{5,6,107,124,236\})$
Interception	$dse.ds.name =$ "interceptionEvent"	$oe.typeId = 8 \vee (oe.typeId \in \{1,2\} \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 236) \wedge (\exists q_2 \in oe.Q : q_2.qualifierId = 140) \wedge (\exists q_3 \in oe.Q : q_3.qualifierId = 141) \wedge (\nexists q_4 \in oe.Q : q_4.qualifierId \in \{5,6,107,124\}))$
Freekick	$dse.ds.name =$ "freekickEvent"	$oe.typeId \in \{1,2\} \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 5) \wedge (\nexists q_2 \in oe.Q : q_2.qualifierId \in \{6,107,124,236\})$
Cornerkick	$dse.ds.name =$ "cornerkickEvent"	$oe.typeId \in \{1,2\} \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 6) \wedge (\nexists q_2 \in oe.Q : q_2.qualifierId \in \{5,107,124,236\})$
Throwin	$dse.ds.name =$ "throwinEvent"	$oe.typeId \in \{1,2\} \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 107) \wedge (\nexists q_2 \in oe.Q : q_2.qualifierId \in \{5,6,124,236\})$
Goalkick	$dse.ds.name =$ "goalkickEvent"	$oe.typeId \in \{1,2\} \wedge (\exists q_1 \in oe.Q : q_1.qualifierId = 124) \wedge (\nexists q_2 \in oe.Q : q_2.qualifierId \in \{5,6,107,236\})$

Table 10.1 Event Extraction Conditions. Conditions which a data stream element (*dse*) and an Opta F24 data feed element (*oe*) has to fulfill in order to be regarded as a successful pass, interception, freekick, cornerkick, throwin, or goalkick event.

files. Since we analyze only the first halftime with StreamTeam-Football, we extract only the events of the first halftime. For doing so, we ignore all event elements whose period identifier is not 1. To distinguish between the event types we make use of the type identifiers, the outcome, and the qualifier identifiers. The Opta column in Table 10.1 lists the conditions which an event element has to fulfill to be regarded as a candidate event and thus stored in a file. For instance, an event element is regarded as a throwin event and stored in the Opta throwin event file if its type identifier is 1 or 2, if it has a qualifier child element whose qualifier identifier is 107, and if it has no qualifier child element whose qualifier identifier is 5, 6, 124, or 236.

As the timestamps and the positions have a different format we transfer them into the format used by StreamTeam-Football. That is, we convert the event element timestamps which are provided in the ISO 8601 date format [ISO88] into milliseconds since the start of the match by means of subtracting the timestamp of a dedicated halftime start event that is contained in the Opta F24 data feed.[8] Moreover, we extract additional end positions from the values of the qualifier child elements whose qualifier identifier is 140 and 141, and transform all positions into the coordinate system used by StreamTeam-Football.

10.2.1.2 Event Comparison

To perform the qualitative evaluation we compare the extracted events separately for each event type. For doing so, we first read the extracted StreamTeam-Football and Opta event data from the two generated files. Subsequently, we iterate through the events detected by StreamTeam-Football in generation time order and check if there is a matching Opta event. This is done by iterating through the Opta events (again in timestamp order) and comparing each Opta event with the current StreamTeam-Football event. Opta events which were already the match for a previous StreamTeam-Football event with a lower generation timestamp are skipped in order to prevent that the same Opta event is used as the matching event for two StreamTeam-Football events.

An Opta event matches a StreamTeam-Football event and thus the check in line 8 of Algorithm 10.1 is passed if the following conditions hold: First, the difference between their timestamps has to be lower than a given time threshold. Moreover, the Euclidean distances between all positions which are extracted and stored into the files (see Section 10.2.1.1) have to be lower than a given distance threshold. That is, for successful passes events the distance threshold has to be met for the start and end position of the pass, for interception events the distance threshold has to be met for the end position (i.e., for the position where the ball was received), and for the set play events (i.e., the freekick, cornerkick, throwin, and goalkick events) the distance threshold has to be met for the start position (i.e., the position where the ball was kicked/thrown).

We calculate three metrics, namely the correct detection percentage, the wrong detection percentage, and the missed detection percentage. In doing so we regard an event that is detected by StreamTeam-Football to be a *correct detection* if there is a matching event in the Opta F24 data feed and to be a *wrong detection* if this is not the case. Moreover, we regard an event that is contained in the Opta F24 data feed to be a *missed detection* if this event was no match for any event detected by StreamTeam-Football.

The algorithm for performing the event comparison and calculating the metrics is the following:

[8] The Opta F24 data feed contains event elements whose type identifier is 32 which announce the start of the halftimes.

Algorithm 10.1 Event Comparison Loop

Input: Events (of a given type) detected by StreamTeam-Football (*STE*)
Events (of the same type) contained in the Opta F24 data feed (*OE*)
Output: Metrics tuple

```
1: $numDetections \leftarrow |STE|$
2: $numOptaEvents \leftarrow |OE|$
3: $numCorrectDetections \leftarrow 0$
4: $OE_2 \leftarrow OE$    ▷ Makes a deep copy of $OE$.
5: for all $ste \in STE$ do    ▷ Iterates over events detected by StreamTeam-Football in generation time order.
6:     for all $oe \in OE$ do    ▷ Iterates over events contained in the Opta F24 data feed in timestamp order.
7:         if $oe \in OE_2$ then    ▷ If $oe$ was not already the match for another event detected by StreamTeam-Football.
8:             if $oe$ matches $ste$ then    ▷ Temporal and spatial comparison
9:                 $numCorrectDetections \leftarrow numCorrectDetections + 1$
10:                $OE_2 \leftarrow OE_2 - oe$
11:                break Opta event loop
12:            end if
13:        end if
14:    end for
15: end for
16: $numWrongDetections \leftarrow numDetections - numCorrectDetections$
17: $numMissedDetections \leftarrow |OE_2|$
18: $correctDetectionPercentage = \frac{numCorrectDetections}{numDetections}$
19: $wrongDetectionPercentage = \frac{numWrongDetections}{numDetections}$
20: $missedDetectionPercentage = \frac{numMissedDetections}{numOptaEvents}$
21: return $\langle correctDetectionPercentage, wrongDetectionPercentage,$
22:        $missedDetectionPercentage \rangle$
```

In order to investigate the spatial and temporal quality of the analysis results more deeply we perform the event comparison not only once for each event type for a single time threshold and a single distance threshold but for multiple time and distance threshold combinations. More precisely, we perform Algorithm 10.1 once for each element of the Cartesian product of the event types set {"successfulPass", "interception", "freekick", "cornerkick", "throwin", "goalkick"}, the time threshold set $\{1\,\mathrm{s}, 2\,\mathrm{s}, 3\,\mathrm{s}, 4\,\mathrm{s}, 5\,\mathrm{s}\}$, and the distance threshold set $\{1\,\mathrm{m}, 3\,\mathrm{m}, 5\,\mathrm{m}, 7\,\mathrm{m}, 9\,\mathrm{m}, \infty\}$. The distance threshold ∞ is equivalent to having no distance threshold at all and contained in the distance threshold set to evaluate only the temporal quality of the analysis results while ignoring the spatial accuracy. We argue that it is reasonable to do so as wrong spatial assignments can happen fast, especially since the positions of the Opta events, as all other Opta event data, are captured manually by Opta employees [Nut18] while the positions of the StreamTeam-Football events are derived from the tracked positions. For instance, we argue that it makes sense to regard a freekick event in the Opta F24 data feed as the matching event for a freekick event detected by StreamTeam-Football if the assigned timestamps are almost the same even if the Euclidean distance between their positions is greater than nine meters. In contrast, we argue that an Opta event whose timestamp differs by more than five seconds from the timestamp of an event that is detected by StreamTeam-Football is definitely disqualified for being a matching event.

10.2.2 Results

In this section, we will present and discuss the results of our qualitative evaluation, or more precisely of the event comparison for the six selected atomic events.

10.2.2.1 Successful Passes

The qualitative evaluation results for the successful passes are given in Figure 10.2. Figure 10.2(a) and Figure 10.2(b) show the correct detection percentages and the missed detection percentages, respectively, for different time and distance threshold combinations.[9] In general, the results show that StreamTeam-Football's successful pass detection quality is not perfect but pretty good.

As expected the higher the time and distance thresholds are the better are the comparison results and thus the higher are the correct detection percentages and the lower are the wrong and missed detection percentages. This trend follows directly from our event comparison method (see Section 10.2.1.2) and is thus observable in the qualitative evaluation results for all event types.

If the time threshold is greater than or equal to two seconds and the distance threshold is greater than or equal to five meters, the correct detection percentage is at least 51 % and the missed detection percentage is at most 51 %. If the thresholds are set to four seconds and nine meters, 75 % of StreamTeam-Football's successful pass detections are correct and only 27 % of the Opta events are missed. If the spatial accuracy is ignored (i.e., if the distance threshold is set to ∞), even up to 92 % of StreamTeam-Football's detections are correct and down to 11 % of the Opta events are missed.

[9] We refrained from additionally depicting the wrong detection percentages as the wrong detection percentage is always 1 minus the correct detection percentage. That is the wrong detection percentage is for instance 29 % if the correct detection percentage is 71 %.

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.00	0.05	0.12	0.15	0.15	0.39
2 s	0.01	0.27	0.51	0.58	0.59	0.74
3 s	0.01	0.33	0.63	0.71	0.73	0.87
4 s	0.01	0.33	0.64	0.73	0.75	0.90
5 s	0.01	0.33	0.64	0.73	0.75	0.92

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	1.00	0.95	0.89	0.85	0.85	0.62
2 s	0.99	0.74	0.51	0.44	0.43	0.28
3 s	0.99	0.68	0.39	0.31	0.29	0.16
4 s	0.99	0.68	0.38	0.29	0.27	0.13
5 s	0.99	0.68	0.38	0.29	0.27	0.11

(b) Missed Detection Percentage

Figure 10.2 Qualitative Evaluation Results for Successful Pass Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the successful pass events for different time and distance threshold combinations. All values are rounded to two decimals.

10.2.2.2 Interceptions

Figure 10.3 lists the qualitative evaluation results for the interceptions. Already the first glance reveals that StreamTeam-Football exhibits a poor interception detection quality. If the spatial accuracy is not ignored, at most 5 % of the interceptions detected by StreamTeam-Football are correct and at least 82 % of the Opta events are missed.

However, one of the reasons for these bad results are the different event definitions which underly Opta's manual labeling and StreamTeam-Football's algorithmic detection. Opta defines an interception to be an intended pass which was intercepted by a player by moving into the pass and receiving or blocking[10] the ball [Opt18]. In contrast, StreamTeam-Football regards all passes which are received by a player of the opposing team as interceptions as long as the ball was not kicked in the defense zone while the player was attacked.[11] This is even the case if the pass was not intended and thus for instance if the ball was not actually kicked but jumped away to a player of the opposing team after blocking a pass of this team. In consequence, even if StreamTeam-Football's pass and shot detection worker (see Section 9.2.1.10) performed exactly as intended, the set of interceptions which are extracted from the Opta F24 data feed would be only a subset of the set of interceptions which StreamTeam-Football detects (see Figure 10.4). Besides actual wrong detections (e.g., two interceptions are detected wrongly if a wrong ball hit is detected when the ball was very close to an opposing player during a successful pass) this is another reason for the fact that only 22 interceptions are extracted from the Opta F24 data feed but 79 interceptions are detected by StreamTeam-Football (see Table D.1) and that the missed detection percentages are not as bad as the correct and wrong detection percentages.

Moreover, the fact that the results improve remarkably to up to 18 % correct detections and down to 36 % missed detections if the spatial accuracy is ignored point out that there is a severe mismatch between the positions which StreamTeam-Football derives from the tracking data and the positions which the employees of Opta assign during the manual event labeling process.

[10] If the ball is blocked, Opta labels the event as a blocked pass [Opt18]. However, we extract also these blocked passes as interceptions (see Section 10.2.1.1).

[11] StreamTeam-Football detects a clearance event if the ball was kicked in the defense zone while the player was attacked and received by an opposing player in another zone or left the field.

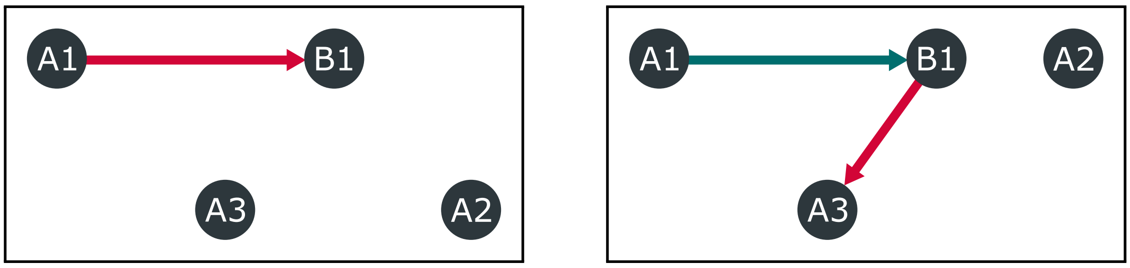
	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.01	0.03	0.05	0.05	0.05	0.13
2 s	0.01	0.03	0.05	0.05	0.05	0.15
3 s	0.01	0.03	0.05	0.05	0.05	0.15
4 s	0.01	0.03	0.05	0.05	0.05	0.18
5 s	0.01	0.03	0.05	0.05	0.05	0.18

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.95	0.91	0.82	0.82	0.82	0.55
2 s	0.95	0.91	0.82	0.82	0.82	0.45
3 s	0.95	0.91	0.82	0.82	0.82	0.45
4 s	0.95	0.91	0.82	0.82	0.82	0.36
5 s	0.95	0.91	0.82	0.82	0.82	0.36

(b) Missed Detection Percentage

Figure 10.3 Qualitative Evaluation Results for Interception Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the interception events for different time and distance threshold combinations. All values are rounded to two decimals.



(a) A1 kicks the ball and B1 receives it without moving into a pass

(b) A1 intends to pass the ball to A2 but B1 blocks the ball in a way that it jumps to A3

Figure 10.4 Different Interception Definitions. The gray dots visualize players of team A and B. The mint arrows reflect interceptions according to our and Opta's definition while the red arrows reflect events which are interceptions according to our definition but not according to Opta's definition.

10.2.2.3 Throwins

The qualitative evaluation results for the throwins are presented in Figure 10.5. Based on these results, StreamTeam-Football's throwin detection quality can neither be characterized as clearly good nor as clearly bad. If the spatial accuracy is taken into consideration, at most one third of StreamTeam-Football's throwin detections are correct and at least 46 % of the Opta events are missed. However, if the spatial accuracy is ignored the results improve to 57 % correct detections and only 8 % missed detections. This indicates again that there are severe mismatches between the positions which the Opta employees assign manually and the positions which StreamTeam-Football derives from the tracking data.

Moreover, these numbers (43 % wrong detections but only 8 % missed detections) mean that almost all 26 throwins which are extracted from the Opta F24 data feed are the match for a throwin event detected by StreamTeam-Football but that many of the 42 throwin events which StreamTeam-Football detected are wrong detections. We suppose that the main cause for the wrong detections is the fact that StreamTeam-Football detects a throwin whenever the ball enters the field and no other set play (e.g., a cornerkick) was recently detected. This approach works well if nothing unexpected happens. However, there are situations in which the ball leaves and re-enters the field without involving a throwin or a cornerkick. This is for instance the case if the ball enters the field behind the goal line, if the ball continues rolling after a foul, or if the player who should perform the throwin changes. For a human it is very easy to distinguish

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.02	0.14	0.19	0.19	0.19	0.31
2 s	0.02	0.17	0.24	0.29	0.29	0.48
3 s	0.02	0.17	0.24	0.29	0.29	0.50
4 s	0.02	0.17	0.24	0.31	0.33	0.57
5 s	0.02	0.17	0.24	0.31	0.33	0.57

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.96	0.77	0.69	0.69	0.69	0.50
2 s	0.96	0.73	0.62	0.54	0.54	0.23
3 s	0.96	0.73	0.62	0.54	0.54	0.19
4 s	0.96	0.73	0.62	0.50	0.46	0.08
5 s	0.96	0.73	0.62	0.50	0.46	0.08

(b) Missed Detection Percentage

Figure 10.5 Qualitative Evaluation Results for Throwin Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the throwin events for different time and distance threshold combinations. All values are rounded to two decimals.

between such a corner case and an actual throwin. In fact, some of these corner cases seem even obvious once they are discovered. However, covering all these corner cases is hard for an algorithmic approach, especially since throwins are more frequent than other set play events (such as freekicks, cornerkicks, and goalkicks) but still not that frequent that it is easy to test and tweak their detection algorithm. Nevertheless, we plan to fix as many of these corner cases in our future work (see Section 13.3).

10.2.2.4 Freekicks, Cornerkicks, and Goalkicks

Unfortunately the quantity of freekicks, cornerkicks, and goalkicks which are detected by StreamTeam-Football and extracted from the Opta F24 data feed is too low (all single-digit) to draw reliable conclusions about StreamTeam-Football's detection quality. This makes sense as there are not that many freekicks, cornerkicks, and goalkicks in a single halftime of a single match. In contrast, successful passes, interceptions, and throwins are much more frequent events in a football match. The exact quantities are given in Table D.1. For the sake of completeness, we present the qualitative evaluation result graphs for the freekicks, cornerkicks, and goalkicks in Figure D.1, D.2, and D.3. In a nutshell, the results indicate that StreamTeam-Football's cornerkick detection quality is promising but improvable, that its freekick detection quality is yet insufficient (especially with respect to the spatial accuracy), and that its goalkick detection quality is poor. We want to highlight that the weak results for the set play events do not reveal a problem of our general approach but only indicate that the algorithm of the set play detection worker needs to be improved – a task which we plan to face in our future work (see Section 13.3).

10.2.2.5 Conclusion

The qualitative evaluation results reveal that the spatial accuracy has some issues while the temporal accuracy is pretty good. The fact that the correct detection percentages for the one meter distance threshold are almost zero for all event types independent of the selected time threshold shows that there are almost no StreamTeam-Football events whose automatically derived position is very close to the manually assigned position of the corresponding Opta event.[12] Moreover, all percentages are improved remarkably when changing the distance threshold from nine meters to ∞ and thus from a high distance threshold to ig-

[12] The sole exceptions are the cornerkick events whose position is predefined by the football rules.

noring the spatial accuracy completely. In contrast, although increasing the time threshold from one to two seconds improves the percentages notably, further increasing the time threshold to three, four, and five seconds does not have the same effect as increasing the distance threshold.

Unfortunately, since we cannot compare the events which StreamTeam-Football detects with a ground truth as we have no ground truth but only with the events which we extract from the Opta F24 data feed, we are not able to make a reliable statement on the cause for the spatial inaccuracies. However, from reviewing the extracted event files (see Section 10.2.1.1) on a sample basis we suppose that at least some position mismatches are caused by the imperfect Opta F24 data feed.

Moreover, the results show that the detection quality for the events depends on their complexity and well-definiteness. The qualitative evaluation results for the successful passes show that StreamTeam-Football can achieve a high detection quality if there is a clear, unambiguous, universally accepted event definition. In contrast, the results for the interceptions indicate that the detection quality can be very bad if there are multiple different valid event definitions, at least if the detection quality is measured by comparing the detected events with events from another dataset which was created with another definition in mind. Moreover, the results for the throwins show that more complex events which engender the need to handle many corner cases can be problematic for the algorithmic detection approach which StreamTeam-Football follows especially if their frequency is too low to enable fast testing.

Despite the imperfections and the room for improvement which StreamTeam-Football's detection quality has shown in this evaluation, we conclude that the results are sufficient to confirm that StreamTeam-Football is a convincing proof-of-concept for analyzing football matches in real-time and thus a non-trivial real-time team collaboration analysis application which fulfills the analysis demands of football coaches, match analysts, and sports scientists.

10.3 Quantitative Evaluation

In this section, we will present our quantitative evaluations of StreamTeam's data stream analysis system. More precisely, we will describe the measurements which we made while StreamTeam-Football, our football analysis application which we showed in Section 10.2 to be a non-trivial real-time team collaboration analysis application, performs its analysis workload and discuss the results.

10.3.1 Processing Time Ambiguity

In our generic stream time model we have discussed properties of five different time notions. StreamTeam provides support for those time notions which are not only interesting theoretical concepts but realizable in practice.[13] Namely, these are the generation time, the ingestion time, and the processing time notion. In this section, we will confirm the theoretical statements which we have made in Section 6.1 about the (un)ambiguity of the timestamps when regarding a single data stream analysis system by means of performing measurements while StreamTeam-Football analyzes a match.

According to our model the generation timestamp of each data stream element is globally unambiguous as we define the generation timestamp of every data stream element to be either created by the clock of the raw input stream generating device which generated the (raw input stream) element or to be inherited but to be never created by a clock of a component of the data stream analysis system (see Section 6.1.1). This property is guaranteed in StreamTeam since also in StreamTeam no clock but only inheritance logic is used to assign generation timestamps to new data stream elements (see Section 8.2.3.1). Hence, there is nothing to be shown.

Moreover, as discussed in our model the ingestion timestamp of each raw input stream element is unambiguous when regarding only a single data stream analysis system since we define the ingestion timestamp to be assigned by the entry component of the data stream analysis system which received the raw input stream element (see Section 6.1.3). Also this property is guaranteed in StreamTeam as we regard the Kafka broker to which a raw input stream element is pushed as the entry component and leverage Kafka's log append time as the ingestion timestamp of the raw input stream element (see Section 8.2.3.1). Hence, there is again nothing to be shown.

In contrast, we have stated in our model that processing timestamps are only unambiguous when regarding a single processor but neither on a global scope nor on a data stream analysis system level since each processor assigns its individual processing timestamp to each data stream element when it starts processing this element using its local clock (see Section 6.1.3). In the remainder of this section, we will confirm this theoretical processing timestamp ambiguity statement by showing that the processing timestamps which the Samza tasks of StreamTeam-Football assign to the same data stream elements differ.

[13] Production and emission timestamps cannot be acquired in practice (see Section 6.1.2) and are thus not assigned in StreamTeam (see Section 8.2.3.1).

10.3.1.1 Method

According to our generic system model, the analysis subtask defined by a worker is performed by the processors which are assigned to this worker (see Section 5.3). In StreamTeam, each worker is implemented as a Samza job which is separated into Samza tasks, the equivalent to processors in our model (Section 8.2). Hence, we have to show that different Samza tasks assign different processing timestamps to the same data stream elements to confirm our theoretical statement that processing timestamps are only unambiguous when regarding a single processor but neither on a global scope nor on a data stream analysis system level.

In order to do so, we measure the difference between the processing timestamps which the Samza tasks that perform the analysis subtasks defined by StreamTeam-Football's analysis workers assign to the same data stream elements. More precisely, we investigate how much the processing timestamps which are assigned to field object state stream elements for the ball (hereafter referred to as ball object state stream elements) differ. We have chosen ball object state stream elements since the field object state stream is an input stream of all but three workers of StreamTeam-Football's analysis workflow and since using only the ball object state stream elements enables identifying each element by the combination of its key (i.e., the match identifier) and its generation timestamp.[14]

To show that the statement is even true if there is only one processor for each worker, we have configured Kafka in a way that each data stream has only one partition and Samza in a way that there is only one Samza container for each StreamTeam worker. Hence, there is only one Samza container for each worker in StreamTeam-Football's analysis workflow that executes the sole Samza task of this worker which processes all elements of the sole partition of every input stream. Moreover, we replay and analyze the first halftime of our match only once.

To measure the processing timestamp differences we log every processing timestamp that is assigned by a Samza task to a ball object state stream element. After the analysis, we extract the processing timestamps from the logs of the Samza containers of all eleven workers which have the field object state stream in their input stream set. The extracted processing timestamps are grouped by

[14] Note that each Samza task which processes the elements of a field object state stream partition processes the field object state stream elements for all objects (i.e., players and the ball) of a certain match since filtering by object identifiers is only done by the filter modules in the module graphs of the StreamTeam worker.

the ball object state stream elements they are assigned to. The resulting sets are then used to calculate the processing timestamp standard deviation for each ball object state stream element.

10.3.1.2 Results

Figure 10.6 shows the distribution of the standard deviations of the processing timestamps which are assigned to the ball object state stream elements over time. The standard deviation is not the same for all ball object state stream elements but varies slightly. On average the standard deviations of the processing timestamps which are assigned to the same ball object state stream element is 5.94 milliseconds. The fact that the standard deviation is not always zero shows that the same data stream element is not necessarily assigned with the same processing timestamp at every Samza task and thus confirms our theoretical processing timestamp ambiguity statement.

10.3.2 Performance

In this section, we will present the performance evaluation of StreamTeam-Football. The objective of this evaluation is to show that StreamTeam-Football is able to analyze not only a single but even multiple matches in real-time. Moreover, we investigate how StreamTeam-Football's performance changes when the number of concurrent matches is increased in order to evaluate how StreamTeam's data stream analysis system scales with respect to the number of processed and emitted data stream elements.

10.3.2.1 Method

The general evaluation setup which we have described in Section 10.1 is not only used for the qualitative evaluation (Section 10.2) and the processing time ambiguity evaluation (Section 10.3.1) but also for the performance evaluation. However, we do not only measure the performance for the simplest configuration in which only a single match is concurrently replayed and thus analyzed, in which each data stream has only a single partition, and in which there is only a single Samza container for each StreamTeam worker. Instead, we vary the number of concurrent matches, the number of partitions into which each data stream is split (`num.partitions`, see Section 8.2.2) and thus the number of Samza tasks per StreamTeam worker, and the number of Samza containers for each StreamTeam worker (`job.container.count`, see Section 8.2.5) to investi-

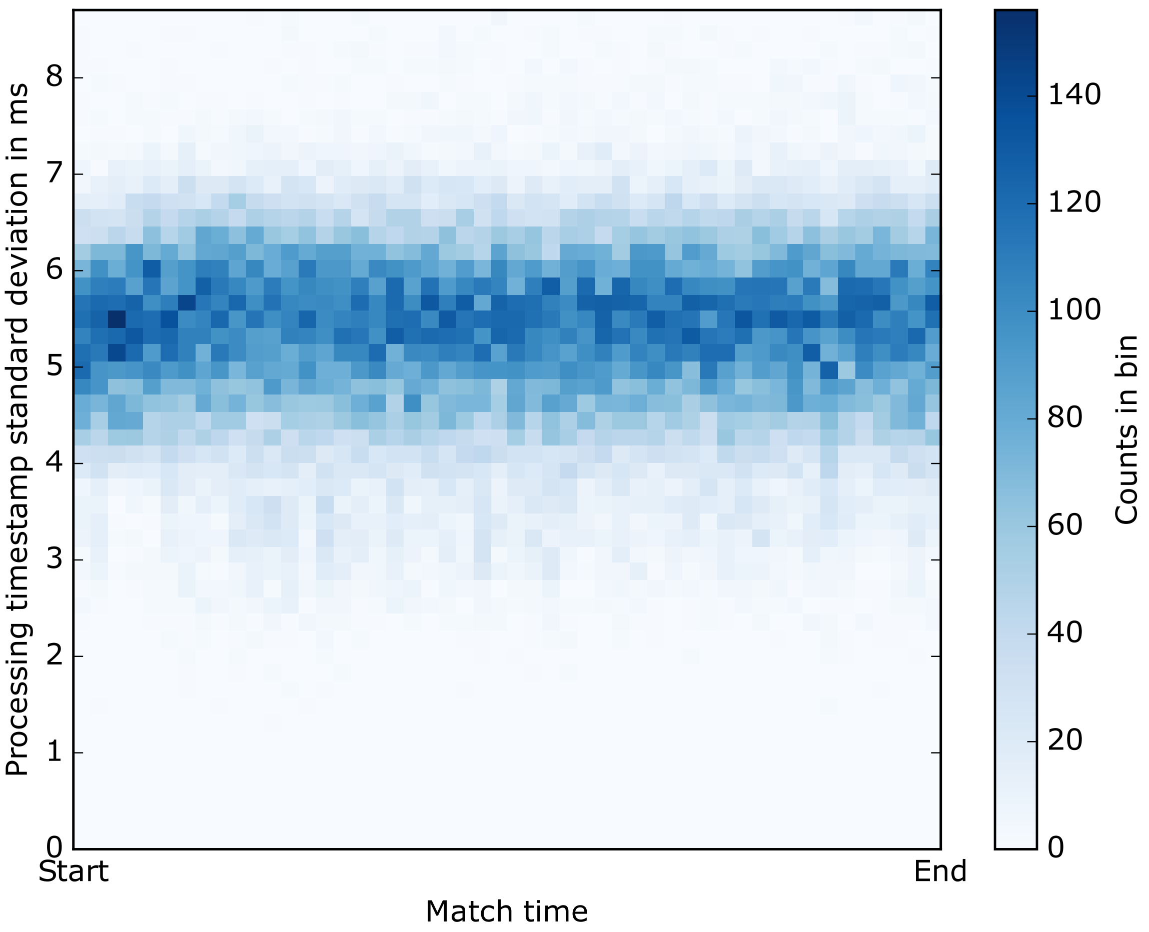


Figure 10.6 Processing Timestamp Standard Deviation Distribution over Time. The distribution over time is illustrated by means of a color-encoded 2D histogram. The x-axis is divided into 50 match time intervals (from the start to the end of the replay) and the y-axis is divided into 50 standard deviation intervals (from zero to the 99th percentile). The color of each box that results from this division represents the number of ball object state stream elements whose generation timestamp is in the corresponding match time interval and whose processing timestamp standard deviation is in the corresponding standard deviation interval.

gate the effect of these parameters on StreamTeam-Football's performance. More precisely, we measure the performance for one to five concurrent matches for all valid[15] combinations of one to three partitions and one to three containers.[16]

In order to assess the real-timeness of the analysis results, we measure the

[15] Note that combining two containers and one partition, three containers and one partition, as well as three containers and two partitions is not possible since the number of containers has to be smaller or equal to the number of tasks and thus to the number of partitions [Sam17e].

[16] Although it would be possible to configure the number of containers individually for each StreamTeam worker (see Section 8.2.5) we configure all StreamTeam workers to have the same number of containers in order to keep the number of combinations manageable.

overall latency between sending the last raw input stream element which contributed (directly or transitively) to the data of an event, state, or statistics stream element and receiving this output stream element at our evaluation consumer, a dedicated lightweight Kafka consumer whose only purpose is to log the system time after retrieving an element with the Consumer API of Kafka's original Java library [Kaf16]. This latency is measured for different data stream elements which are produced and emitted by different workers at different positions in the workflow. Namely, we measure the latency for (i) field object state stream elements for the ball emitted by the field object state generation worker (see Section 9.2.1.1), (ii) heatmap statistics stream elements for player A1 for the full game interval emitted by the heatmap worker (see Section 9.2.1.14), (iii) kick event stream element emitted by the kick detection worker (see Section 9.2.1.9), (iv) pass statistics stream elements for team B emitted by the pass and shot detection worker (see Section 9.2.1.10), and (v) pass sequence event stream elements emitted by the pass combination detection worker (see Section 9.2.1.11).

Since all sensor simulators and the evaluation consumer are deployed on the same machine (see Section 10.1), the latency could in theory be calculated by subtracting the system time when the last raw input stream element which contributed (directly or transitively) to the data of the output stream element was sent from the system time when the output stream element was received. The former is logged by the sensor simulators and the latter is logged by the evaluation consumer. However, we do not measure the exact latency in our evaluation using this approach since identifying which raw input stream elements contributed to the data of the output stream element is very hard due to the complex logic of the worker-specific modules.

Instead, we acquire a worst-case approximation of the exact latency. For doing so, we leverage the fact that the generation timestamp of each event, state, and statistics stream element is inherited (directly or transitively) from the raw input stream element with the greatest generation timestamp which contributed to the data of this element (see Section 6.1.1 and Section 8.2.3.1). As mentioned above, it is very hard to determine which elements from the set of raw position sensor data stream elements that have the same generation timestamp and the same key as the output stream element contributed to the data of the output stream element. However, it is guaranteed that at least one of these elements contributed to the data as the raw position sensor data stream elements are the only raw input stream elements which are sent and consumed during the match. In consequence, subtracting the minimum of all system times at which a raw position sensor data stream element with the same key and generation timestamp as the output stream element was sent from the system time when the output stream element was received results in a value which is greater than or equal to but never smaller than the exact latency.[17] Hence, the result of this subtraction is a worst-case approximation of the exact latency. Formally, we define this latency approximation which we use in our performance evaluation as follows:

Definition 10.1 Latency (Worst-Case Approximation)

The latency of the event, state, or statistics stream element *dse* is the difference between the system time when *dse* was received and the minimum of all system times at which a raw position data stream element dse_2 with the same key k and generation timestamp ts as *dse* was sent.

$$\begin{aligned} latency(dse) = receiveTime(dse) - \\ \min\{sendTime(dse_2) \mid dse_2.k = dse.k \wedge dse_2.ts = dse.ts \wedge \\ dse_2.ds.name = \text{“rawPositionSensorData”}\} \end{aligned}$$

After the analysis is finished for a certain configuration, we calculate the latency, or more precisely the worst-case latency approximation, for each of the above listed output stream elements by means of iterating over the logs of the sensor simulators and the evaluation consumer and subtracting the correct system times. The resulting latencies are then used to calculate the mean, the median, the 90th percentile, and the 99th percentile latency for each data stream subset[18]. Moreover, we calculate the standard deviations of the latencies and plot the empirical Cummulative Distribution Function (CDF) for each data stream subset.

Moreover, we fetch the values of some Samza metrics [Sam17g] which provide an insight into the performance of the individual Samza jobs and thus the individual StreamTeam workers of StreamTeam-Football's analysis workflow. More precisely, we issue some queries (see Table D.2) to the REST API of

[17] If only those raw position sensor data stream elements which were sent at the minimum system time contributed to the data of the output stream element, the result is equal to the exact latency. Otherwise, i.e., if other raw position sensor data stream elements with the same key and generation timestamp which were sent a little later contributed to the data of the output stream element, the result is slightly greater than the exact latency.

[18] We use the term data stream subset instead of data stream as the latency is not measured for all elements of all five data streams. For instance, the "ballObjectState" data stream subset comprises all field object stream elements for the ball.

Data stream subset	Mean	Median	90th pctl.	99th pctl.	Std. dev.
ballObjectState	24.10 ms	20.00 ms	49.00 ms	73.00 ms	14.86 ms
A1FullGameHeatmapStatistics	79.08 ms	73.00 ms	92.00 ms	561.24 ms	66.49 ms
kickEvent	31.74 ms	26.00 ms	59.00 ms	81.00 ms	16.45 ms
BPassStatistics	39.17 ms	31.00 ms	60.70 ms	88.71 ms	39.73 ms
passSequenceEvent	41.71 ms	37.00 ms	61.90 ms	89.39 ms	14.71 ms

Table 10.2 Latency Statistics for Single Match, Single Container, and Single Partition Configuration. The table lists the mean, the median, the 90th percentile, and the 99th percentile latencies as well as the latency standard deviations of the five data stream subsets. All values are rounded to two decimals.

the Prometheus instance which the Samza Prometheus Exporter feeds during the analysis with the measurements which Samza emits. Namely, we query the average duration of a process call (and thus the average duration for executing the single element processor graph) and the average duration of a window call (and thus the average duration for executing the window graph) over the last 45 minutes for each Samza job. Moreover, we query the aggregated number of process calls and window calls per second again as an average over the last 45 minutes for each Samza job.

10.3.2.2 Results

Table 10.2 lists statistics about the latencies which we measured when only a single match was replayed, each data stream had only a single partition, and there was only a single container for each StreamTeam worker. Moreover, Figure 10.7 displays the empirical CDFs for all five data stream subsets for the same configuration. As expected, the further back in the workflow the worker which emits an output stream element is positioned the higher is the latency of this output stream element. This makes sense since for instance a pass sequence event can only be detected after a successful pass event is detected by the preceding worker. The sole exception of this trend are the latencies of the heatmap statistics stream elements. We suppose that the elevated latencies of the heatmap statistics stream elements are caused by the transmission of these elements since

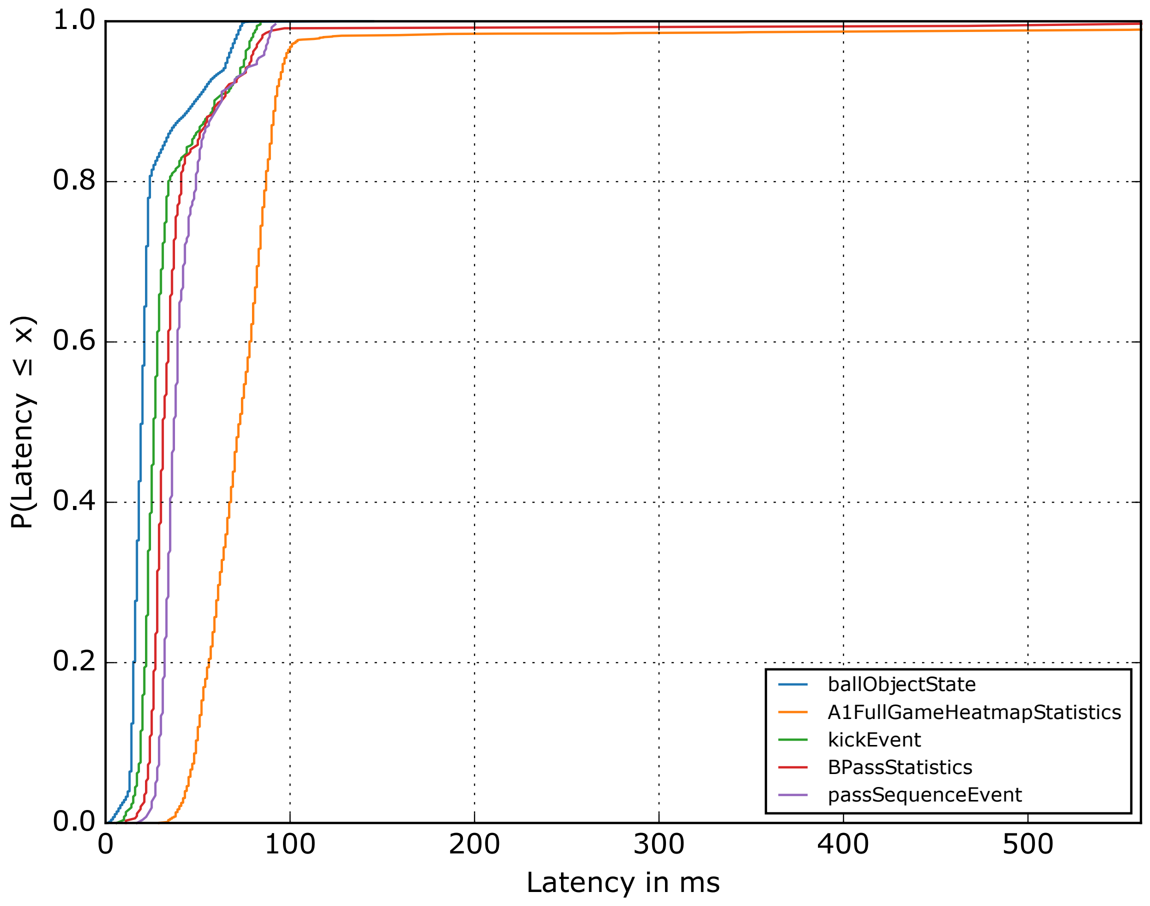


Figure 10.7 Latency CDFs for Single Match, Single Container, and Single Partition Configuration. The figure shows the empirical CDFs of the latencies measured for the elements of the five data stream subsets. For the sake of keeping the figure readable the CDFs are capped at the 99th percentile.

the generation of the heatmaps is quite fast[19] but the payload of the heatmap statistics stream elements is larger than the payload of all other output stream elements and all heatmap statistics stream elements are emitted at the same time (when the window graph of the heatmap worker is executed).

Moreover, we argue that the measured latencies are low enough to justify calling the analysis a real-time analysis, especially since these latencies do not only cover the analysis process in the data stream analysis system but the whole time span from sending the raw input stream elements to receiving the analysis results in output stream elements. Only a very small share of the output stream elements have a latency which is higher than 100 milliseconds. Since the football analysis scenario tolerates even that some analysis results are only available some seconds after the corresponding situation happened in the match (see Sec-

[19] As listed in Table D.5 and Table D.6 the average duration of a process and a window call during the analysis was 0.024 ms and 12.29 ms, respectively.

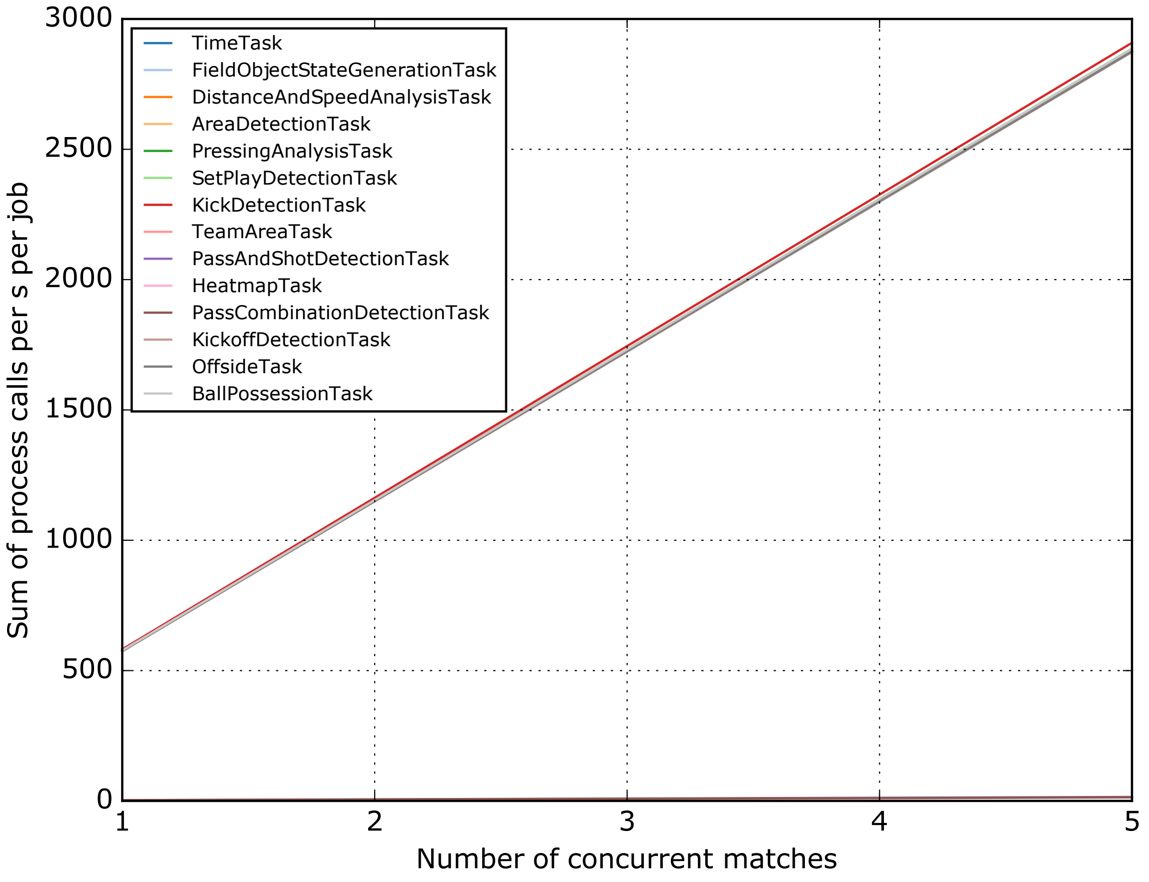


Figure 10.8 Average Aggregated Number of Process Calls per Second for Single Container and Single Partition Configurations. The figure shows the average aggregated number of process calls per second during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names in the legend end with "Task" since the legend shows the class names and ending the worker class file names with "Task" is Samza's naming convention [Sam17a]. Table D.3 lists all values.

tion 2.1), we argue that the real-time demands of the football analysis scenario are definitely met. Hence, the latency measurements for the simplest configuration (one match, one container, and one partition) confirm that StreamTeam-Football is able to analyze a single match in real-time.

Figure 10.8, 10.9, 10.10, and 10.11 show how the values of the four Samza metrics change in the one partition and one container configurations when the number of concurrently replayed and thus analyzed matches is increased. For each StreamTeam worker, the average aggregated number of process calls per second increases linearly with the number of matches but the average duration of each process call stays constant (except for some outliers). This is as expected since the process function and thus the single element processor graph of a StreamTeam worker is executed for single input stream elements (see Sec-

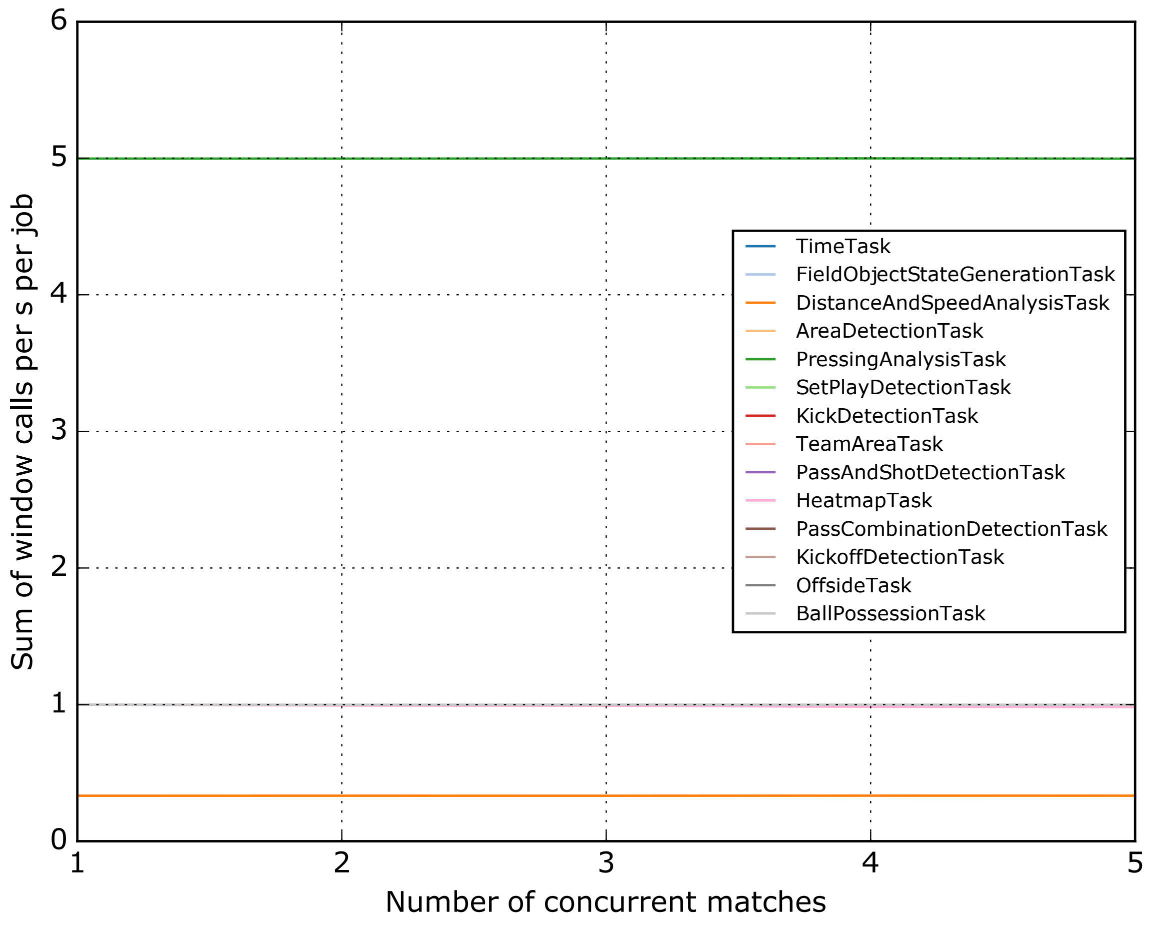


Figure 10.9 Average Aggregated Number of Window Calls per Second for Single Container and Single Partition Configurations. The figure shows the average aggregated number of window calls per second during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names in the legend end with "Task" since the legend shows the class names and ending the worker class file names with "Task" is Samza's naming convention [Sam17a]. Table D.4 lists all values.

tion 8.2.1) whose number scales linearly with the number of matches.[20] In contrast, the average aggregated numbers of window calls per second remain constant but the average durations of the window calls increase with the number of matches.[21] Also this is as expected since the window function of each Samza task is called with the same frequency no matter how many matches are analyzed in parallel but the more matches are analyzed in parallel the more keys are active and thus the more often the process functions of the successor modules of

[20] The more matches are replayed the more raw position sensor data stream elements are sent. Moreover, also the number of detected events, calculated states, and generated statistics and thus the number of event, state, and statistics stream elements scales linearly with the number of matches.

[21] Note that not every worker of StreamTeam-Football's analysis workflow has a timer.

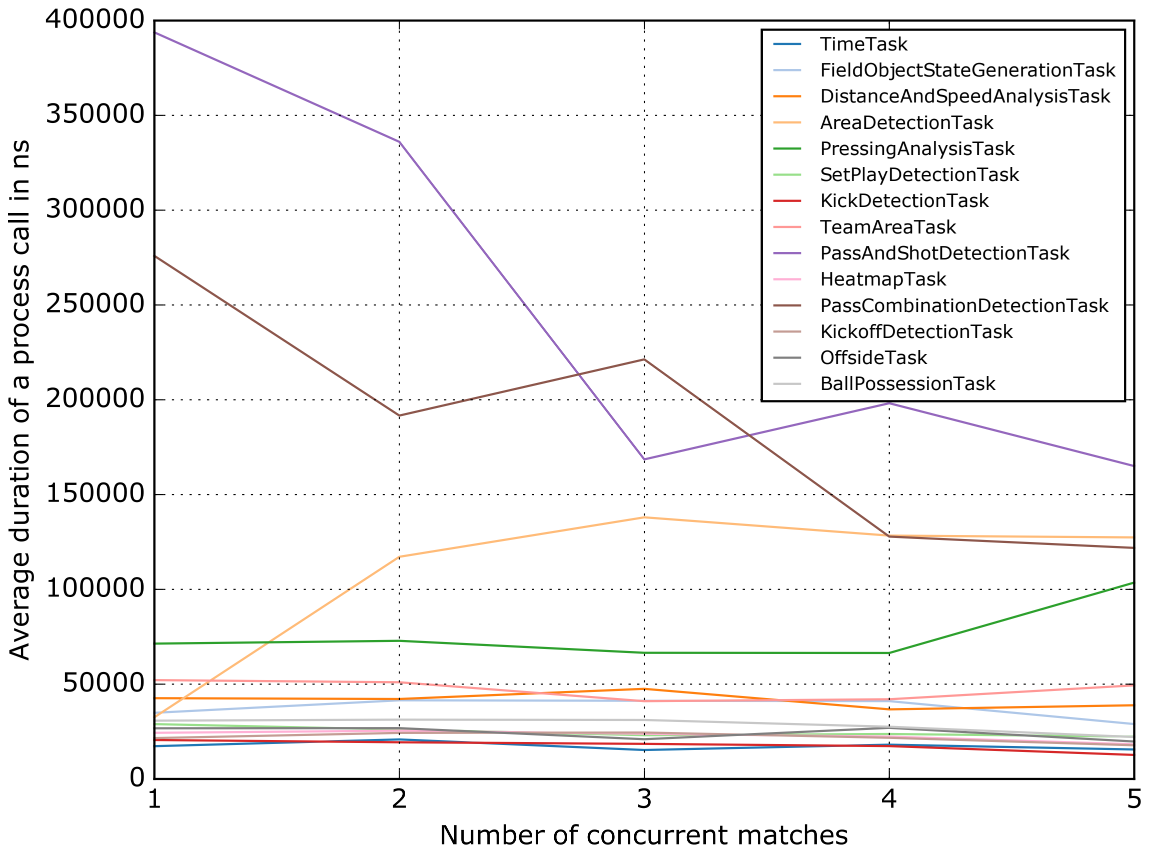


Figure 10.10 Average Duration of a Process Call for Single Container and Single Partition Configurations. The figure shows the average duration of a process call in nanoseconds during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names in the legend end with "Task" since the legend shows the class names and ending the worker class file names with "Task" is Samza's naming convention [Sam17a]. Table D.5 lists all values.

each active keys window module are called (see Section 8.2.2.2). In consequence, the overall workload of the data stream analysis system which can be calculated by multiplying the number of process calls with the duration of each process call and the number of window calls with the duration of each window call is shown to increase approximately linearly with the number of matches.

Table 10.3 and Figure 10.12 list and visualize, respectively, how the mean latencies change for an increasing number of matches. The mean latencies of the field object state stream elements, the kick event stream elements, the pass statistics stream elements, and the pass sequence event stream elements show a trend to increase with the number of matches. However, the observed increase is only weak. In contrast, the mean latencies of the heatmap statistics stream elements increase remarkably and linearly with the number of matches. Although

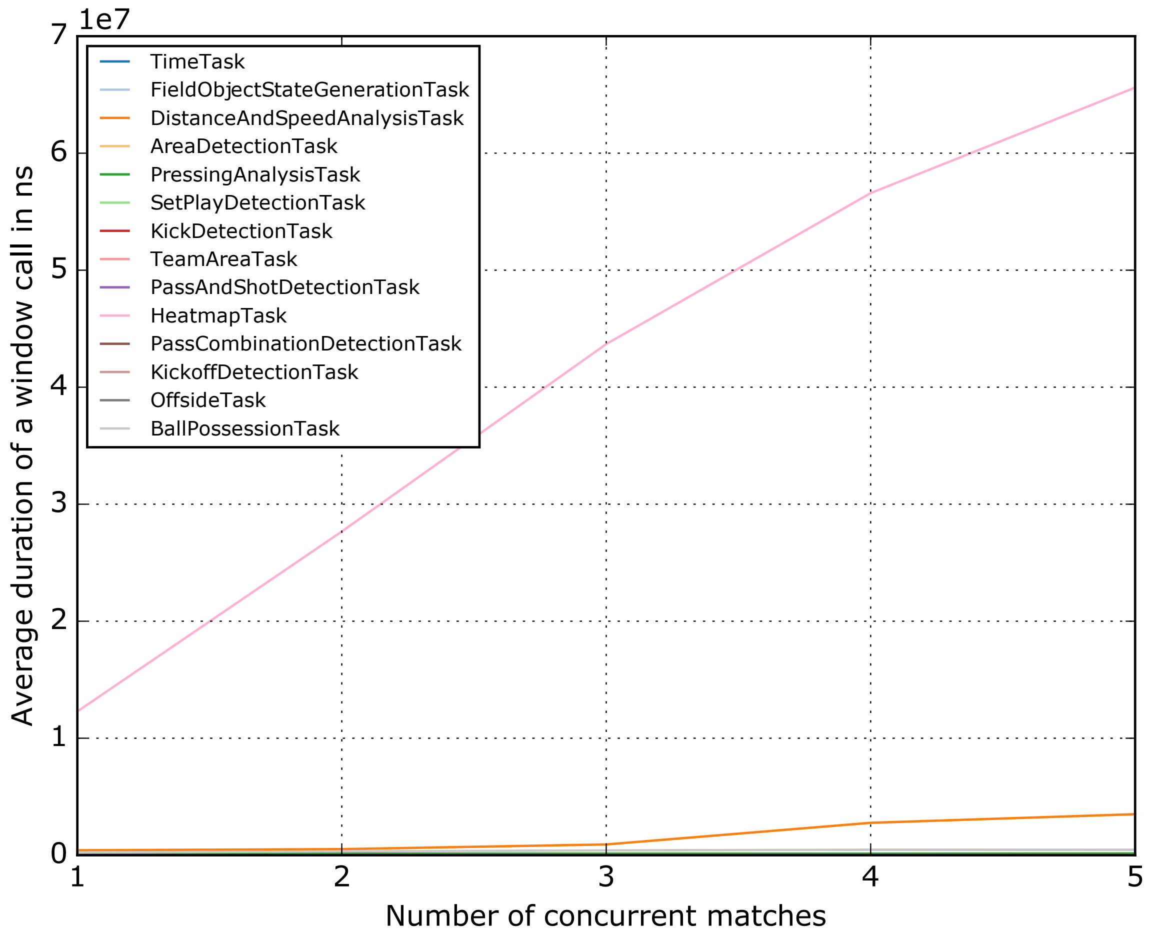


Figure 10.11 **Average Duration of a Window Call for Single Container and Single Partition Configurations.** The figure shows the average duration of a window call in nanoseconds during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names in the legend end with “Task” since the legend shows the class names and ending the worker class file names with “Task” is Samza’s naming convention [Sam17a]. Table D.6 lists all values.

also the time required to generate all heatmaps for all concurrent matches, or more precisely the average duration of each window call which the Samza task of the heatmap worker triggers, increases linearly with the number of matches (see Figure 10.11), the durations are too small (65.60 milliseconds for five concurrent matches) to be the only cause of this latency increase. Instead, we argue that this is another fact that backs our supposition that the transmissions are the reason for the elevated latencies of the heatmap statistics stream elements.

The other latency statistics and the CDFs which we provide in Appendix D exhibit the same trends as the mean latencies. Moreover, the fact that the most latencies are only slightly higher if five concurrent instead of only a single match

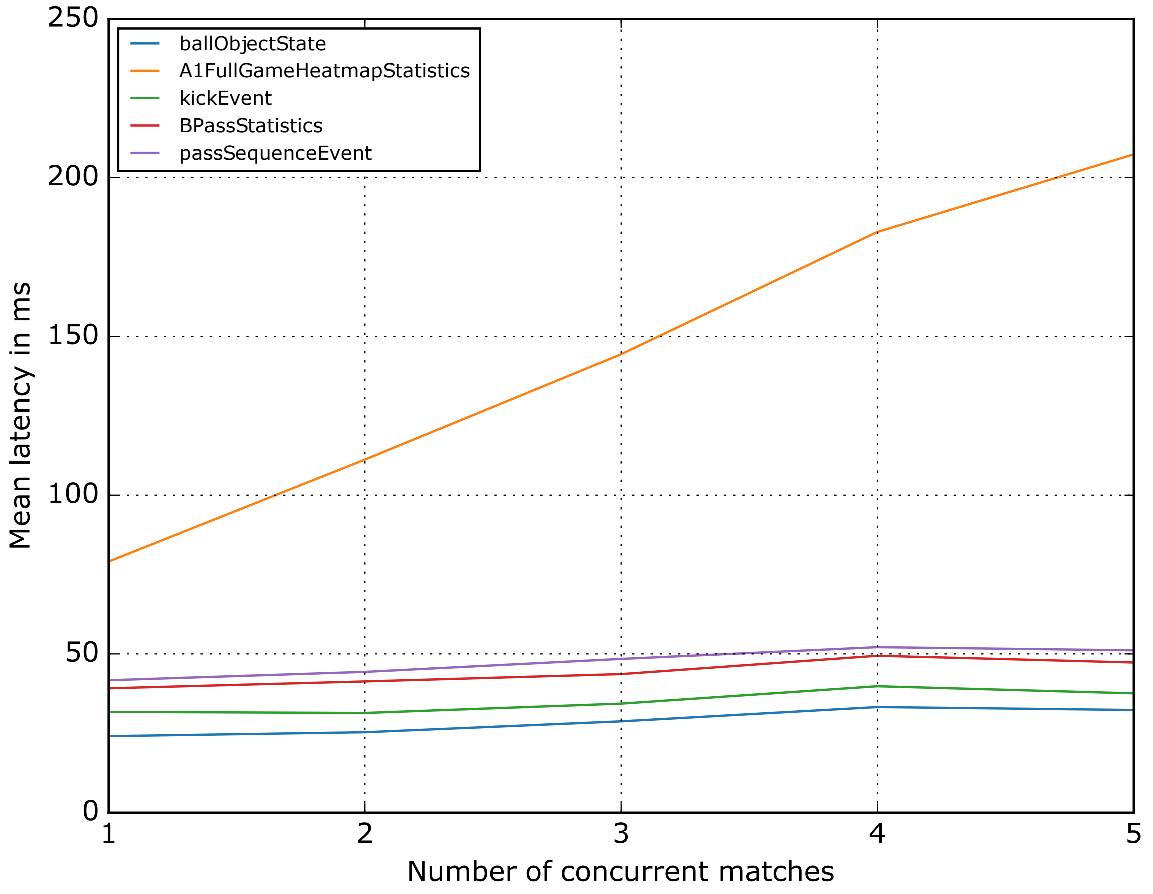


Figure 10.12 Mean Latencies for Single Container and Single Partition Configurations. The figure shows the mean latencies for the five data stream subsets for an increasing number of concurrent matches.

are analyzed and that even the 99th percentile latency for the heatmap statistics stream elements is below three seconds confirms that StreamTeam-Football is able to analyze not only a single but even multiple matches in parallel while still meeting the real-time demands of the football analysis scenario even without using all parallelism and distribution features which StreamTeam inherits from Kafka and Samza, or more precisely without increasing the number of partitions and containers.

Finally, we take a brief look on how StreamTeam-Football's performance changes if Kafka's and Samza's parallelism and distribution features are used. Figure 10.13 and Figure 10.14 show how the values of the four Samza metrics change for the heatmap worker and the ball possession worker, respectively, if the number of containers and the number of partitions are increased. The figures for the other workers of StreamTeam-Football's analysis workflow are given in Appendix D. As expected, the average aggregated number of process calls per second remains unchanged if the number of containers and partitions

Data stream subsets	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
ballObjectState	24.10 ms	25.33 ms	28.76 ms	33.26 ms	32.34 ms
A1FullGameHeatmapStatistics	79.08 ms	111.20 ms	144.38 ms	182.91 ms	207.33 ms
kickEvent	31.74 ms	31.40 ms	34.34 ms	39.81 ms	37.58 ms
BPassStatistics	39.17 ms	41.33 ms	43.64 ms	49.41 ms	47.29 ms
passSequenceEvent	41.71 ms	44.34 ms	48.42 ms	52.12 ms	51.13 ms

Table 10.3 Mean Latencies for Single Container and Single Partition Configurations. The table lists the mean latencies for the five data stream subsets for an increasing number of concurrent matches. All values are rounded to two decimals.

is changed. This makes sense as the number of data stream elements which the tasks of a Samza job process together depends neither on the number of partitions nor on the number of containers but only on the number of matches. Moreover, the average aggregated number of window calls per second scales linearly with the number of partitions. Also this is as expected since the number of partitions determines the number of tasks and the window function is called in the same frequency at each task no matter how many tasks exist (see Section 8.2.2).

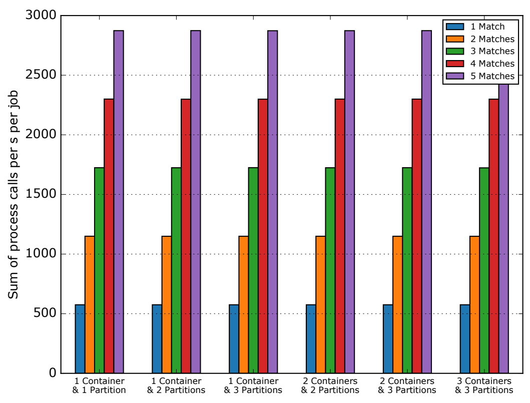
Unfortunately, there is no clear trend for the average process call duration and the average window call duration. The average process call durations seem to decrease when the number of partitions and/or containers is increased. We suppose that this is a benefit of distributing the state and the process calls to more tasks which are distributed to more containers. Moreover, increasing the number of partitions and/or containers seems to decrease the average window call durations. This is as expected since the workload in each window call is reduced if the number of tasks is increased as there are less active keys in each partition and since the tasks are distributed to more containers. However, there are many outliers which break these trends.

In addition, there is also no clear trend for the latencies of the output stream elements. Figure 10.15 shows the mean latencies for all five data stream subsets for all configurations. The figures for the other latency statistics are presented in Appendix D. The figures give the impression that there might be a little trend for the latencies to decrease when the number of partitions and containers is increased. However, again there are many outliers.

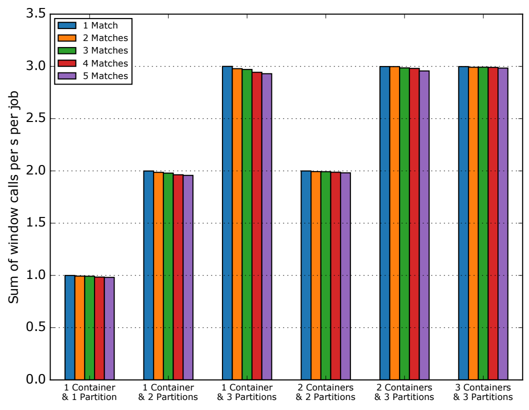
We suppose that one reason for the many outliers is that Samza's hash-based partitioning (see Section 8.2.2) does not guarantee that the matches are uniformly distributed to the partitions and thus to the tasks and containers.[22] In order to make reliable statements about the effect of using the parallelism and distribution features it would be necessary to expand the configuration space to much more parallel matches, much more partitions, and much more containers and redo the evaluations. However, doing so would require a huge amount of computing resources. Moreover, expanding the configuration space would mainly evaluate the distribution and scalability features which StreamTeam inherits from Kafka and Samza. As we have indicated in the introduction of this chapter, this is out of the scope of this thesis.

Instead, the main objective of our performance evaluation was to show that StreamTeam-Football is able to analyze multiple parallel matches in real-time. This could be confirmed with the results we have presented above. Moreover, the results could confirm that StreamTeam-Football can make use of the parallelism and distribution features which StreamTeam inherits from Kafka and Samza.

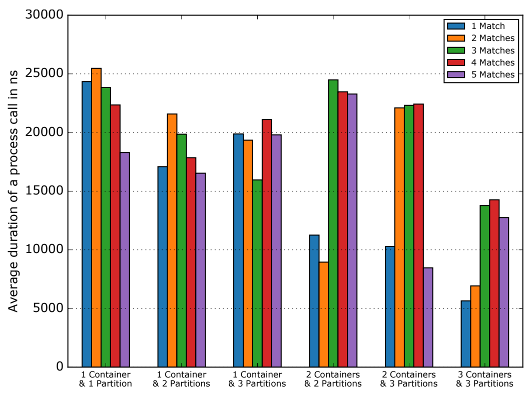
[22] Note that, although hash-based partitioning is known to yield to an almost uniform distribution when hashing a huge number of keys, this not the case for such low numbers.



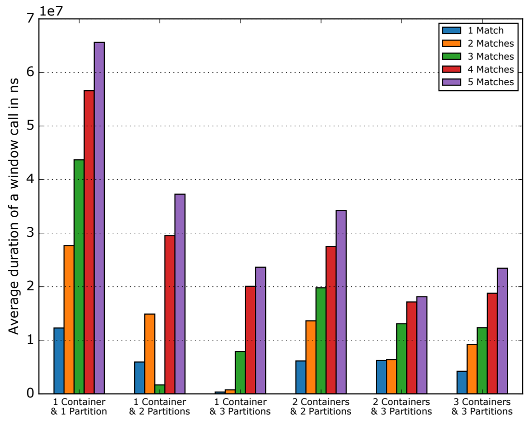
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

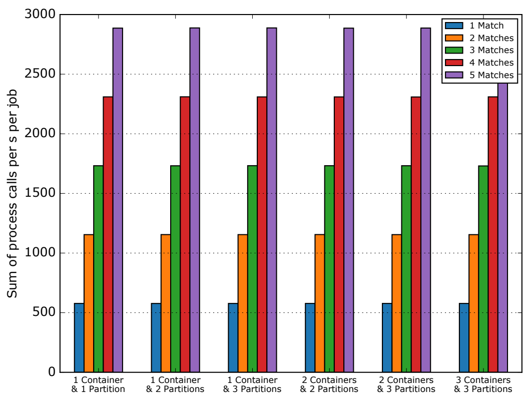


(c) Average Duration of a Process Call

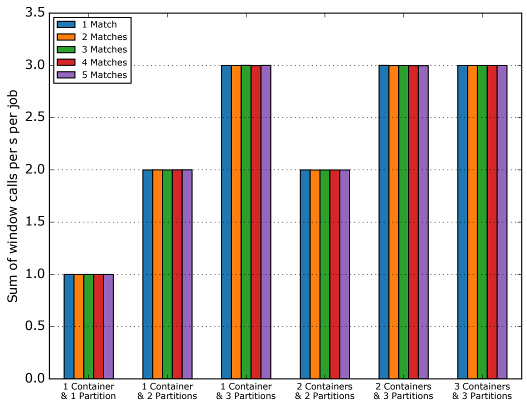


(d) Average Duration of a Window Call

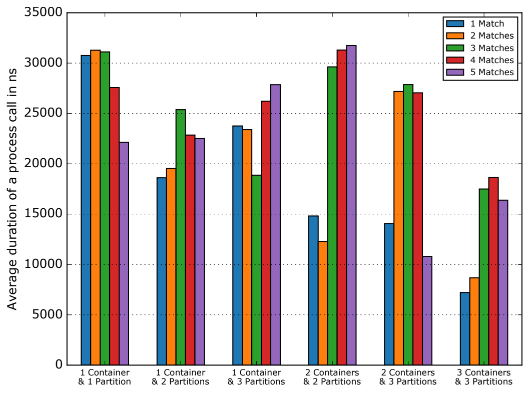
Figure 10.13 Samza Metrics for all Configurations for the Heatmap Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the heatmap worker for all configurations.



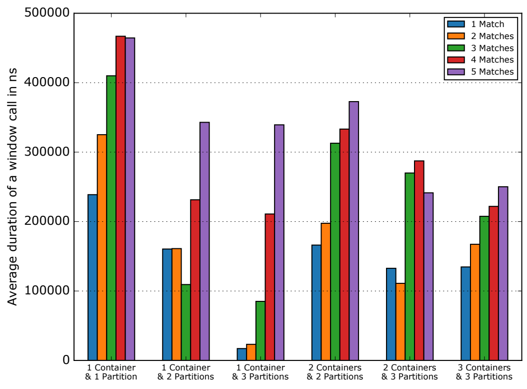
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

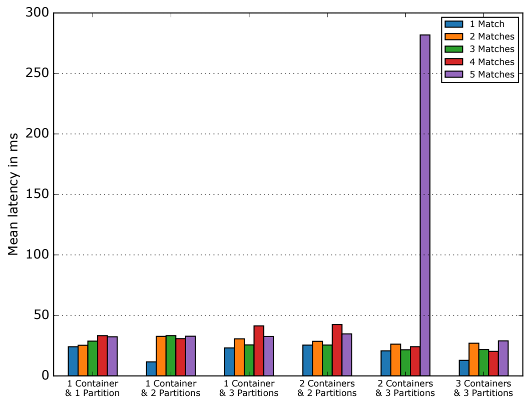


(c) Average Duration of a Process Call

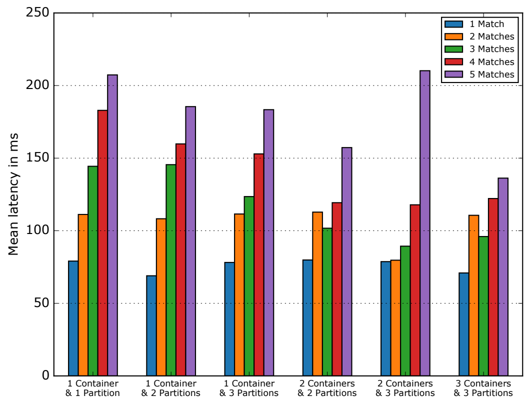


(d) Average Duration of a Window Call

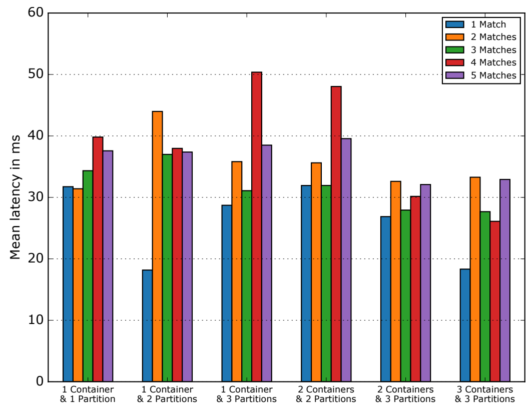
Figure 10.14 Samza Metrics for all Configurations for the Ball Possession Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the ball possession worker for all configurations.



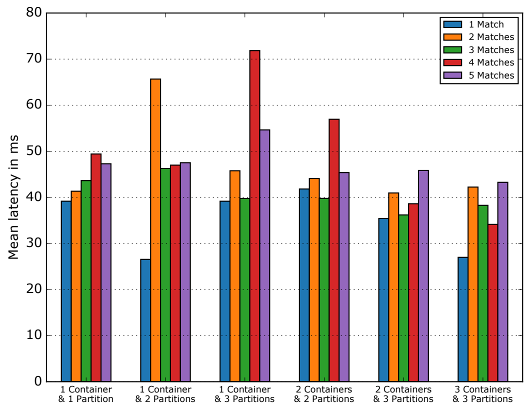
(a) ballObjectState



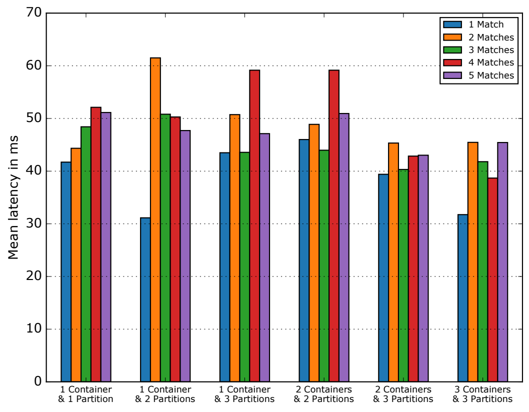
(b) A1FullGameHeatmapStatistics



(c) kickEvent



(d) BPassStatistics



(e) passSequenceEvent

Figure 10.15 Mean Latencies for all Configurations. (a), (b), (c), (d), and (e) show the mean latencies for the five data stream subsets for all configurations.

Part IV

Conclusion

11 Related Work

In this chapter, we will first summarize the related work on generic data stream analysis which we have discussed in Part II and Chapter 8. Subsequently, we will discuss different team sports analysis approaches.

11.1 On Data Stream Analysis

In order to make it easier for the reader to compare our generic model presented in Part II and our generic data stream analysis infrastructure (i.e., StreamTeam) presented in Chapter 8 with existing literature we have integrated the related work on generic data stream analysis into Part II and Chapter 8. More precisely, we have integrated a short overview of the diverse data stream analysis system types into Chapter 3. Moreover, we have discussed related work for aspects of our generic model and our generic data stream analysis system prototype in dedicated "Literature Discussion" boxes.

Literature Discussion 5.1, 5.2, and 5.3 discuss which other terms are used in literature to refer to raw input stream generating devices, workers, workflows, and entry components. Literature Discussion 4.1 and 5.4 encompass related work on key-based data parallelism and state which other terms are used to denote processors. The background of the processing procedure of and the buffering at the processors is presented in Literature Discussion 5.5 and 6.5.

Literature Discussion 6.1, 6.2, 6.3, and 6.4 discuss related work on the generation time notion, on the processing time notion, on the ingestion time notion, and on sequence numbers, respectively. An overview of how orderings with respect to generation timestamps and ingestion timestamps are discussed in literature is given in Literature Discussion 6.6 and 6.7.

Finally, Literature Discussion 8.1 and 8.2 discuss related work on our generic data stream analysis system prototype implementation, namely on our worker code modularization and our ingestion time assignment approach.

11.2 On Team Sports Analysis

In this section, we will discuss related work on our real-time football analysis application (i.e., StreamTeam-Football) which we have presented in Chapter 9. More precisely, we will present different approaches towards analyzing team sports matches.

11.2.1 Software-Aided Manual Analysis

Until today systems like StreamTeam-Football which analyze team sports matches fully automatically are only rarely used in practice. Although it is common to generate statistics by aggregating event data, the detection of the events and thus the generation of the event basis for further analyses is still performed manually by match analysts or even coaches. That is, people have to capture the events which happen in a match manually either live during the match or post-match while reviewing videos of the match. This includes selecting the correct event type as well as assigning the correct temporal and spatial information to the event. However, there are software products which assist these people in the manual event detection process.

On the one hand, commercial dataset providers develop their own software solutions to assist their employees in generating the datasets they sell. For instance, Opta, the company which produces and sells the F24 data feeds [Opt20a; Opt20b] which are accepted and prevalent in industry and which we have used in our qualitative evaluation (see Section 10.2), equips its employees with software to capture the events live during the match [Nut18].

On the other hand, there are companies which develop and sell software products that enable the match analysts of clubs, associations, or broadcasters to capture the events on their own. For instance, Coach Capture [Chy20a], myDartfish Live S [Dar20], and Sportscode [Hud20] contain interfaces which assist the user in capturing event manually while watching a live stream or a recorded video of the match. Such solutions are particularly useful for small clubs which cannot afford the prices of the commercial datasets as the coach of a team can capture the events on his/her own.

Moreover, there are approaches in academia to leverage cheap crowdsourcing platforms to generate event datasets. For instance, CrowdSport [SGS14] combines the events which multiple microworkers capture in video snippets to one event dataset. In doing so, they take the diverse trustworthiness of the heterogeneous microworkers into account in order to improve the quality of the dataset [SGS14]. Interestingly, the evaluation results of CrowdSport show that the microworkers have more problems in assigning correct spatial information than in assigning correct temporal information [SGS14]. As the event capturing process in CrowdSport is similar to the one performed by the Opta employees, these results back our supposition that some spatial inaccuracies which we measured in our qualitative evaluation are not caused by StreamTeam-Football but by imperfect positions in the F24 data feed (see Section 10.2.2.5).

11.2.2 Automatic Video-Based Analysis

As indicated in Section 11.2.1, humans rely on visual and aural input to detect events in a match. It is therefore no wonder that there are academic approaches towards analyzing team sports matches which use the visual and/or aural features of a match video to detect events in the match and thus somehow mimic the human approach.

For instance, Fleischman and Roy [FR07] leverage visual features (camera motions and scene categories), aural features (sound categories), and the closed captioning text in order to detect events in a baseball match using unsupervised learning methods. Moreover, Chen et al. [CFL+14] use visual features to detect events in an American football match by means of applying computer vision techniques.

11.2.3 Automatic Position-Based Analysis

In contrast to these video-based systems, there are academic approaches towards analyzing team sports matches on the basis of raw positions of the players and the ball. We differentiate between two categories of position-based team sports analysis systems. On the one hand, there are offline systems which analyze past matches on the basis of complete position datasets. On the other hand there, are real-time systems (such as StreamTeam-Football) which analyze live matches on the basis of continuous position streams.

11.2.3.1 Tracking

The raw positions which serve as an input for the position-based team sports analysis systems are generated by sensor-based or video-based tracking systems which we do not regard to be a part of the analysis system.

Sensor-based tracking requires that each object which should be tracked is equipped with a small physical device. There are many commercial products on the market. RedFIR [GFW+11], LPM [SPF04], Clearsky T6 [Cat20a], TRACAB RF [Chy20d], and NBN23 Performance [NBN20] leverage a local positioning system which is deployed around the field. In contrast, OptimEye S5 [Cat20b], TRACAB GO [Chy20b], Stats Perform GPS [Sta20b], and FieldWiz [Adv20] make use of global navigation sattelite systems such as GPS, Galileo, GLONASS, or Beidou. While local positioning systems are more precise they are not portable. Moreover, they are pretty expensive and thus not affordable for small clubs.

In contrast, video-based tracking systems extract the positions of the objects from a single or multiple videos. There are commercial products such as TRACAB Optical Tracking [Chy20c], inmotio's ball tracking system [Inm20], and SportVU [Sta20a] which extract the positions from videos which are recorded with dedicated cameras that are mounted at known positions around the field. Moreover, there are recent approaches in academia towards tracking players using broadcasted video material [SJL+18].

It is also possible to combine both approaches. For instance, the SFISM uses inmotio's tracking solution [Inm20] which tracks players using the sensor-based LPM system [SPF04] but the ball using their own video-based tracking system since they argue that a sensor in the ball would influence its trajectory and is thus not accepted by the players and coaches.

11.2.3.2 Offline

As mentioned above, offline position-based team sports analysis systems analyze past matches on the basis of complete position datasets. More precisely, they consume a static dataset which contains all positions that were tracked during the match as the input for their analyses. In consequence, these systems have all benefits of static data analysis (see Section 3.1). That is, these systems have the option to iterate multiple times over the complete dataset and to access specific data items. Moreover, they are not bound to strict temporal requirements since the analysis is anyways not performed for a live match.

Typically, offline position-based team sports analysis systems leverage the plethora of well-established machine learning methods which have been developed for static datasets. For instance, Richly et al. [RBR+16; RMS17] detect events in position datasets of a football match by means of applying different machine learning approaches, namely classification with a Support Vector Machine (SVM) [CV95], the K-Nearest Neighbors (KNN) algorithm [Alt92], the Random Forest approach [Bre01], and a three-layered Neural Network [Bis06, pp. 225–290]. Moreover, Sangüesa et al. [SMB+17] as well as Wang and Zemel [WZ16] leverage machine learning approaches to classify plays in basketball matches. While Sangüesa et al. [SMB+17] apply a Principal Component Analysis (PCA) [WEG87] and diverse machine learning algorithms (classification trees, SVMs, and KNN), Wang and Zemel [WZ16] leverage a normal Neural Networks and a Recurrent Neural Network (RNN) [RHW86].

11.2.3.3 Real-Time

In contrast, real-time team sports analysis systems analyze live matches on the basis of continuous position streams. In consequence, these systems have all the disadvantages associated with data stream analysis (see Section 3.1). Namely, they cannot access all input positions of the whole match from begin on and multiple times but only volatile portions of the input data and have thus to store all important information in the state. Moreover, they have to perform the whole workload in real-time.

However, the real-timeness of these systems is not only a technical challenge but also their main selling point since the analysis results which they emit are not only available after the match but even live during the match. Hence, they cannot only be used to improve the performance of a team in the next matches but even in the currently ongoing match (see Section 2.1).

The first approaches towards analyzing team sports matches in real-time on the basis of position streams were proposed as solutions for the DEBS 2013 Grand Challenge [MZJ13]. The organizers of this challenge provide a position dataset of a football match captured with the RedFIR tracking system [GFW+11] and define four analysis tasks, namely generating running statistics, ball possession statistics as well as heatmaps and detecting shots on goal. We want to highlight that our generic generation timestamp inheritance rule (see Section 6.1.1) matches the way the organizers define the output stream element timestamps since they specify the timestamp of each analysis result to be the timestamp of the last input which updated the result [MZJ13].

Eight approaches have been presented as solutions for this challenge at the

DEBS 2013 conference. Jacobsen et al. [JMR+13] propose not only a single but three solutions, namely one which bases on the CEP system Esper [Esp20], one which bases on the worker-based data stream analysis system Storm [TTS+14], and one application-specific (or more precisely challenge-specific) system the authors developed from scratch. Moreover, also Wu et al. [WMT13] and Jergler et al. [JDN+13] propose application-specific systems. Madsen et al. [MSZ13] propose a solution based on Enorm [MZS16], a programming framework on top of Storm [TTS+14] which enables developers to implement real-time analysis applications in an online MapReduce style. Gal et al. [GKS+13] propose a solution which uses the streams framework [BB12] to connect Esper [Esp20] processors to a workflow. Badiozamany et al. [BMT+13] propose a solution based on an extended version of the DSMS which has been proposed in [ZR11]. It is worth mentioning that the solutions proposed in [JMR+13], [JDN+13], and [GKS+13] contain a simple result visualization component although this was no requirement in the challenge.

Also the first efforts which our research group made towards analyzing football matches in real-time were triggered and influenced by the Grand Challenge. More precisely, the first football analysis workflow which we have developed to debug and evaluate PAN [Pro14; PGS16a; PGS16b] solve two of the four Grand Challenge analysis tasks (generating ball possession statistics and heatmaps) and the first version of our sensor simulators [Pro14] used the Grand Challenge dataset to generate raw position sensor data stream elements. Hence, although we changed early to replaying ball-enriched LPM datasets [SPF04; Inm20] provided by the SFISM or TRACAB Optical Tracking datasets [Chy20c] instead of the Grand Challenge dataset and although already our first more extensive football analysis application implemented in PAN [Bri16] performed different analyses as those defined in the Grand Challenge, we argue that the Grand Challenge as well as their solutions [JMR+13; WMT13; JDN+13; MSZ13; GKS+13; BMT+13] which we considered when developing the first workers had and still have an impact on our work and thus also on StreamTeam-Football.

In addition, the Grand Challenge dataset has been used to demonstrate Herakles [MBC+15; BBC+15]. Herakles is a real-time football analysis application that is deployed by submitting continuous queries to OdysseusP2P [Mic14], a distributed DSMS. Herakles does not only solve the four analysis tasks of the Grand Challenge but detects for instance pass events and generates pass statistics. Moreover, Herakles has a considerable user interface whose features are comparable to those which StreamTeam-Football's user interface provides. However, Herakles still performs much less analyses than StreamTeam-Football. This is true in general as Herakles detects for instance no set play events, no dribbling events, and no pass sequence events. But the crucial difference is that StreamTeam-Football, in contrast to Herakles, performs analyses (such as detecting duels, calculating a pressing metric, or calculating team area states) which consider the interaction of multiple players or even whole teams.

12
Summary

In the beginning of this thesis, we have described the need to analyze collaborative team behavior on the basis of data about the individuals that form the team. Moreover, we have identified five challenges which performing such analyses poses on the system. In addition, we have shown that these five challenges are relevant in practice by sketching the real-time football analysis scenario which we have used as the running example throughout this thesis. In a nutshell, the challenges can be summarized as five requirements:

1. The team collaboration analysis application has to be implemented on top of a data stream analysis system which processes input data stream elements in real-time.

2. The data stream analysis system has to support analyzing the input stream elements not only separately for each input stream or input stream generating device but jointly.

3. The data stream analysis system has to provide support for splitting the overall analysis task into cleanly separated analysis subtasks.

4. The data stream analysis system has to support parallelism and a distributed deployment in order to be able to scale with respect to the number of analyses and with respect to the number of input stream elements for which these analyses have to be performed.

5. The collaborative team behavior analysis logic has to be based on a strong theoretical foundation about the different stream time notions and the basic spatial functions and relations.

In Part II, we have addressed the theoretical part of these challenges. After introducing the fundamentals on data stream analysis we have presented our data stream model which defines data streams, data stream elements, and data stream partitions. Based on investigations of the information which is shipped in data stream elements that are consumed or produced by team collaboration analysis applications, we have introduced a novel schema for encoding common information in a consistent way in generic data stream independent attributes and data stream specific information in a data stream specific payload attribute. Moreover, we have introduced a distinction between atomic and non-atomic data stream elements and a separation of the data streams into four categories which reflect their semantics in collaborative team behavior analysis.

Subsequently, we have defined our system model of a worker-based data stream analysis system. More precisely, we have defined the conceptual and physical components of a worker-based data stream analysis system, described how key-based data parallelism is supported, described the processing procedure at the processors, and addressed machine and network related aspects of our system model. A novelty of our system model is that it considers the different data stream categories and differentiates between the input and output streams of the data stream analysis system. Moreover, we have introduced sophisticated well-formation constraints for the workflow. In addition, we have not limited our system model to the components of a single data stream analysis system but further modeled the raw input stream generating devices and stated which additional constraints have to be regarded when deploying multiple coexisting data stream analysis systems.

In order to establish a strong theoretical foundation for the temporal aspects of the analysis logic, we have presented an extensive stream time model which goes far beyond existing literature on time notions in data stream analysis. More precisely, we have defined, compared, and discussed different time notions as well as the orderings introduced by the sequence numbers contained in and by the timestamps which can be assigned to the data stream elements. Moreover, we have presented a novel simultaneousness concept which covers if two data stream elements refer to approximately the same moment in time or not and if it is even possible in a certain team collaboration analysis scenario that two elements refer to approximately the same moment in time.

To provide a theoretical foundation for the spatial aspects of the analysis logic, we have defined basic spatial functions and relations which can be used as building blocks to develop the logic for detecting, calculating, and generating collaborative team events, states, and statistics.

In Part III, we have addressed the technical part of the challenges by presenting our implementation which is published on GitHub (see Appendix B). More precisely, we have presented StreamTeam, our generic real-time data stream analysis infrastructure which is designed to be used as a foundation for developing team collaboration analysis applications, and StreamTeam-Football, the real-time football analysis application which we have implemented on top of StreamTeam. At its heart StreamTeam contains our prototype implementation of a worker-based data stream analysis system which expects data to be structured as defined in our data stream model, whose architecture is designed according to our system model, and which supports all time notions that we have defined in our stream time model. Moreover, StreamTeam introduces novel approaches to modularize the code (even inside a worker) and to facilitate separating the analysis by application-specific keys in order to assist domain experts without a profound software engineering background in developing their own analyses. With StreamTeam-Football we have implemented the first analysis application which performs complex team behavior analyses in a football match in real-time, visualizes the live analysis results in a user interface, and stores them persistently for offline activities such as video scene retrieval. The analysis logic of StreamTeam-Football's workers relies on our theoretical foundation for using spatial and temporal information such as our formal definitions of the different time notions, our simultaneousness concept, and the basic spatial functions and relations. Our qualitative evaluations show that StreamTeam-Football is a non-trivial real-time team collaboration analysis application which fulfills the analysis demands of football coaches, match analysts, and sports scientists. Moreover, our quantitative evaluations confirm that StreamTeam-Football is able to analyze multiple football matches in parallel in real-time, that StreamTeam's data stream analysis systems scales with respect to the number of processed and emitted data stream elements, and that the theoretical statements on the (un)ambiguity of the diverse timestamps which we have posed in our stream time model are correct.

In conclusion, this thesis has presented novel approaches to address the challenges which analyzing collaborative team behavior on the basis of data about the individuals poses on the system that performs the analyses and thus makes an important contribution towards implementing team collaboration analysis applications in general. Moreover, this thesis has presented with StreamTeam-Football a convincing proof-of-concept for automatically analyzing the collaborative behavior in football matches in real-time on the basis of player and ball position streams. In fact, StreamTeam-Football has not only aroused interest in general public – the Bieler Tagblatt [Sza17], 20 Minuten [Fri17], Tageswoche [Wal17], Barfi.ch [Sta17], and Spick [Lan18] have published articles and the public relations team of the University of Basel has produced a video which has been published via the university news channel [Uni17] – but also currently ongoing attempts to refine our prototype into a commercial product that is planed to be used in practice to analyze matches of professional football teams.

13 Future Work

Research never stops. As always, new contributions offer potential for future work. In this chapter, we will present some directions into which our work can be continued in future projects.

13.1 Machine Learning

With this thesis, we have proposed an algorithmic approach towards analyzing the collaborative behavior of teams in real-time. That is, we have specified the logic for detecting events, calculating states, and generating statistics by means of implementing the code of multiple workers which form the overall analysis workflow. The main driving force behind this design decision was our collaboration with sports scientists from the SFISM. Their approach is to convert obscure terms like pressing which still lack a clear, universally accepted definition into clearly modeled concepts that can be converted into analysis algorithms [SRP+19]. With our work, we have not only implemented the first concepts which our colleagues have defined in [SRP+19] but further created an infrastructure which they can use to implement and evaluate concepts they model in the future. Moreover, algorithmic analysis logic has the advantage that it can be easily adapted and extended – especially if the overall analysis is separated into analysis subtasks as in our model and implementation – if the analysis demands change.

An alternative approach is to define events not by means of specifying algorithms but by means of applying machine learning techniques. In the last years, a multitude of machine learning approaches for data stream analysis in general and event detection in particular have been proposed. It would be very interesting to implement another real-time football analysis application which uses only machine learning techniques and to compare its detection quality with the detection quality of StreamTeam-Football. Moreover, we see a huge potential in combining both approaches, for instance, by means of adding some machine learning based workers that perform analyses which are very difficult to implement algorithmically (e.g., formation detection) to the existing analysis workflow of StreamTeam-Football. Another interesting area of research is to tackle the question how to generate the huge amount of training data which are required to use machine learning techniques (e.g. by means of leveraging crowdsourcing platforms as done in [SGS14]).

13.2 Probabilistic Event Detection

In our work, we have followed a binary event detection model. That is, we only detect that an atomic event occurred or that a non-atomic event started, is active, or ended. As a direction for future work we suggest to extend our model and implementation by introducing support for probabilistic events [ASA+17]. More precisely, we suggest to explore the potential of treating prolonged interactions whose success cannot be guaranteed from begin on (e.g., passes) not as atomic events but as probabilistic non-atomic events. Moreover, we suggest to investigate if assigning the occurrence of a very complex and controversially discussed team behavior pattern, such as an offside trap, a probability receives more acceptance than simply reporting them as events.

13.3 Analyzing Football

Although StreamTeam-Football is already a convincing proof-of-concept for automatically analyzing the collaborative behavior in football matches in real-time, there is still potential to improve it in future projects. Since StreamTeam-Football has been implemented as a research prototype, its current analysis workflow exhibits some limitations (see Section 9.2.2) which should be addressed before using StreamTeam-Football in practice. Furthermore, the code of some of its workers can be improved to increase its detection quality.

In addition, StreamTeam-Football can also be used as a basis for future research in football analysis. As mentioned above, it can be used by sports scientists to implement and evaluate concepts they model in the future [SRP+19]. Moreover, we suggest to leverage the potential of our model and our generic infrastructure implementation to analyze not only positions but also health data about the players (e.g. their heart rate) in StreamTeam-Football's analysis workflow. This could be used, for instance, to assess if a player is not willing to or not capable of fulfilling its role in the team tactics. In addition, the analysis results which StreamTeam-Football emits in data stream elements and stores in a MongoDB instance can be used as input for other team sports analysis systems, such as SportSense, and thus assist the research in other fields, such as video scene retrieval.

13.4 Analyzing other Team Sports

Football is not the only team sports which could profit from analyzing the collaborative behavior of the team in real-time. Other famous team sports, such as American football, ice hockey, and basketball, exhibit similar collaborative team behavior. Our model and our generic data stream analysis infrastructure have already been designed with arbitrary team sports scenarios in mind. Adapting StreamTeam-Football for other team sports in order to investigate how much of the analysis workers can be generalized and reused would be an interesting field of research both from a computer science and a sports science perspective.

Moreover, besides "physical" team sports, also many eSports involve sophisticated team tactics which are worth being analyzed in a collaborative team behavior analysis application. A first promising attempt towards analyzing DotA2 matches in an application that has been implemented on top of an earlier version of the StreamTeam infrastructure has been already made in [Zum19]. We suggest continuing this attempt in future projects as eSports is a steadily growing market.

13.5 Analyzing Disaster Management

Besides team sports scenarios we have also listed disaster management scenarios as scenarios which could benefit from analyzing the collaborative team behavior (see Section 2.2). Collaborative team behavior analysis in disaster management scenarios shares the challenges which we have addressed in this thesis and thus profits from the contributions which we have made in this thesis. However, it also poses some additional challenges which we have not addressed yet in our model and implementation since they have been out of the scope of this thesis. For instance, obstacles or dense smoke might create long-lasting network partitions which we have excluded in our model. Moreover, while it is valid that we have regarded failure tolerance as optional in team sports analysis scenarios this is not the case in disaster management scenarios. Extending our model and implementation towards addressing the additional challenges and implementing an application on top of StreamTeam which performs collaborative analyses in a forest fire fighting scenario (e.g., based on the data that are generated by the simulator proposed in [Obr16]) would be an interesting direction for future projects.

Appendix

A

Additional Counterexamples for Consistency between Sequence Number Ordering and Generation Time Ordering

In Section 6.2.5 we have leveraged a counterexample in which we assume that the proxy approach is used to assign sequence numbers to prove that the sequence number ordering is not guaranteed to be consistent with the generation time ordering of a data stream partition (see Proof 6.8) and to prove that the processing time ordering is not guaranteed to be consistent with the generation time ordering of a data stream partition (see Proof 6.9).

However, the proxy approach is not the only sequence number assignment approach which we have presented in our model (see Section 6.2.1.1). In this appendix we will show that inconsistencies can not only be introduced by an additional proxy. Instead, even if the shared counter approach or the local counter approach are used to assign sequence numbers the consistency cannot be guaranteed and thus that Theorem 6.8 and Theorem 6.9 would also be valid if we restricted our model to one of these assignment approaches.

A.1 Shared Counter Approach

Even if the shared counter approach (see Section 6.2.1.1) is used to assign sequence numbers to elements of a data stream partition the generation time ordering of this partition would not be guaranteed to be consistent with the sequence number ordering and the processing time ordering.

Example A.1 Shared Counter Example for the Unguaranteed Consistency between Sequence Number Ordering and Generation Time Ordering and between Processing Time Ordering and Generation Time Ordering

Assume that there are two raw input stream generating devices igd_1 and igd_2 which emit data stream elements of the same partition of data stream ds and which assign sequence numbers to these elements by means of the shared counter approach (see Section 6.2.1.1). Moreover, assume that igd_1 generates new raw input data it will emit in dse_1 after 25 milliseconds and that igd_2 generates new raw input data it will emit in dse_2 after 37 milliseconds. Hence, the generation timestamps of both data stream elements are as follows:

$$dse_1.ts = 25 \qquad dse_2.ts = 37$$

Assume that after generating the raw input data both raw input stream generating devices immediately attempt to access the shared counter at the coordination service to obtain a sequence number for the data stream element shipping the raw input data. Further assume that the transmission delay between igd_1 and the coordination service is 100 milliseconds and that the transmission delay between igd_2 and the coordination service is only 20 milliseconds. Since the shared counter access of igd_2 reaches the coordination service after 57 (37+20) milliseconds and thus earlier than the shared counter access of igd_1 which reaches the coordination service after 125 (25+100) milliseconds, dse_2 is assigned a smaller sequence number than dse_1.

$$dse_2.\xi < dse_1.\xi$$

According to Definition 6.10 and Definition 6.11, this means that dse_1 is ordered before dse_2 by means of the generation timestamps assigned by the raw input stream generating devices but that dse_2 is ordered before dse_1 and thus reversely by means of the sequence numbers assigned with the shared counter approach.

$$\langle dse_1, dse_2\rangle \in \prec_{ts}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in \prec_{\xi}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

Moreover, assume that igd_1 and igd_2 emit dse_1 and dse_2 as soon as they have assigned a sequence number to them using the shared counter and that there is a processor pr which processes both elements sequentially with respect to their sequence numbers as defined in Section 5.3.2 and assigns the following processing timestamps to them:

$$\tau\left(dse_1, pr\right) = 229 \qquad \tau\left(dse_2, pr\right) = 79$$

According to Definition 6.12, this means that dse_2 is ordered before dse_1 by means of the processing timestamps assigned by pr. In consequence, the processing time ordering and the generation time ordering conflict.

$$\langle dse_1, dse_2\rangle \in \prec_{ts}(ds, k, \xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$
$$\langle dse_2, dse_1\rangle \in \prec_{\tau}\left(ds, k, pr, \xi\right) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_1.\xi \text{ and } \xi \geq dse_2.\xi$$

A.2 Local Counter Approach

Even if a processor consumed only input stream partitions whose sequence number ordering and generation time ordering are consistent, if this processor was the sole component which generates elements of a certain output stream partition, and if this processor used the local counter approach without further coordination (see Section 6.2.1.1) to assign sequence numbers to elements of this output stream partition the generation time ordering of this output stream partition would not be guaranteed to be consistent with the sequence number ordering and the processing time ordering.

Example A.2 Local Counter Example for the Unguaranteed Consistency between Sequence Number Ordering and Generation Time Ordering and between Processing Time Ordering and Generation Time Ordering

Assume that there is a processor pr_1 which emits time window based statistics for object A and B as elements of the same partition of the statistics stream ds periodically (based on its local clock using the timer feature presented in our system model) every 150 and 200 milliseconds, respectively. Moreover, assume that pr_1 is the only component which emits elements of this statistics stream partition and that pr_1 assigns sequence numbers to these elements immediately before emitting them by means of the local counter approach without any further coordination (see Section 6.2.1.1). Hence, the sequence numbers of the first statistics stream element for object A dse_{outA} emitted after 150 milliseconds and the first statistics stream element for object B dse_{outB} emitted after 200 milliseconds are as follows:

$$dse_{outA}.\xi = 1 \qquad dse_{outB}.\xi = 2$$

Assume that the sequence number ordering and the generation time ordering of all input stream partitions consumed by pr_1 are consistent. Further assume that the generation timestamps of the input stream element dse_{inA} and the input stream element dse_{inB} are the largest of all input stream elements which have updated the statistic for object A and B, respectively. More precisely, assume that their generation timestamps are as follows:

$$dse_{inA}.ts = 135 \qquad dse_{inB}.ts = 123$$

Since the generation timestamp of a statistics stream element is inherited from the input stream element with the largest generation timestamp which has updated the statistic contained in the element (see Section 6.1.1), the generation timestamps of dse_{outA} and dse_{outB} are as follows:

$$dse_{outA}.ts = 135 \qquad dse_{outB}.ts = 123$$

According to Definition 6.10 and Definition 6.11, this means that dse_{outB} is ordered before dse_{outA} by means of the generation timestamps assigned by the raw input stream generating devices but that dse_{outA} is ordered before dse_{outB} and thus reversely by means of the sequence numbers assigned with the local counter approach.

$$\langle dse_{outB}, dse_{outA}\rangle \in \prec_{ts}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_{outA}.\xi \text{ and } \xi \geq dse_{outB}.\xi$$
$$\langle dse_{outA}, dse_{outB}\rangle \in \prec_{\xi}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_{outA}.\xi \text{ and } \xi \geq dse_{outB}.\xi$$

Moreover, assume that there is another processor pr_2 which processes dse_{outA} and dse_{outB} sequentially with respect to their sequence numbers as defined in Section 5.3.2 and assigns the following processing timestamps to them:

$$\tau\left(dse_{outA}, pr_2\right) = 158 \qquad \tau\left(dse_{outB}, pr_2\right) = 207$$

According to Definition 6.12, this means that dse_{outA} is ordered before dse_{outB} by means of the processing timestamps assigned by pr. In consequence, the processing time ordering and the generation time ordering conflict.

$$\langle dse_{outB}, dse_{outA}\rangle \in \prec_{ts}(ds,k,\xi) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_{outA}.\xi \text{ and } \xi \geq dse_{outB}.\xi$$
$$\langle dse_{outA}, dse_{outB}\rangle \in \prec_{\tau}\left(ds,k,pr_2,\xi\right) \text{ for all } \xi \in \mathbb{N}_0 \text{ with } \xi \geq dse_{outA}.\xi \text{ and } \xi \geq dse_{outB}.\xi$$

B

GitHub Repositories

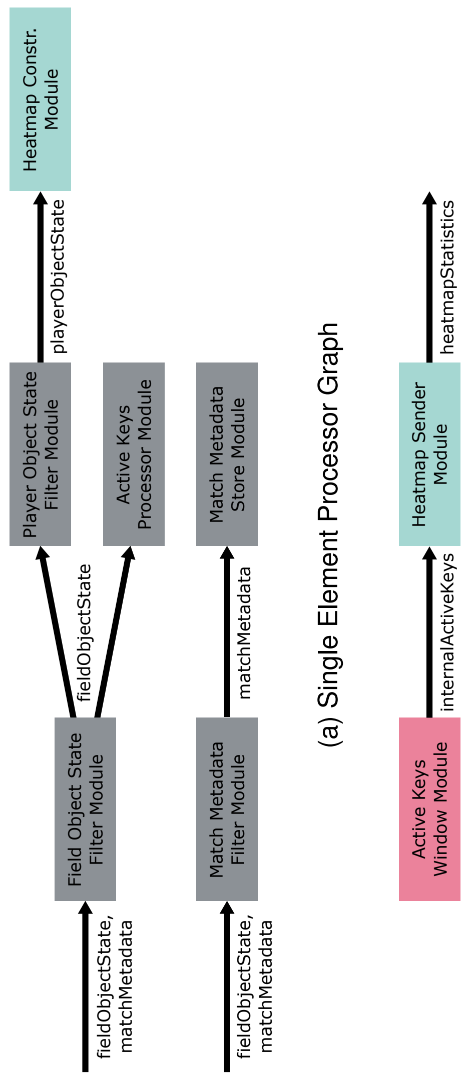
The code of all our generic and application-specific implementations is published in the following repositories of the StreamTeam GitHub project (https://github.com/streamteam) under the GNU Affero General Public License v3.0. All descriptions given in Chapter 8 and Chapter 9 as well as the evaluation presented in Chapter 10 refer to the code which is tagged with version 1.0.1.

- **streamteam-cluster-monitor:** Code of the cluster monitor (see Section 8.3.2)
- **streamteam-cluster-scripts:** Scripts for starting the integrated football analysis infrastructure (see Figure 9.14)
- **streamteam-data-model:** Generic code of the data stream model (see Section 8.2.4), football-specific payload message types, and football-specific wrapper classes
- **streamteam-data-stream-analysis-system:** Generic data stream analysis system code (see Section 8.2) and football analysis workers (see Section 9.2.1)
- **streamteam-evaluation:** Code/Scripts for the evaluation (see Chapter 10)
- **streamteam-kafka-rest-proxy:** Code of the Kafka REST proxy (see Section 8.3.1)
- **streamteam-mongodb-stream-importer:** Code of the MongoDB stream importer (see Section 9.4.2) and a script for creating the MongoDB collections (see Section 9.4.1)
- **streamteam-real-time-football-web-client:** Code of the football-specific real-time user interface (see Section 9.3)
- **streamteam-sensor-simulator:** Code of the sensor simulator (see Section 9.1)

C

Sample Module Graphs

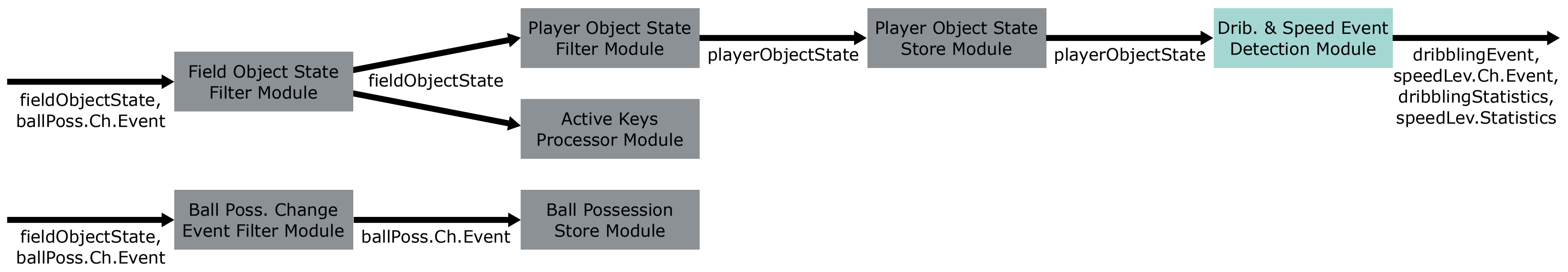
In order to give some concrete module graph examples, we illustrate the module graphs of three workers of StreamTeam-Football's analysis workflow (see Section 9.2) in this appendix.


fieldObjectState, matchMetadata
Field Object State Filter Module
fieldObjectState
Player Object State Filter Module
Active Keys Processor Module
playerObjectState
Heatmap Constr. Module
fieldObjectState, matchMetadata
Match Metadata Filter Module
matchMetadata
Match Metadata Store Module
Active Keys Window Module
internalActiveKeys
Heatmap Sender Module
heatmapStatistics

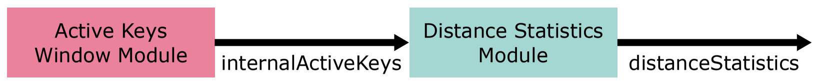
(a) Single Element Processor Graph

(b) Window Graph

Figure C.1 **Module Graphs of the Heatmap Worker.** (a) and (b) show the single element processor graph and the window graph of the heatmap worker (see Section 9.2.1.14). The gray, red, and mint boxes represent the generic single element processor modules, the generic window modules, and the worker-specific single element processor modules, respectively. The black arrows show which modules have successor modules to which they hand over the output stream elements they generate, which modules generate no output stream elements at all, which modules generate the output stream elements of the worker, and which modules process the input stream elements of the worker. The labels below the arrows denote to which data stream subset(s) the input and/or output stream elements belong.

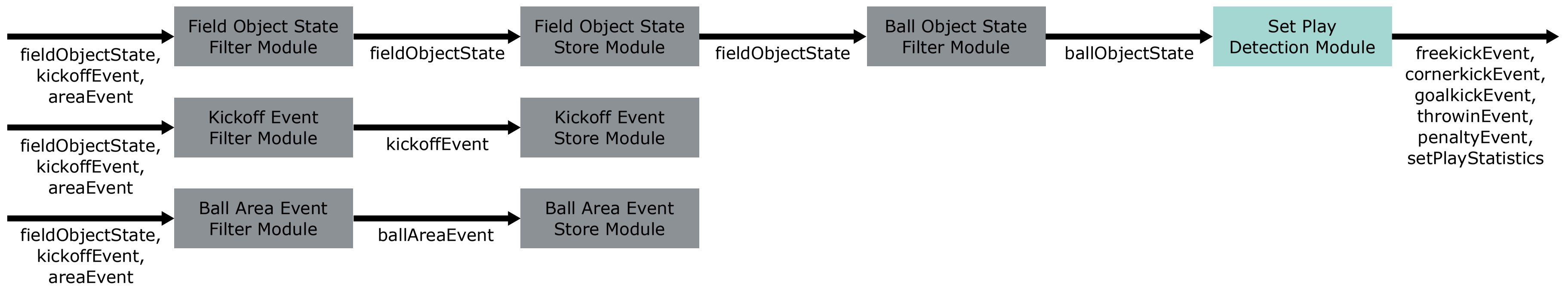

fieldObjectState,
ballPoss.Ch.Event
Field Object State Filter Module
fieldObjectState
Player Object State Filter Module
playerObjectState
Player Object State Store Module
playerObjectState
Drib. & Speed Event Detection Module
dribblingEvent,
speedLev.Ch.Event,
dribblingStatistics,
speedLev.Statistics
Active Keys Processor Module
fieldObjectState,
ballPoss.Ch.Event
Ball Poss. Change Event Filter Module
ballPoss.Ch.Event
Ball Possession Store Module

(a) Single Element Processor Graph



(b) Window Graph

Figure C.2 **Module Graphs of the Distance and Speed Analysis Worker.** (a) and (b) show the single element processor graph and the window graph of the distance and speed analysis worker (see Section 9.2.1.12). The gray, red, and mint boxes represent the generic single element processor modules, the generic window modules, and the worker-specific single element processor modules, respectively. The black arrows show which modules have successor modules to which they hand over the output stream elements they generate, which modules generate no output stream elements at all, which modules generate the output stream elements of the worker, and which modules process the input stream elements of the worker. The labels below the arrows denote to which data stream subset(s) the input and/or output stream elements belong.



(a) Single Element Processor Graph

Empty

(b) Window Graph

Figure C.3 Module Graphs of the Set Play Detection Worker. (a) and (b) show the single element processor graph and the window graph of the set play detection worker (see Section 9.2.1.5). The gray and mint boxes represent the generic single element processor modules and the worker-specific single element processor modules, respectively. The black arrows show which modules have successor modules to which they hand over the output stream elements they generate, which modules generate no output stream elements at all, which modules generate the output stream elements of the worker, and which modules process the input stream elements of the worker. The labels below the arrows denote to which data stream subset(s) the input and/or output stream elements belong. The window graph is empty since the analysis subtask which the set play detection worker performs does not require time-triggered code execution.

D Additional Evaluation Data

In this appendix, we will present additional evaluation data which we have omitted in Chapter 10 in form of tables and graphs.

D.1 Qualitative Evaluation

Event	Number of StreamTeam-Football events	Number of Opta events
Successful pass	364	377
Interception	79	22
Freekick	4	8
Cornerkick	5	6
Throwin	42	26
Goalkick	2	5

Table D.1 Event Quantities. List how many events of each type are detected by StreamTeam-Football and extracted from the Opta F24 data feed (only first halftime, see Section 10.2.1.1).

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.00	0.00	0.25	0.25	0.25	0.50
2 s	0.00	0.00	0.25	0.25	0.25	0.75
3 s	0.00	0.00	0.25	0.25	0.25	0.75
4 s	0.00	0.00	0.25	0.25	0.25	0.75
5 s	0.00	0.00	0.25	0.25	0.25	0.75

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	1.00	1.00	0.88	0.88	0.88	0.75
2 s	1.00	1.00	0.88	0.88	0.88	0.62
3 s	1.00	1.00	0.88	0.88	0.88	0.62
4 s	1.00	1.00	0.88	0.88	0.88	0.62
5 s	1.00	1.00	0.88	0.88	0.88	0.62

(b) Missed Detection Percentage

Figure D.1 Qualitative Evaluation Results for Freekick Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the freekick events for different time and distance threshold combinations. All values are rounded to two decimals.

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.40	0.40	0.40	0.40	0.40	0.60
2 s	0.40	0.40	0.40	0.40	0.40	0.60
3 s	0.40	0.40	0.40	0.40	0.40	0.60
4 s	0.40	0.40	0.40	0.40	0.40	0.60
5 s	0.40	0.40	0.40	0.40	0.40	0.60

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.67	0.67	0.67	0.67	0.67	0.50
2 s	0.67	0.67	0.67	0.67	0.67	0.50
3 s	0.67	0.67	0.67	0.67	0.67	0.50
4 s	0.67	0.67	0.67	0.67	0.67	0.50
5 s	0.67	0.67	0.67	0.67	0.67	0.50

(b) Missed Detection Percentage

Figure D.2 Qualitative Evaluation Results for Cornerkick Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the cornerkick events for different time and distance threshold combinations. All values are rounded to two decimals.

	1 m	3 m	5 m	7 m	9 m	∞
1 s	0.00	0.00	0.00	0.00	0.00	0.00
2 s	0.00	0.00	0.00	0.00	0.00	0.00
3 s	0.00	0.00	0.00	0.00	0.00	0.00
4 s	0.00	0.00	0.00	0.00	0.00	1.00
5 s	0.00	0.00	0.00	0.00	0.00	1.00

(a) Correct Detection Percentage

	1 m	3 m	5 m	7 m	9 m	∞
1 s	1.00	1.00	1.00	1.00	1.00	1.00
2 s	1.00	1.00	1.00	1.00	1.00	1.00
3 s	1.00	1.00	1.00	1.00	1.00	1.00
4 s	1.00	1.00	1.00	1.00	1.00	0.80
5 s	1.00	1.00	1.00	1.00	1.00	0.80

(b) Missed Detection Percentage

Figure D.3 Qualitative Evaluation Results for Goalkick Events. (a) and (b) show the correct detection percentage and the missed detection percentage for the goalkick events for different time and distance threshold combinations. All values are rounded to two decimals.

D.2 Performance Evaluation

Metric	Query URL
Average process call duration	`http://10.34.58.65:9090/api/v1/query?query=avg(avg_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_ns[45m]))%20by%20(samza_job)`
Average window call duration	`http://10.34.58.65:9090/api/v1/query?query=avg(avg_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_ns[45m]))%20by%20(samza_job)`
Average aggregated number of process calls per second	`http://10.34.58.65:9090/api/v1/query?query=sum(((sum_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_calls[45m])-(min_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_calls[45m])*count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_calls[45m])))/count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_calls[45m]))/(count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:process_calls[45m])/2))%20by%20(samza_job)`
Average aggregated number of window calls per second	`http://10.34.58.65:9090/api/v1/query?query=sum(((sum_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_calls[45m])-(min_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_calls[45m])*count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_calls[45m])))/count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_calls[45m]))/(count_over_time(samza:org_apache_samza_container_SamzaContainerMetrics:window_calls[45m])/2))%20by%20(samza_job)`

Table D.2 **Prometheus Queries.** Queries which retrieve the values of some Samza metrics for our performance evaluation when they are issued after the analysis to the REST API of the Prometheus instance that the Samza Prometheus Exporter feeds during the analysis with the measurements which Samza emits.

Worker	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
TimeTask	574.78	1150.10	1724.32	2298.58	2874.37
FieldObjectStateGenerationTask	574.66	1149.16	1724.20	2298.72	2873.62
DistanceAndSpeedAnalysisTask	575.26	1150.31	1725.84	2300.80	2875.95
AreaDetectionTask	574.87	1149.27	1724.41	2298.71	2873.19
PressingAnalysisTask	574.97	1149.74	1725.12	2300.34	2876.39
SetPlayDetectionTask	577.40	1154.72	1732.56	2309.72	2886.70
KickDetectionTask	581.67	1162.95	1745.11	2326.23	2908.71
TeamAreaTask	574.65	1149.41	1724.06	2299.91	2874.20
PassAndShotDetectionTask	3.04	6.10	9.12	12.14	15.21
HeatmapTask	574.83	1149.70	1724.29	2299.88	2873.50
PassCombinationDetectionTask	2.74	5.48	8.23	10.60	13.71
KickoffDetectionTask	574.58	1149.50	1724.08	2299.60	2874.33
OffsideTask	575.09	1150.67	1725.47	2300.74	2874.88
BallPossessionTask	577.40	1154.42	1732.10	2309.64	2886.06

Table D.3 Average Aggregated Number of Process Calls per Second for Single Container and Single Partition Configurations. The table lists the average aggregated number of process calls per second during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names end with “Task” since the table lists the class names and ending the worker class file names with “Task” is Samza’s naming convention [Sam17a]. All values are rounded to two decimals.

Worker	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
TimeTask	0.00	0.00	0.00	0.00	0.00
FieldObjectStateGenerationTask	0.00	0.00	0.00	0.00	0.00
DistanceAndSpeedAnalysisTask	0.33	0.33	0.33	0.33	0.33
AreaDetectionTask	0.00	0.00	0.00	0.00	0.00
PressingAnalysisTask	5.00	5.00	5.00	5.00	5.00
SetPlayDetectionTask	0.00	0.00	0.00	0.00	0.00
KickDetectionTask	0.00	0.00	0.00	0.00	0.00
TeamAreaTask	0.00	0.00	0.00	0.00	0.00
PassAndShotDetectionTask	0.00	0.00	0.00	0.00	0.00
HeatmapTask	1.00	0.99	0.99	0.98	0.98
PassCombinationDetectionTask	0.00	0.00	0.00	0.00	0.00
KickoffDetectionTask	0.00	0.00	0.00	0.00	0.00
OffsideTask	0.00	0.00	0.00	0.00	0.00
BallPossessionTask	1.00	1.00	1.00	1.00	1.00

Table D.4 Average Aggregated Number of Window Calls per Second for Single Container and Single Partition Configurations. The table lists the average aggregated number of window calls per second during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names end with "Task" since the table lists the class names and ending the worker class file names with "Task" is Samza's naming convention [Sam17a]. All values are rounded to two decimals.

Worker	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
TimeTask	17294.99 ns	20854.45 ns	15276.39 ns	18094.32 ns	15550.92 ns
FieldObjectStateGenerationTask	34894.22 ns	41486.53 ns	41231.27 ns	41058.01 ns	28967.67 ns
DistanceAndSpeedAnalysisTask	42608.03 ns	42182.41 ns	47541.49 ns	36707.31 ns	38873.39 ns
AreaDetectionTask	32543.74 ns	117182.66 ns	137997.59 ns	128360.20 ns	127403.70 ns
PressingAnalysisTask	71389.92 ns	72865.86 ns	66546.31 ns	66451.94 ns	103581.73 ns
SetPlayDetectionTask	28937.25 ns	26086.61 ns	23085.83 ns	23722.37 ns	22349.54 ns
KickDetectionTask	20548.51 ns	19348.61 ns	18514.35 ns	17338.65 ns	12678.45 ns
TeamAreaTask	52138.93 ns	51004.01 ns	41034.54 ns	42043.38 ns	49361.44 ns
PassAndShotDetectionTask	393785.35 ns	336047.63 ns	168563.04 ns	198181.09 ns	165030.69 ns
HeatmapTask	24341.29 ns	25469.79 ns	23839.21 ns	22353.16 ns	18291.95 ns
PassCombinationDetectionTask	275904.56 ns	191670.77 ns	221305.20 ns	127831.92 ns	121876.20 ns
KickoffDetectionTask	21569.39 ns	24298.08 ns	24499.88 ns	21719.41 ns	17772.37 ns
OffsideTask	26731.74 ns	26801.36 ns	20959.83 ns	26966.92 ns	19744.24 ns
BallPossessionTask	30752.89 ns	31281.77 ns	31112.66 ns	27565.05 ns	22137.36 ns

Table D.5 Average Duration of a Process Call for Single Container and Single Partition Configurations. The table lists the average duration of a process call in nanoseconds during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names end with “Task” since the table lists the class names and ending the worker class file names with “Task” is Samza’s naming convention [Sam17a]. All values are rounded to two decimals.

Worker	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
TimeTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
FieldObjectStateGenerationTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
DistanceAndSpeedAnalysisTask	422335.47 ns	519215.38 ns	920942.31 ns	2769727.99 ns	3504605.05 ns
AreaDetectionTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
PressingAnalysisTask	128069.54 ns	129940.29 ns	137041.89 ns	133854.87 ns	147654.34 ns
SetPlayDetectionTas	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
KickDetectionTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
TeamAreaTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
PassAndShotDetectionTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
HeatmapTask	12286815.51 ns	27666623.47 ns	43685761.69 ns	56589221.82 ns	65604787.85 ns
PassCombinationDetectionTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
KickoffDetectionTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
OffsideTask	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
BallPossessionTask	238759.99 ns	325126.69 ns	409971.35 ns	466909.15 ns	464486.83 ns

Table D.6 Average Duration of a Window Call for Single Container and Single Partition Configurations. The table lists the average duration of a window call in nanoseconds during the analysis (or more precisely the last 45 minutes of the analysis) for each Samza job and thus StreamTeam worker for an increasing number of concurrent matches. All worker names end with “Task” since the table lists the class names and ending the worker class file names with “Task” is Samza’s naming convention [Sam17a]. All values are rounded to two decimals.

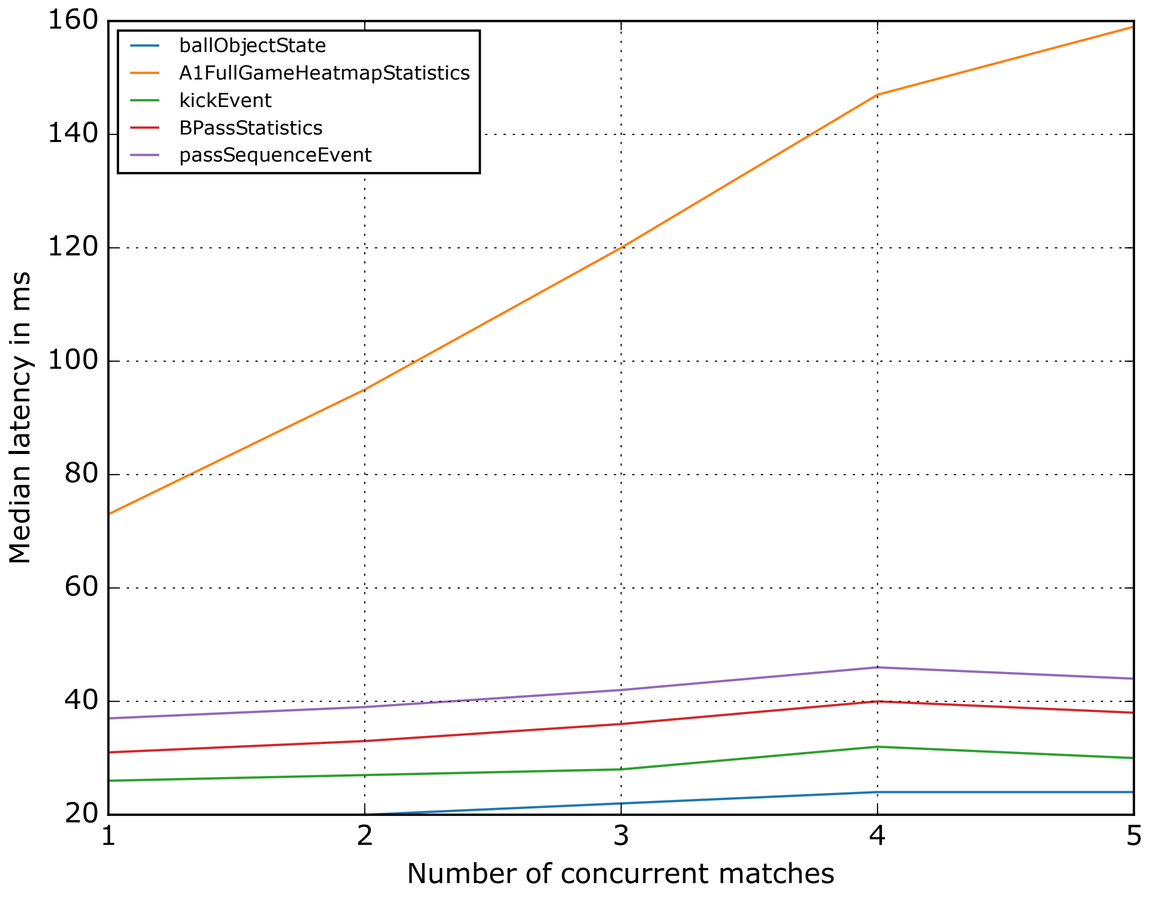


Figure D.4 Median Latencies for Single Container and Single Partition Configurations. The figure shows the median latencies for the five data stream subsets for an increasing number of concurrent matches.

Data stream subset	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
ballObjectState	20.00 ms	20.00 ms	22.00 ms	24.00 ms	24.00 ms
A1FullGameHeatmapStatistics	73.00 ms	95.00 ms	120.00 ms	147.00 ms	159.00 ms
kickEvent	26.00 ms	27.00 ms	28.00 ms	32.00 ms	30.00 ms
BPassStatistics	31.00 ms	33.00 ms	36.00 ms	40.00 ms	38.00 ms
passSequenceEvent	37.00 ms	39.00 ms	42.00 ms	46.00 ms	44.00 ms

Table D.7 Median Latencies for Single Container and Single Partition Configurations. The table lists the median latencies for the five data stream subsets for an increasing number of concurrent matches. All values are rounded to two decimals.

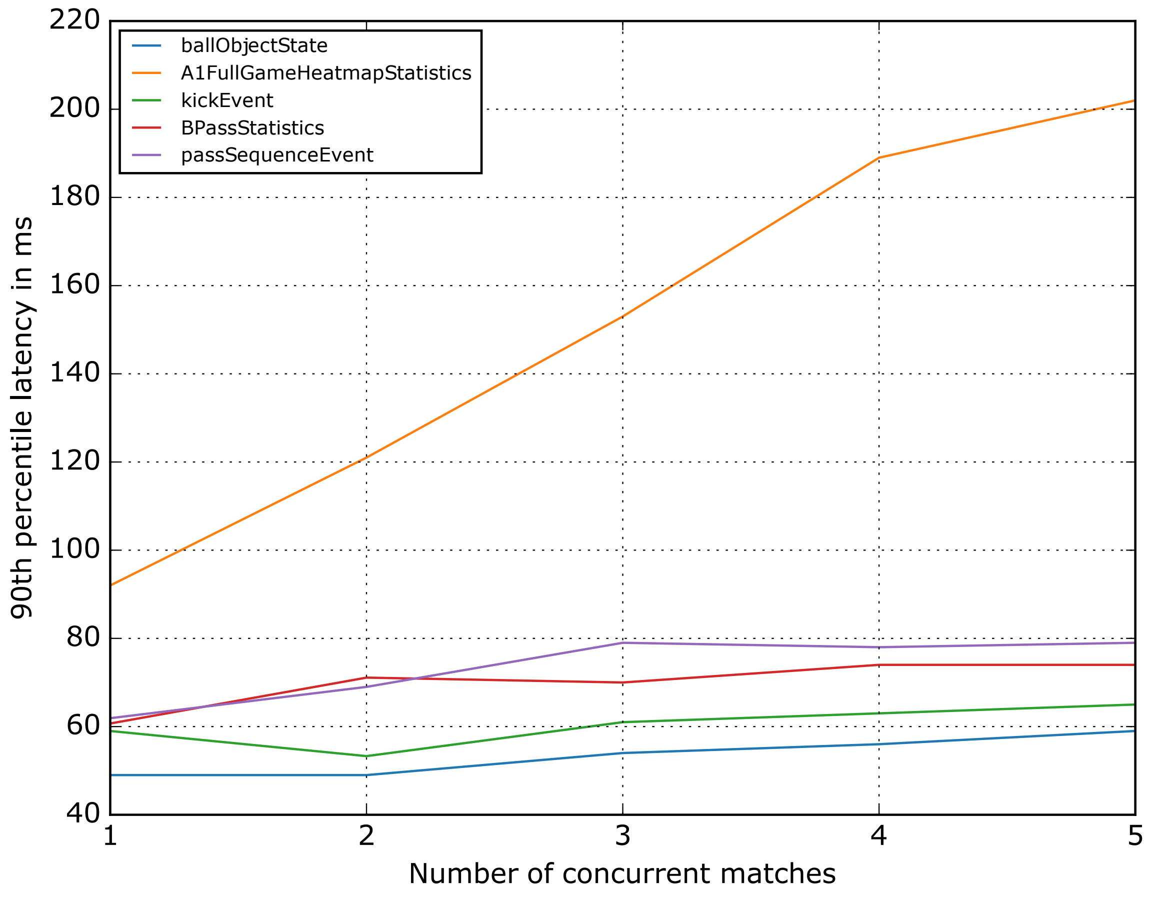


Figure D.5 90th Percentile Latencies for Single Container and Single Partition Configurations. The figure shows the 90th percentile latencies for the five data stream subsets for an increasing number of concurrent matches.

Data stream subset	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
ballObjectState	49.00 ms	49.00 ms	54.00 ms	56.00 ms	59.00 ms
A1FullGameHeatmapStatistics	92.00 ms	121.00 ms	153.00 ms	189.00 ms	202.00 ms
kickEvent	59.00 ms	53.3.00 ms	61.00 ms	63.00 ms	65.00 ms
BPassStatistics	60.70 ms	71.10 ms	70.00 ms	74.00 ms	74.00 ms
passSequenceEvent	61.90 ms	69.00 ms	79.00 ms	78.00 ms	79.00 ms

Table D.8 90th Percentile Latencies for Single Container and Single Partition Configurations. The table lists the 90th percentile latencies for the five data stream subsets for an increasing number of concurrent matches. All values are rounded to two decimals.

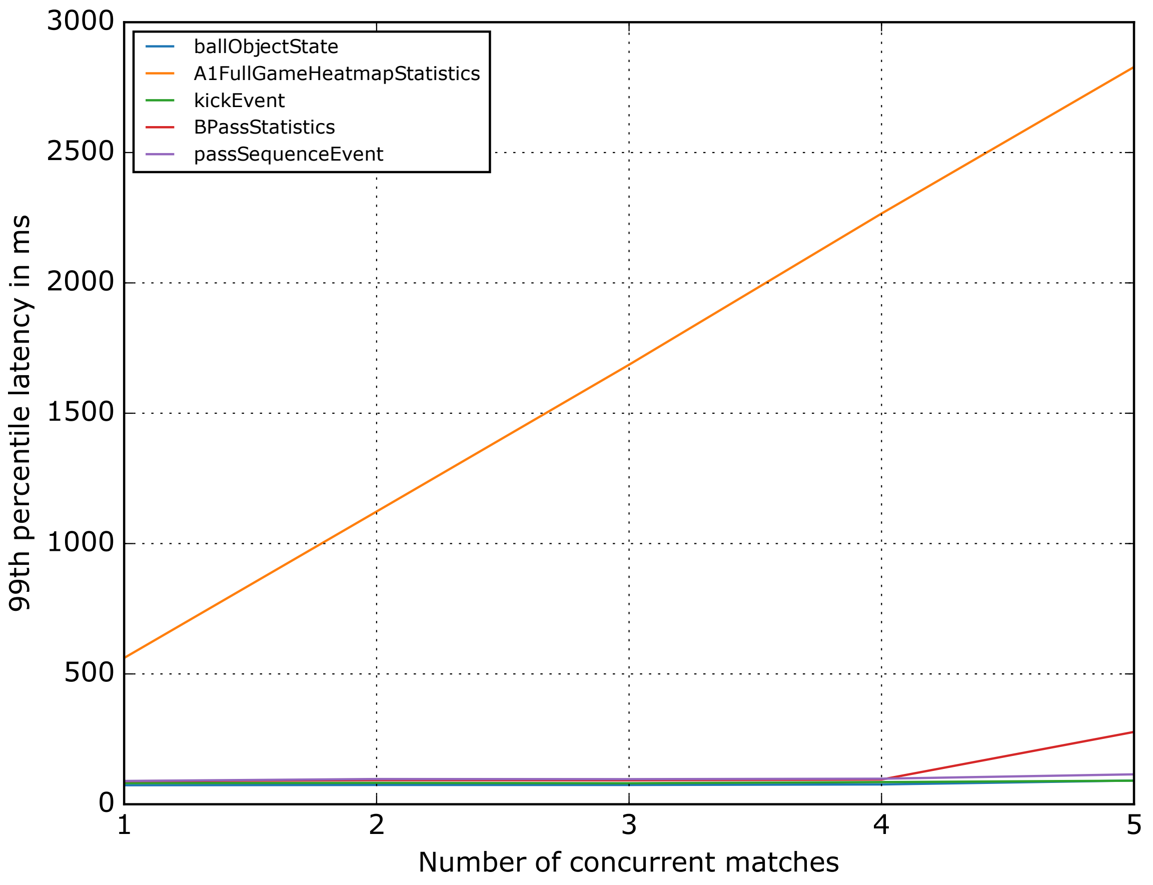


Figure D.6 99th Percentile Latencies for Single Container and Single Partition Configurations. The figure shows the 99th percentile latencies for the five data stream subsets for an increasing number of concurrent matches.

Data stream subset	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
ballObjectState	73.00 ms	74.00 ms	74.00 ms	76.00 ms	91.00 ms
A1FullGameHeatmapStatistics	561.24 ms	1123.00 ms	1686.13 ms	2266.00 ms	2828.12 ms
kickEvent	81.00 ms	82.00 ms	80.60 ms	85.00 ms	90.45 ms
BPassStatistics	88.71 ms	92.42 ms	91.77 ms	94.86 ms	277.45 ms
passSequenceEvent	89.39 ms	96.80 ms	96.26 ms	98.00 ms	114.76 ms

Table D.9 99th Percentile Latencies for Single Container and Single Partition Configurations. The table lists the 99th percentile latencies for the five data stream subsets for an increasing number of concurrent matches. All values are rounded to two decimals.

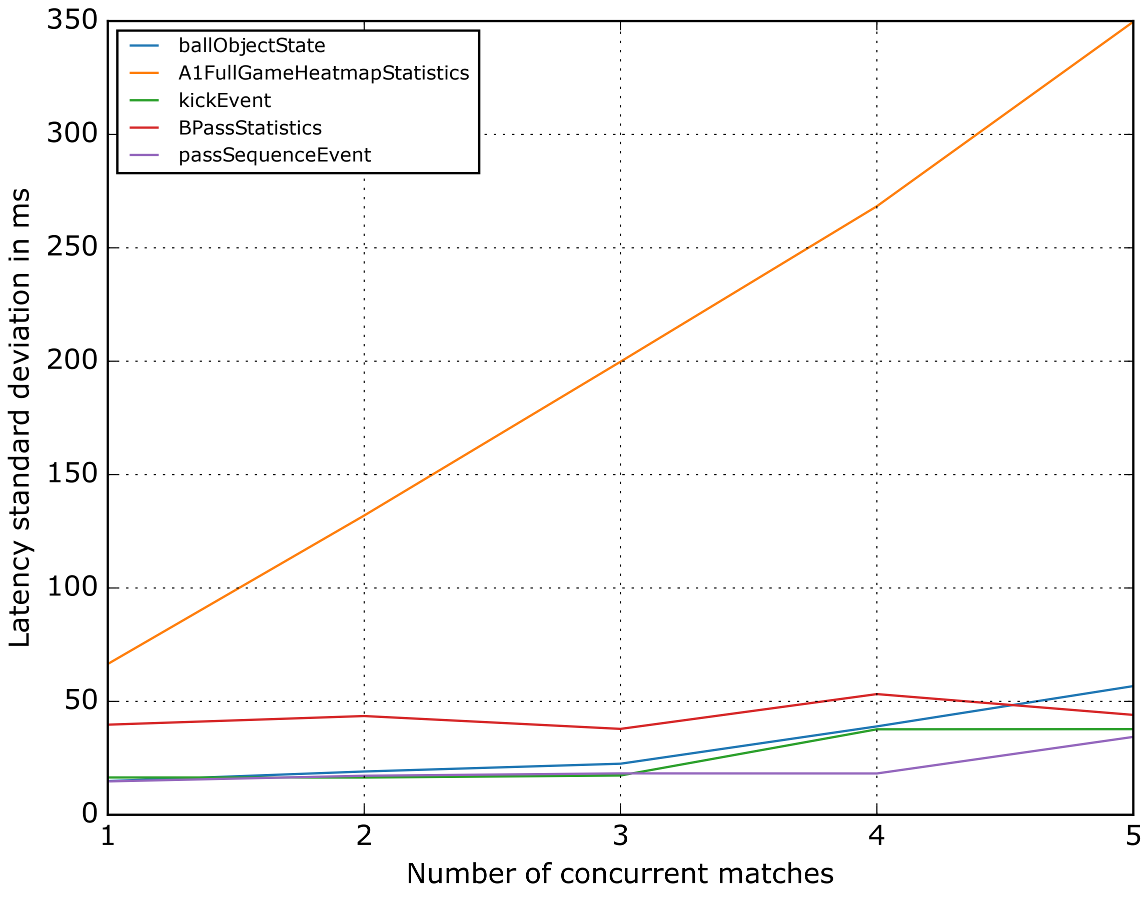
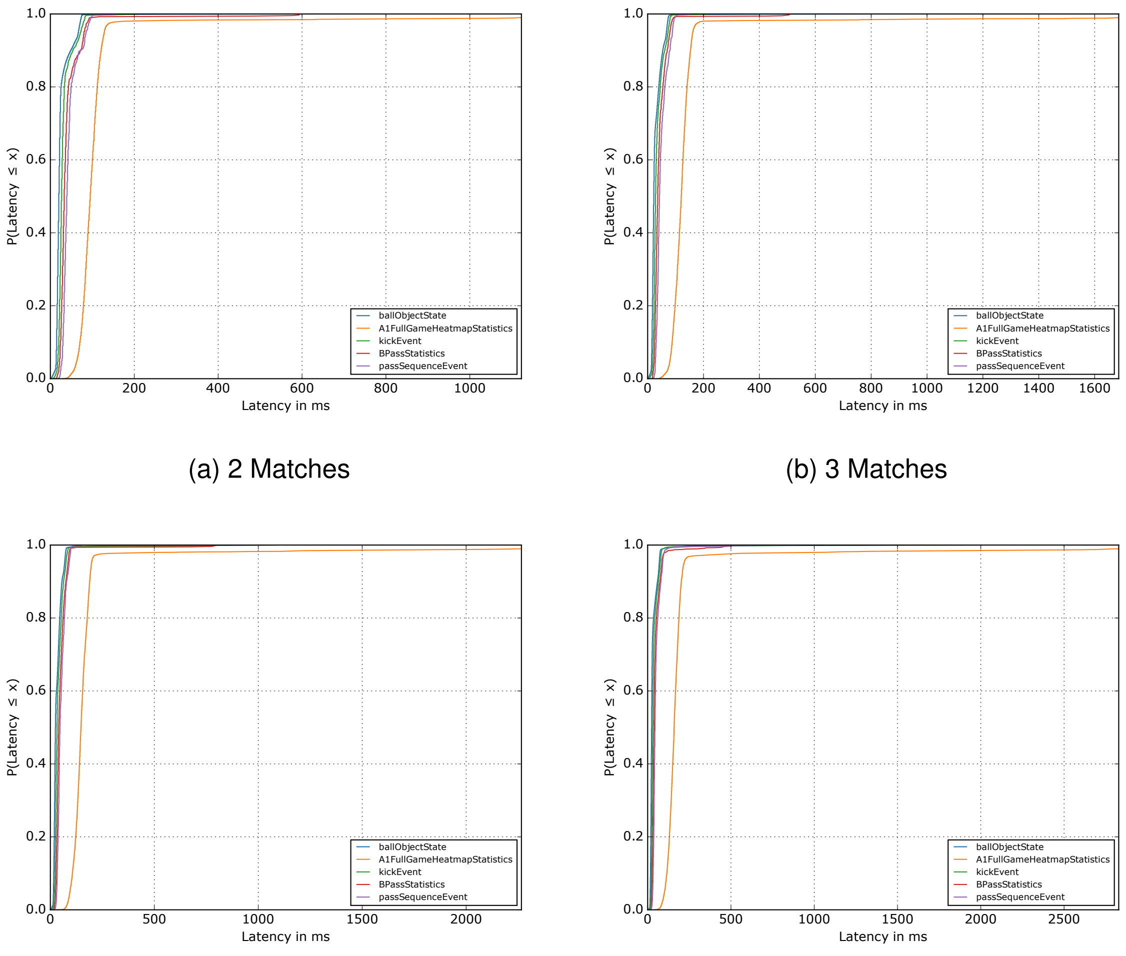


Figure D.7 Latency Standard Deviations for Single Container and Single Partition Configurations. The figure shows the latency standard deviations for the five data stream subsets for an increasing number of concurrent matches.

Data stream subset	1 Match	2 Matches	3 Matches	4 Matches	5 Matches
ballObjectState	14.86 ms	19.08 ms	22.52 ms	38.98 ms	56.74 ms
A1FullGameHeatmapStatistics	66.49 ms	131.95 ms	199.80 ms	268.34 ms	349.69 ms
kickEvent	16.45 ms	16.41 ms	17.32 ms	37.69 ms	37.76 ms
BPassStatistics	39.73 ms	43.54 ms	37.87 ms	53.21 ms	44.05 ms
passSequenceEvent	14.71 ms	17.21 ms	18.25 ms	18.21 ms	34.33 ms

Table D.10 Latency Standard Deviations for Single Container and Single Partition Configurations. The table lists the latency standard deviations for the five data stream subsets for an increasing number of concurrent matches. All values are rounded to two decimals.



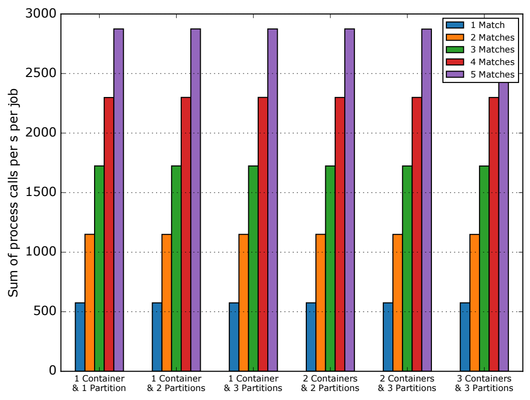
(a) 2 Matches

(b) 3 Matches

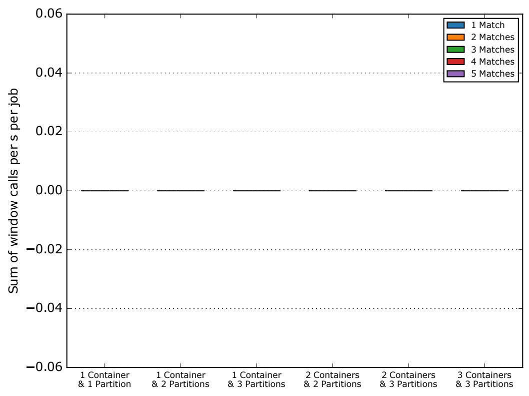
(c) 4 Matches

(d) 5 Matches

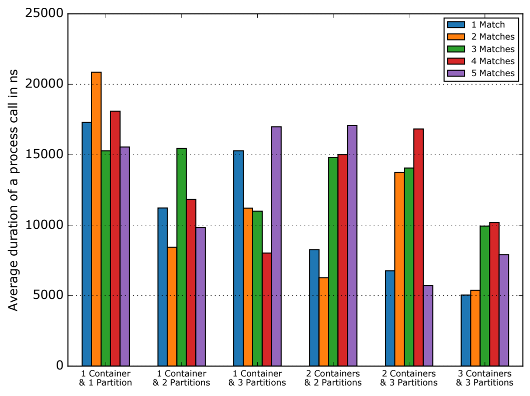
Figure D.8 Latency CDFs for Single Container and Single Partition Configurations. The figure shows the empirical CDFs of the latencies measured for the elements of the five data stream subsets for an increasing number of concurrent matches. For the sake of keeping the figure readable the CDFs are capped at the 99th percentile. The CDFs for the single match configuration are illustrated in Figure 10.7.



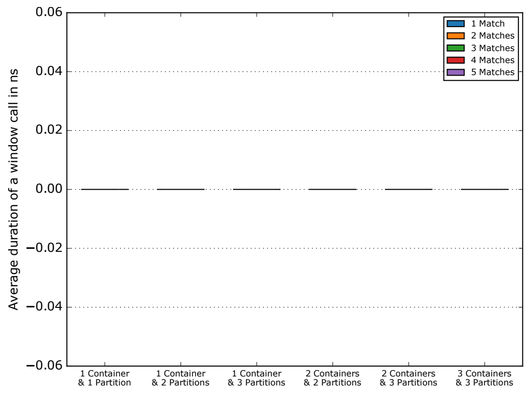
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

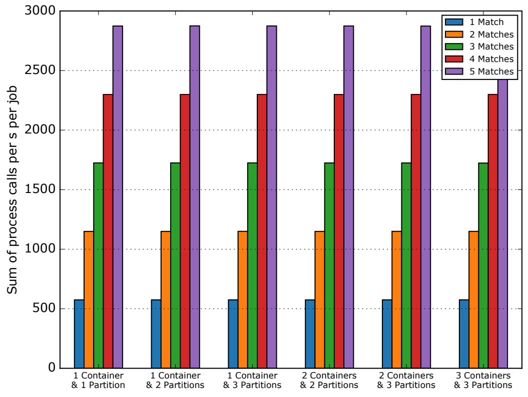


(c) Average Duration of a Process Call

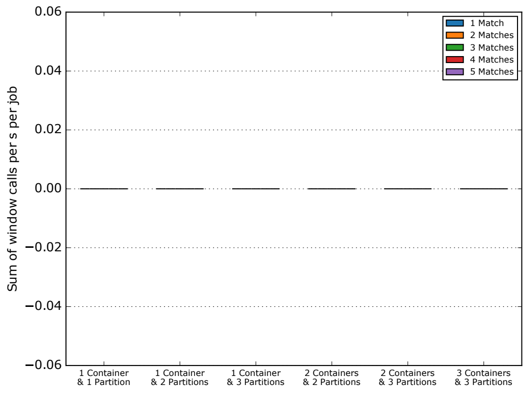


(d) Average Duration of a Window Call

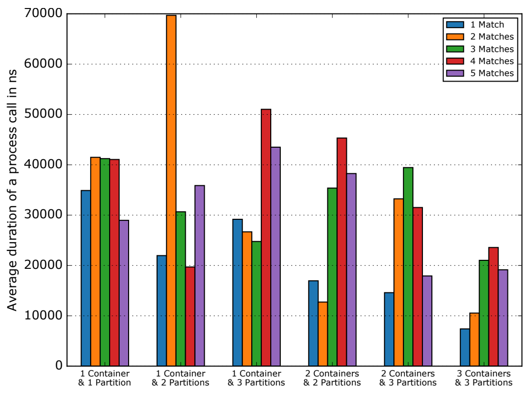
Figure D.9 Samza Metrics for all Configurations for the Time Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the time worker for all configurations.



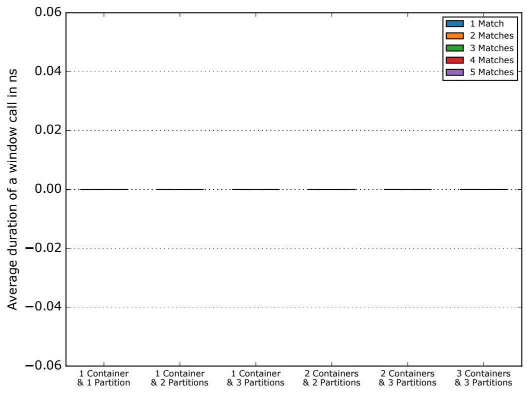
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

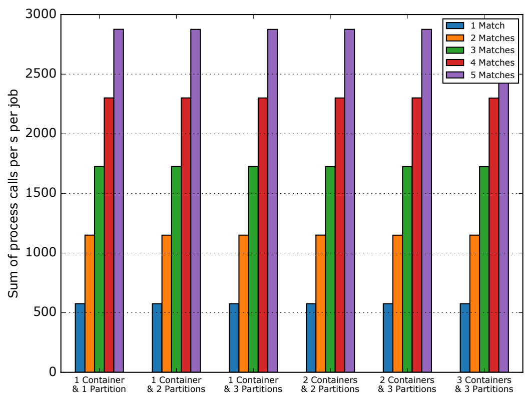


(c) Average Duration of a Process Call

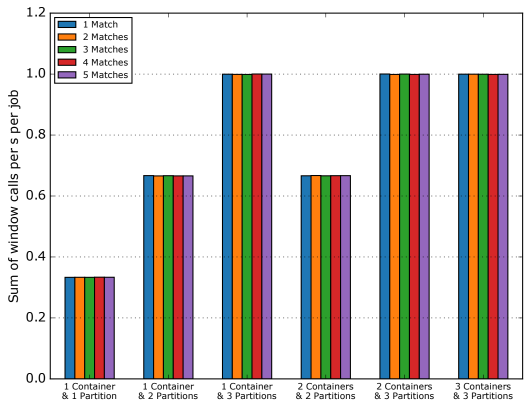


(d) Average Duration of a Window Call

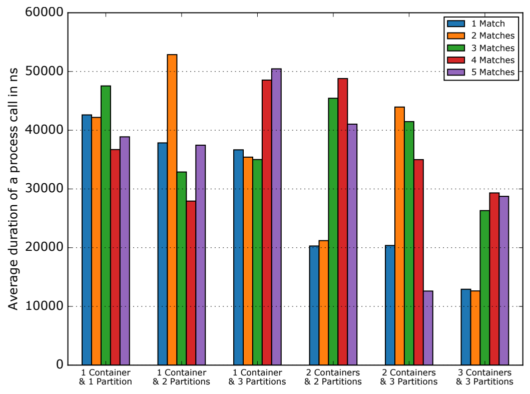
Figure D.10 Samza Metrics for all Configurations for the Field Object State Generation Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the field object state generation worker for all configurations.



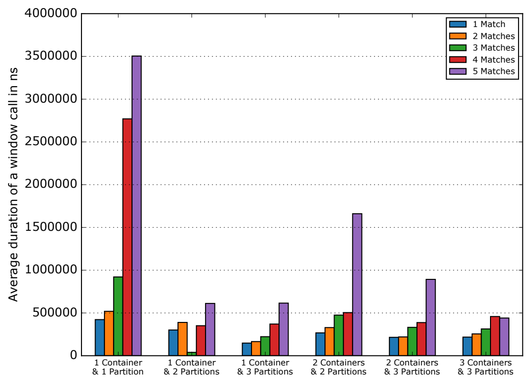
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

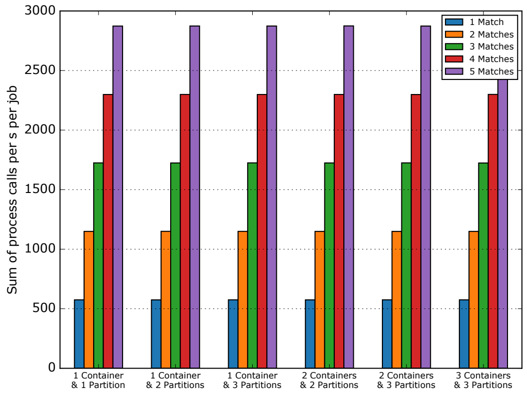


(c) Average Duration of a Process Call

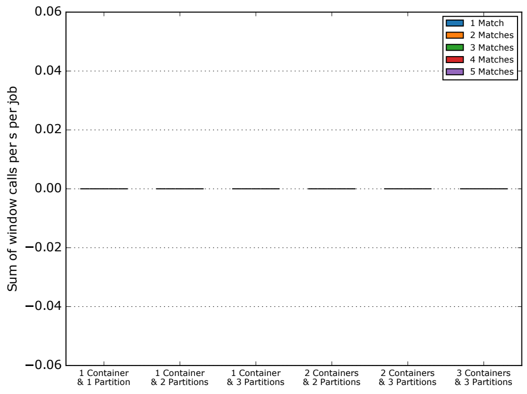


(d) Average Duration of a Window Call

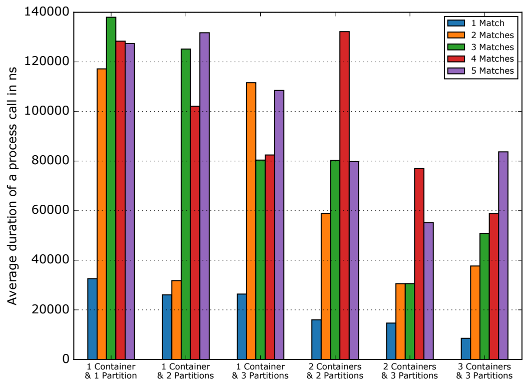
Figure D.11 Samza Metrics for all Configurations for the Distance and Speed Analysis Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the distance and speed analysis worker for all configurations.



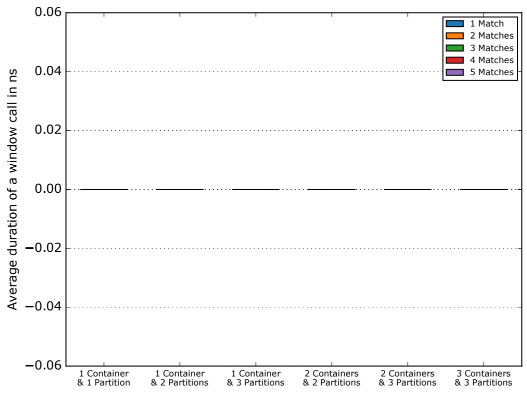
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

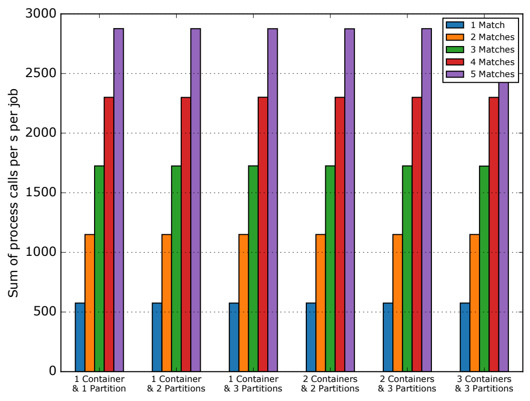


(c) Average Duration of a Process Call

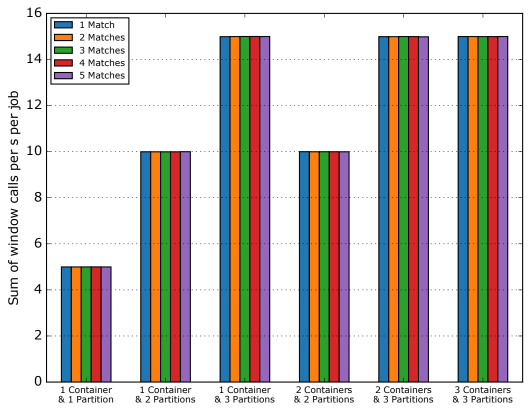


(d) Average Duration of a Window Call

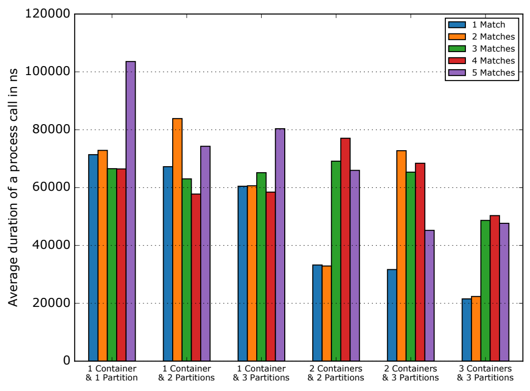
Figure D.12 Samza Metrics for all Configurations for the Area Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the area detection worker for all configurations.



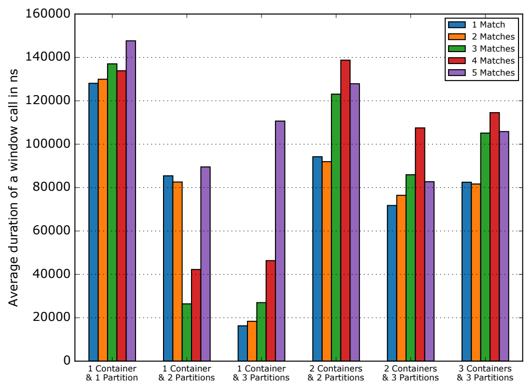
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

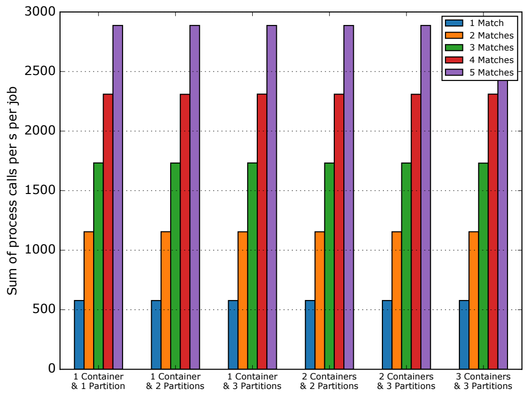


(c) Average Duration of a Process Call

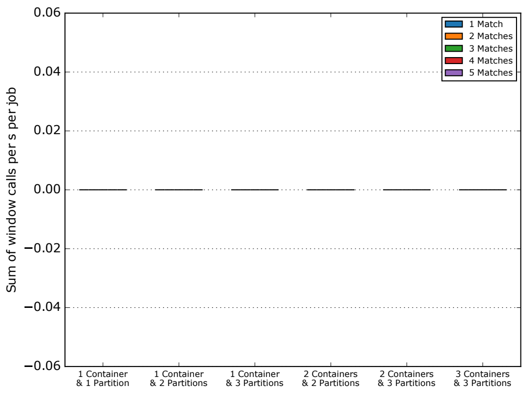


(d) Average Duration of a Window Call

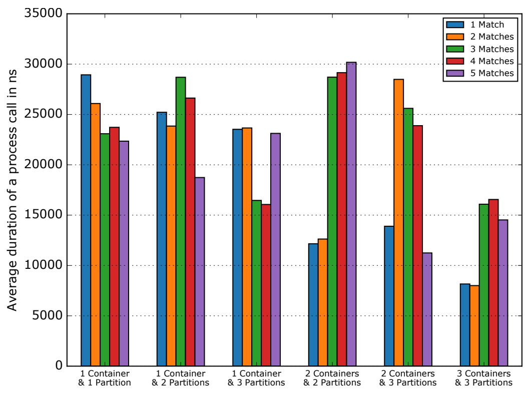
Figure D.13 Samza Metrics for all Configurations for the Pressing Analysis Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the pressing analysis worker for all configurations.



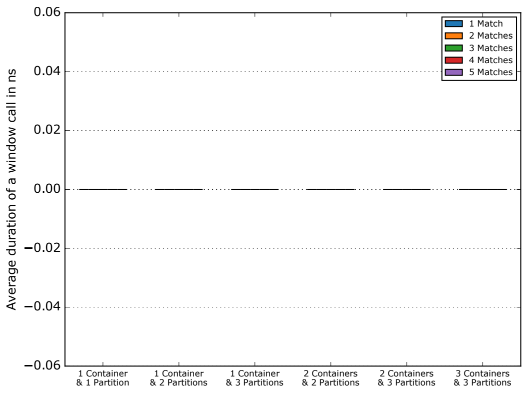
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

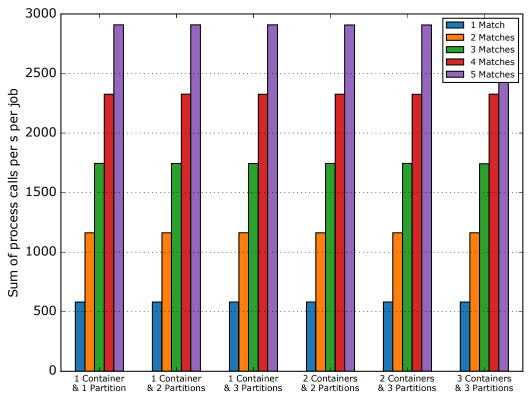


(c) Average Duration of a Process Call

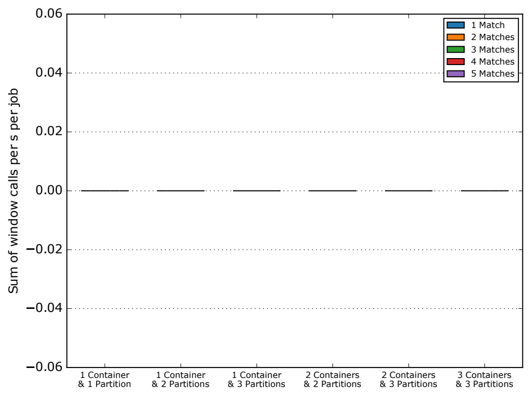


(d) Average Duration of a Window Call

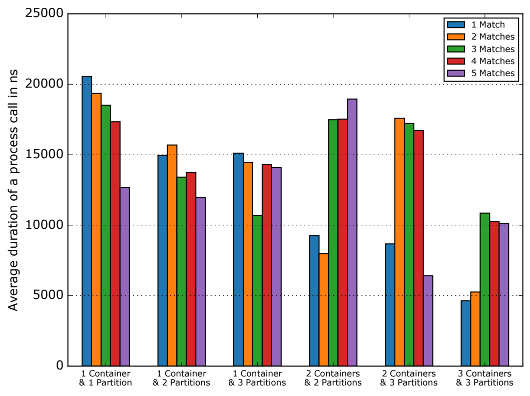
Figure D.14 Samza Metrics for all Configurations for the Set Play Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the set play detection worker for all configurations.



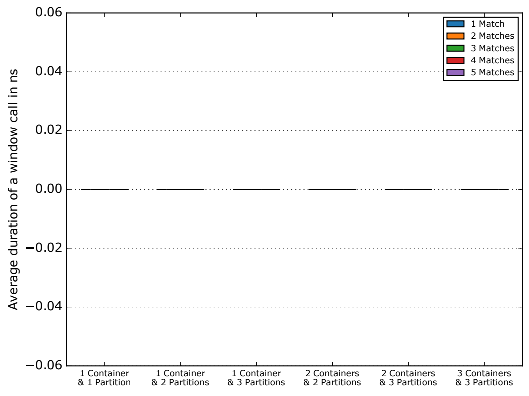
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

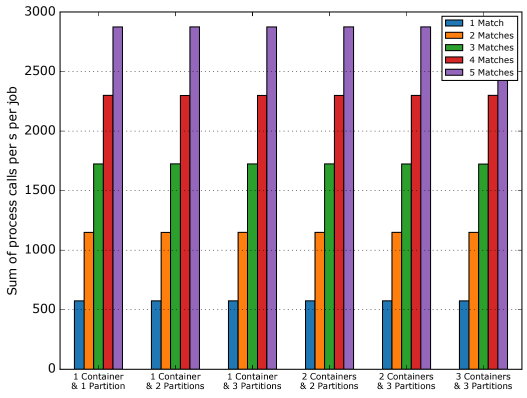


(c) Average Duration of a Process Call

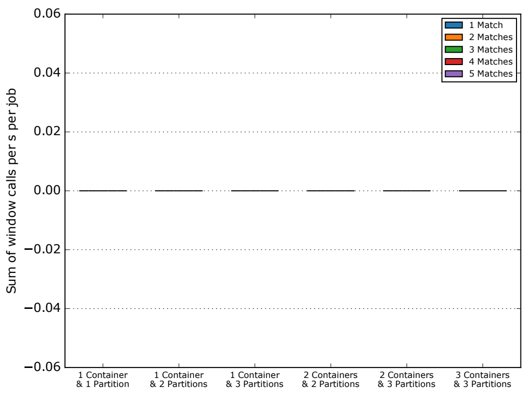


(d) Average Duration of a Window Call

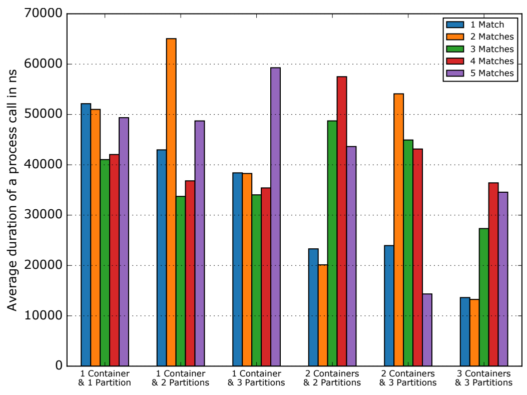
Figure D.15 Samza Metrics for all Configurations for the Kick Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the kick detection worker for all configurations.



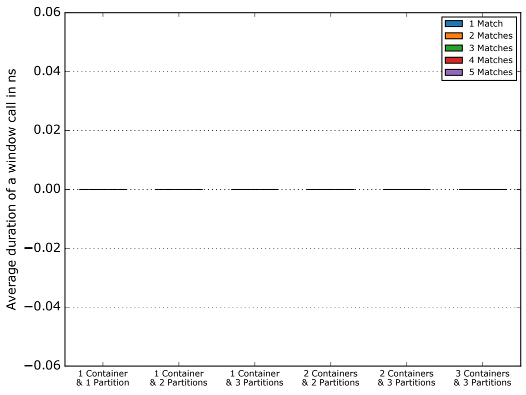
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

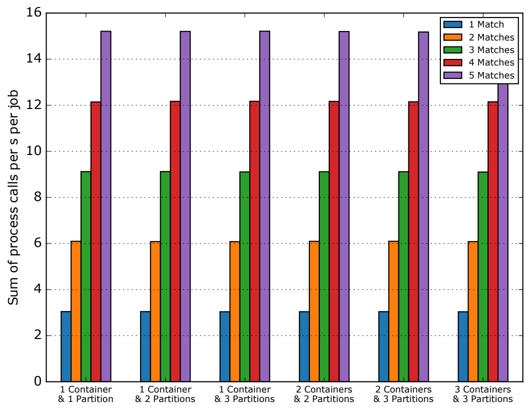


(c) Average Duration of a Process Call

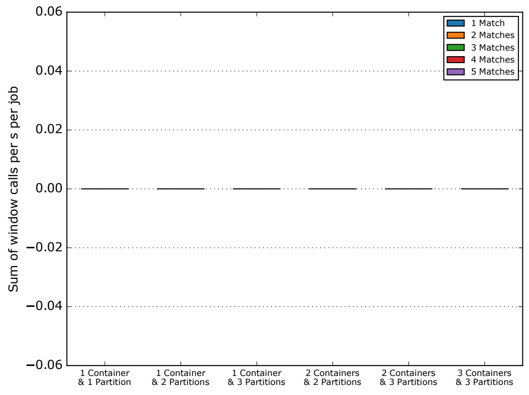


(d) Average Duration of a Window Call

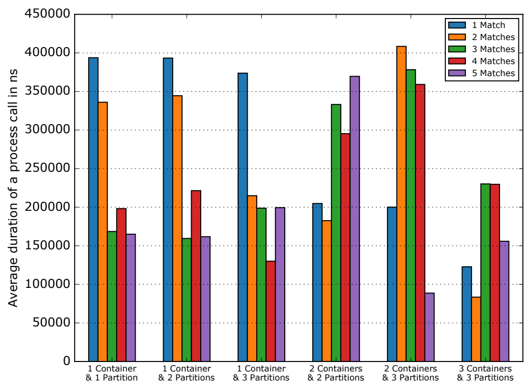
Figure D.16 Samza Metrics for all Configurations for the Team Area Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the team area worker for all configurations.



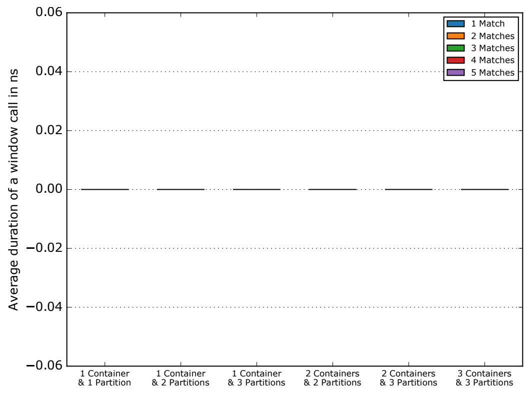
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

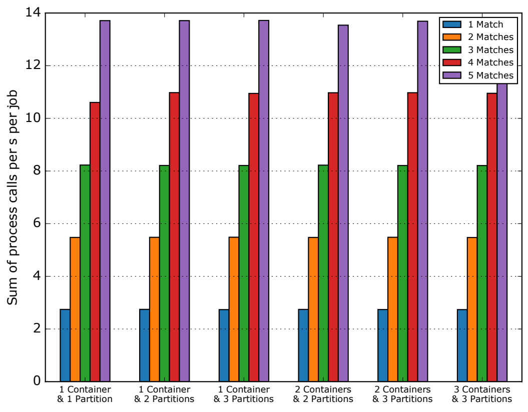


(c) Average Duration of a Process Call

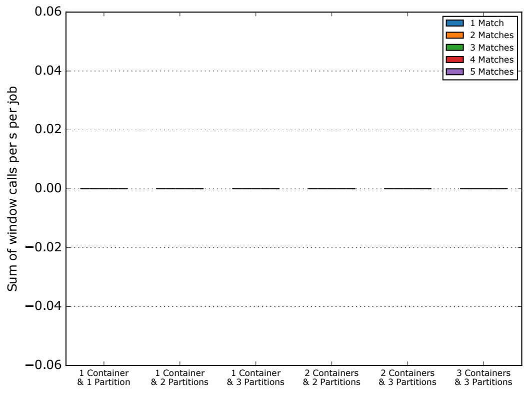


(d) Average Duration of a Window Call

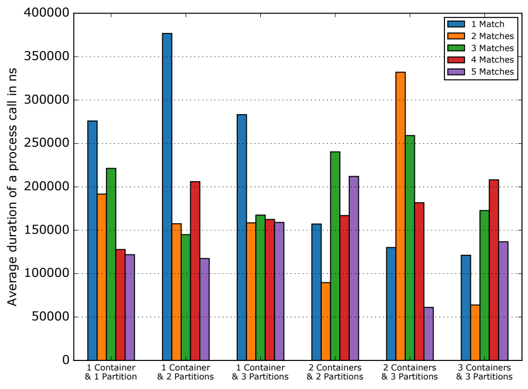
Figure D.17 Samza Metrics for all Configurations for the Pass and Shot Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the pass and shot detection worker for all configurations.



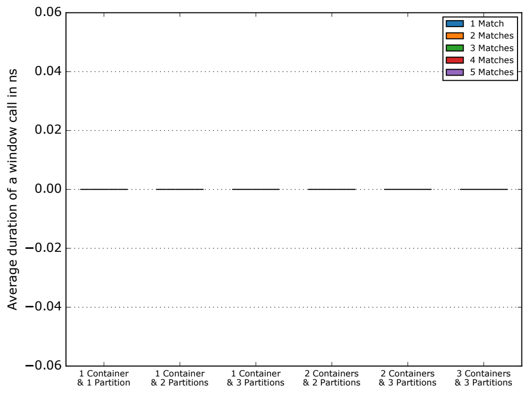
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

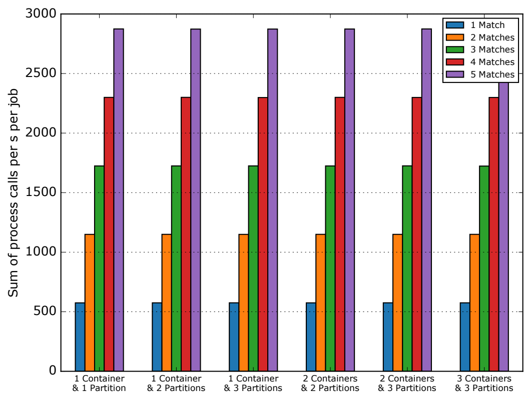


(c) Average Duration of a Process Call

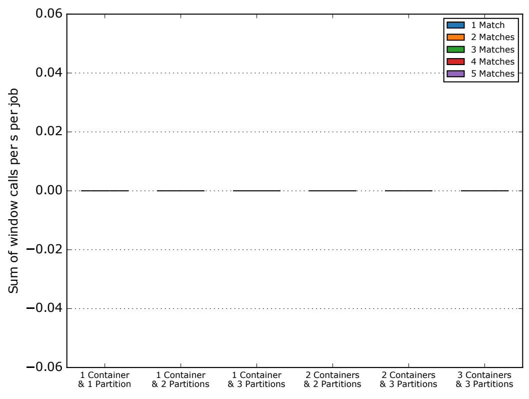


(d) Average Duration of a Window Call

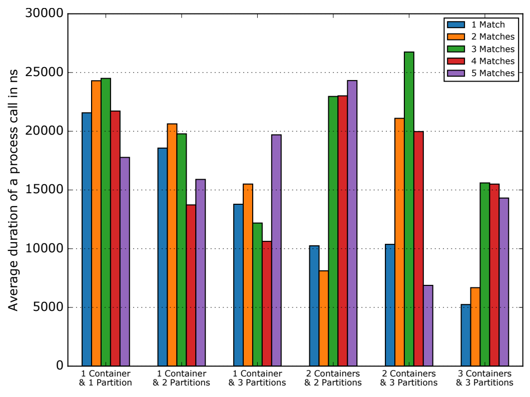
Figure D.18 Samza Metrics for all Configurations for the Pass Combination Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the pass combination detection worker for all configurations.



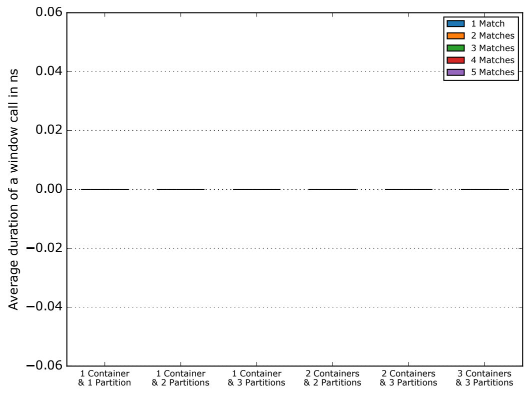
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

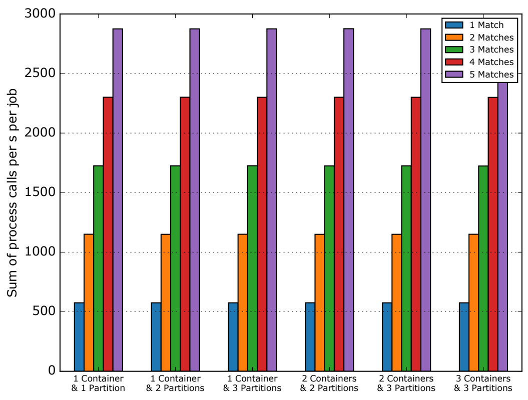


(c) Average Duration of a Process Call

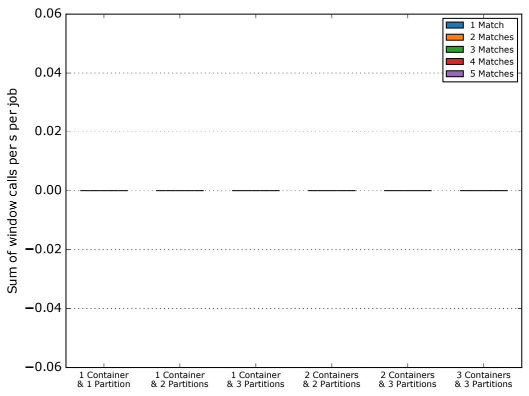


(d) Average Duration of a Window Call

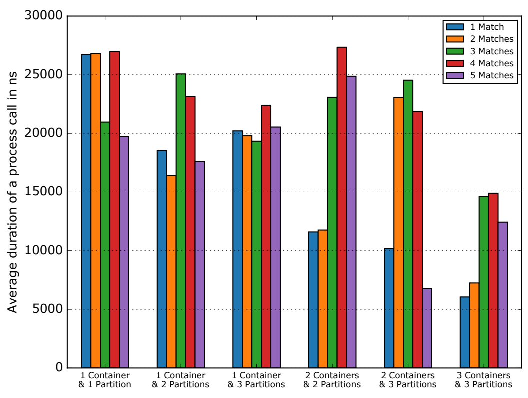
Figure D.19 Samza Metrics for all Configurations for the Kickoff Detection Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the kickoff detection worker for all configurations.



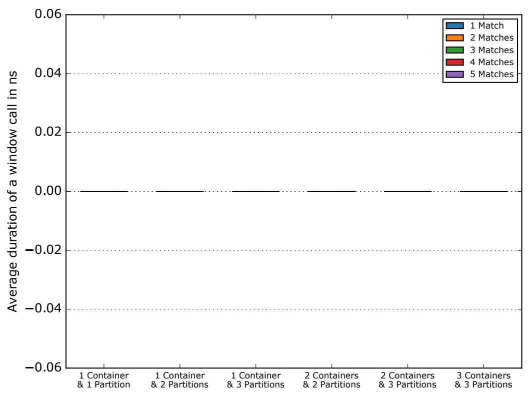
(a) Average Aggregated Number of Process Calls per Second



(b) Average Aggregated Number of Window Calls per Second

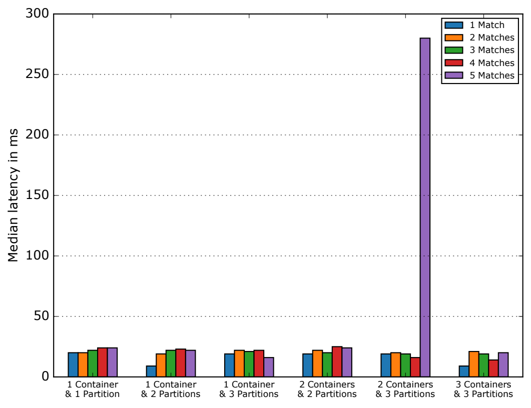


(c) Average Duration of a Process Call

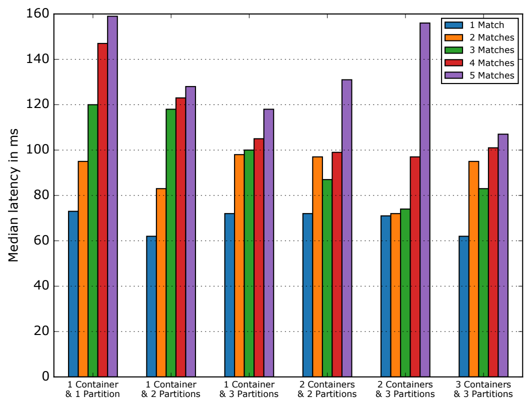


(d) Average Duration of a Window Call

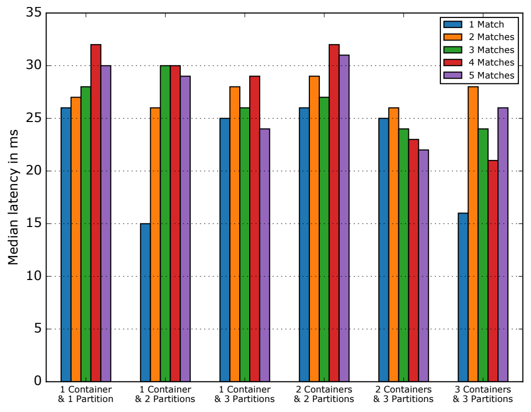
Figure D.20 Samza Metrics for all Configurations for the Offside Worker. (a), (b), (c), and (d) show the values of the four Samza metrics for the offside worker for all configurations.



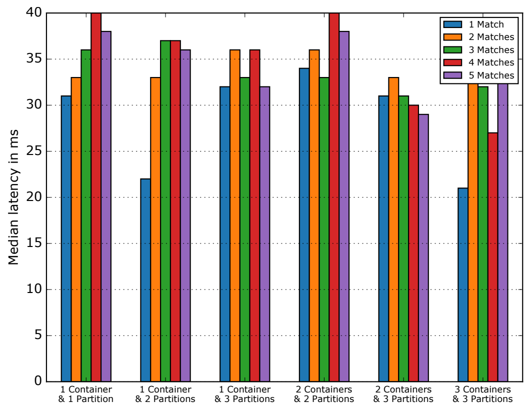
(a) ballObjectState



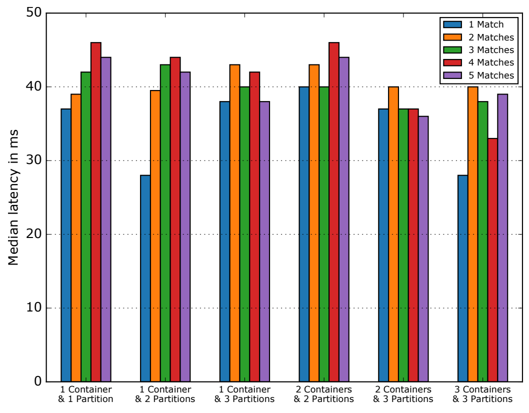
(b) A1FullGameHeatmapStatistics



(c) kickEvent

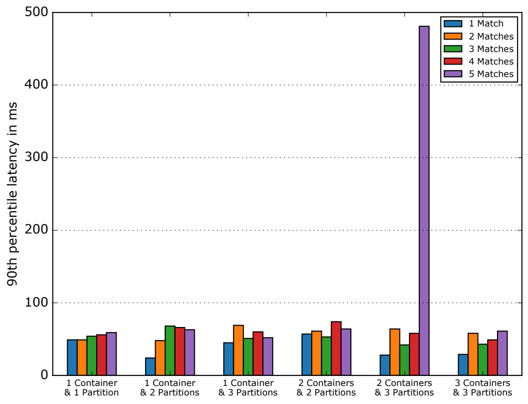


(d) BPassStatistics

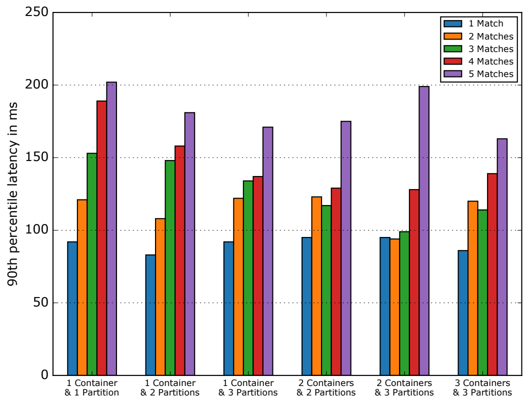


(e) passSequenceEvent

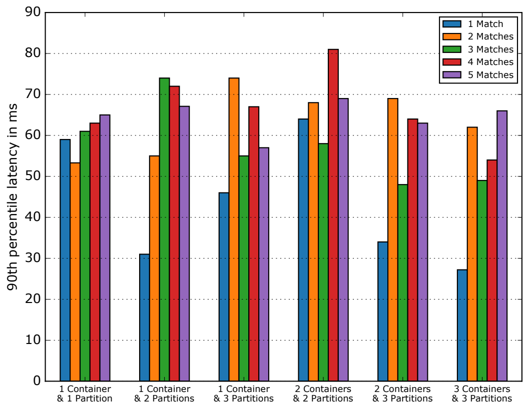
Figure D.21 Median Latencies for all Configurations. (a), (b), (c), (d), and (e) show the median latencies for the five data stream subsets for all configurations.



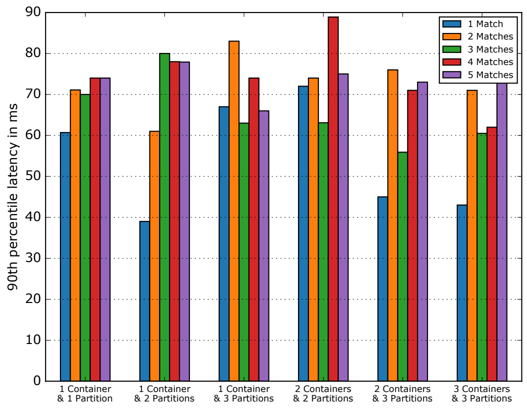
(a) ballObjectState



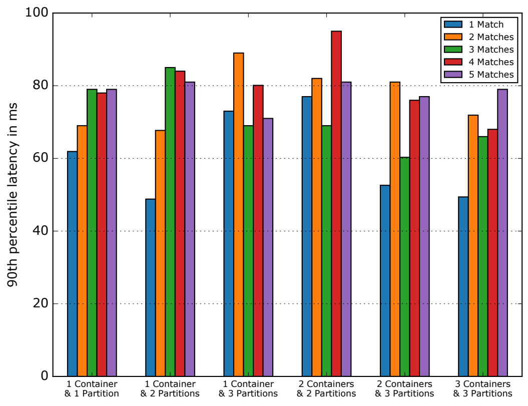
(b) A1FullGameHeatmapStatistics



(c) kickEvent

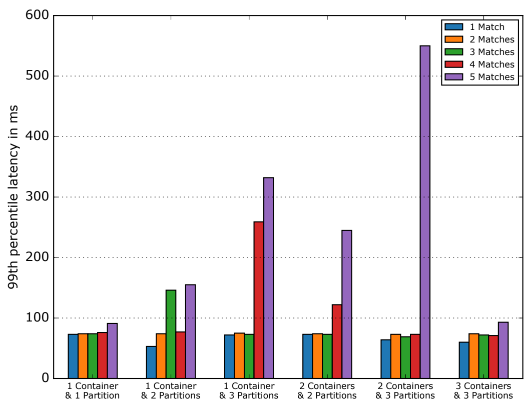


(d) BPassStatistics

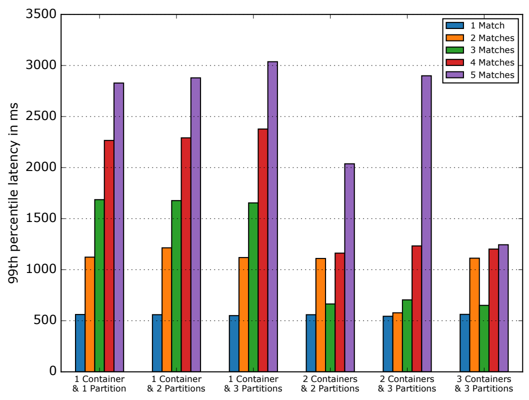


(e) passSequenceEvent

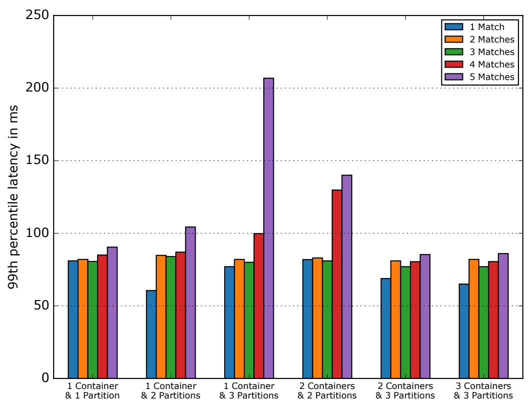
Figure D.22 90th Percentile Latencies for all Configurations. (a), (b), (c), (d), and (e) show the 90th percentile latencies for the five data stream subsets for all configurations.



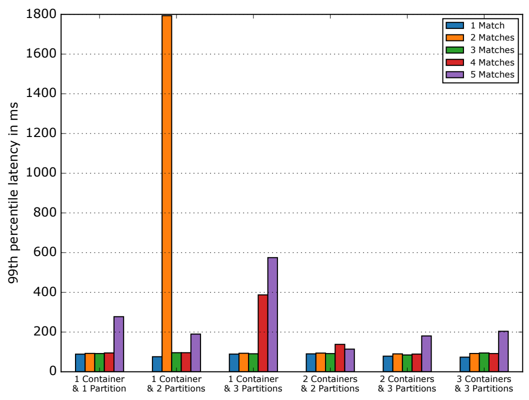
(a) ballObjectState



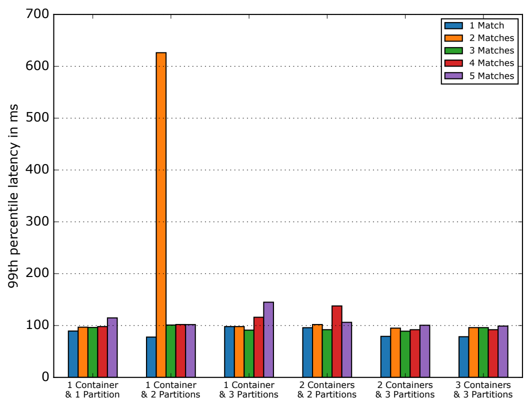
(b) A1FullGameHeatmapStatistics



(c) kickEvent

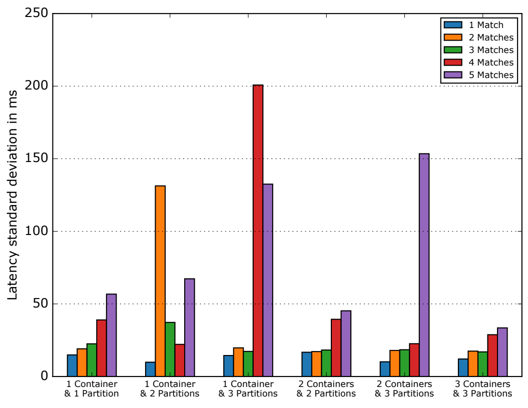


(d) BPassStatistics

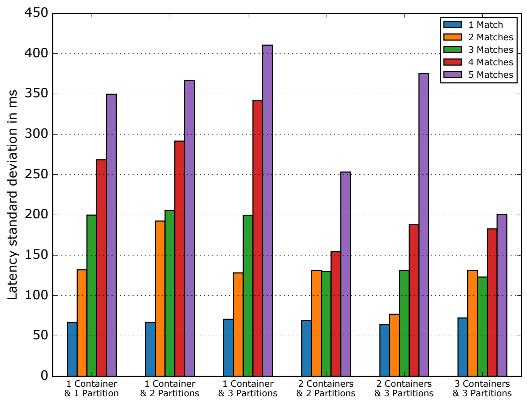


(e) passSequenceEvent

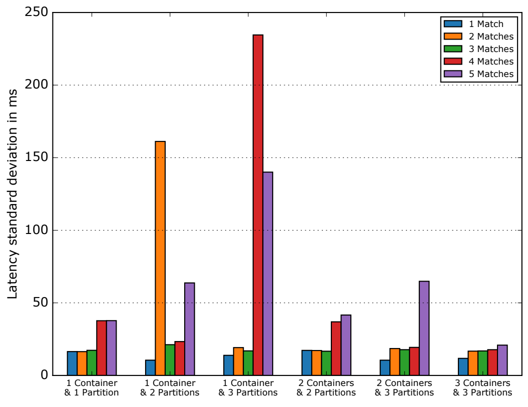
Figure D.23 99th Percentile Latencies for all Configurations. (a), (b), (c), (d), and (e) show the 99th percentile latencies for the five data stream subsets for all configurations.



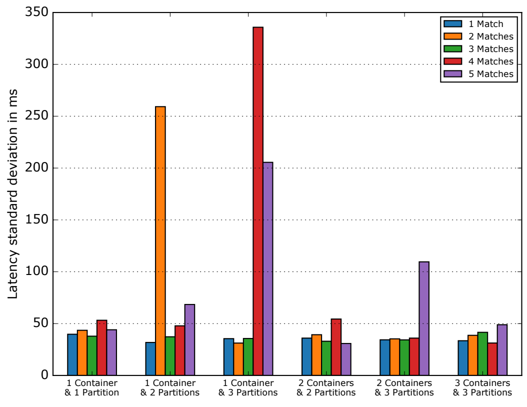
(a) ballObjectState



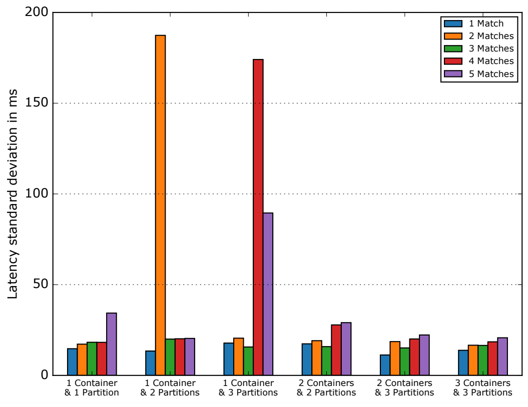
(b) A1FullGameHeatmapStatistics



(c) kickEvent



(d) BPassStatistics



(e) passSequenceEvent

Figure D.24 Latency Standard Deviations for all Configurations. (a), (b), (c), (d), and (e) show the latency standard deviations for the five data stream subsets for all configurations.

Bibliography

[ACÇ+03] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. DOI: `10.1007/s00778-003-0095-z`.

[AE04] Asaf Adi and Opher Etzion. Amit – the situation manager. *The VLDB journal*, 13(2):177–203, 2004. DOI: `10.1007/s00778-003-0108-y`.

[Adv20] Advanced Sport Instruments SA. FieldWiz. https://asi.swiss/products/fieldwiz/, 2020. (Last accessed: 02.04.2020).

[ATM+17] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. Defining the execution semantics of stream processing engines. *Journal of Big Data*, 4(1):12, 2017. DOI: `10.1186/s40537-017-0072-9`.

[AR15] Sarthak Agarwal and KS Rajan. Performance Analysis of MongoDB vs. PostGIS/PostGreSQL Databases for Line Intersection and Point Containment Spatial Queries. In *Proceedings of the Free and Open Source Software For Geospatial Conference (FOSS4G 15)*, pages 37–43, Seoul, South Korea, 2015.

[AÇT08] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based Complex Event Detection across Distributed Sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008. DOI: `10.14778/1453856.1453869`.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013. DOI: `10.14778/2536222.2536229`.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015. DOI: 10.14778/2824032.2824076.

[AS13a] Ihab Al Kabary and Heiko Schuldt. SportSense: Using Motion Queries to Find Scenes in Sports Videos. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management (CIKM 13)*, pages 2489–2492, San Francisco, CA, USA. ACM, 2013. DOI: 10.1145/2505515.2508211.

[AS13b] Ihab Al Kabary and Heiko Schuldt. Towards Sketch-based Motion Queries in Sports Videos. In *Proceedings of the 2013 IEEE International Symposium on Multimedia (ISM 13)*, pages 309–314, Anaheim, CA, USA. IEEE, 2013. DOI: 10.1109/ISM.2013.60.

[AS14a] Ihab Al Kabary and Heiko Schuldt. Enhancing Sketch-based Sport Video Retrieval by Suggesting Relevant Motion Paths. In *Proceedings of the 37th international ACM SIGIR Conference on Research & Development in Information Retrieval (SIGR 14)*, pages 1227–1230, Gold Coast, Queensland, Australia. ACM, 2014. DOI: 10.1145/2600428.2609551.

[AS14b] Ihab Al Kabary and Heiko Schuldt. Using Hand Gestures for Specifying Motion Queries in Sketch-Based Video Retrieval. In *Proceedings of the 36th European Conference on IR Research (ECIR 14)*, pages 733–736, Amsterdam, Netherlands. Springer, 2014. DOI: 10.1007/978-3-319-06028-6_84.

[ASA+17] Elias Alevizos, Anastasios Skarlatidis, Alexander Artikis, and Georgios Paliouras. Probabilistic Complex Event Recognition: A Survey. *ACM Computing Surveys (CSUR)*, 50(5):71, 2017. DOI: 10.1145/3117809.

[Alt92] Naomi S. Altman. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician*, 46(3):175–185, 1992. DOI: 10.1080/00031305.1992.10475879.

[Ame20] Louis Ameline. tooltipster-follower. https://github.com/louisameline/tooltipster-follower, 2020. (Last accessed: 05.02.2020).

[Ame19] Hannes Ametsreiter. Smartphone-Markt: Konjunktur und Trends (bitkom talk). https://www.bitkom-research.de/de/system/files?file=document/Bitkom-Pressekonferenz Smartphone-Markt 20 02 2019 Präsentation_final.pdf, 2019. (Last accessed: 09.04.2020).

[Apa20] Apache HTTP Server Contributors. Apache HTTP Server. https://httpd.apache.org, 2020. (Last accessed: 11.03.2020).

[Apa19] Apache Wiki Contributors. Apache Kafka Clients. https://cwiki.apache.org/confluence/display/KAFKA/Clients, 2019. (Last accessed: 28.11.2019).

[App20] Apple Inc. Health. https://www.apple.com/de/ios/health/, 2020. (Last accessed: 15.04.2020).

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 03)*, pages 665–665, San Diego, CA, USA, 2003. DOI: `10.1145/872757.872854`.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006. DOI: `10.1007/s00778-004-0147-z`.

[AMU+17] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. Tutorial: Complex Event Recognition Languages. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS 17)*, pages 7–10. ACM, 2017. DOI: `10.1145/3093742.3095106`.

[BMT+13] Sobhan Badiozamany, Lars Melander, Thanh Truong, Xu Cheng, and Tore Risch. Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-defined Aggregate Functions. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 325–330, Arlington, TX, USA. ACM, 2013. DOI: `10.1145/2488222.2488284`.

[Bar20] Brian Barnett. graham_scan_js. https://github.com/brian3kb/graham_scan_js, 2020. (Last accessed: 05.02.2020).

[BCK+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008. ISBN: 9783540779735.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 9780387310732.

[BB12] Christian Bockermann and Hendrik Blom. The Streams Framework. Technical report, TU Dortmund, 2012. DOI: 10.17877/DE290R-19178.

[Boo20] Bootstrap Contributors. Bootstrap. https://getbootstrap.com/, 2020. (Last accessed: 05.02.2020).

[Bra86] Bart Braden. The Surveyor's Area Formula. *The College Mathematics Journal*, 17(4):326–337, 1986. DOI: 10.1080/07468342.1986.11972974.

[BBC+15] Michael Brand, Tobias Brandt, Carsten Cordes, Marc Wilken, and Timo Michelsen. Herakles: A System for Sensor-Based Live Sport Analysis using Private Peer-to-Peer Networks. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*:71–80, 2015.

[Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001. DOI: 10.1023/A:1010933404324.

[Bre08] Gert Brettlecker. *Efficient and Reliable Data Stream Management*. PhD thesis, University of Basel, 2008. DOI: 10.5451/unibas-004538352.

[Bri20] Brightcove, Inc. Video.js. https://videojs.com, 2020. (Last accessed: 05.02.2020).

[Bri16] Frederik Brix. *Complex Event Detection in Real Time DataStreams*. Bachelor's thesis, University of Basel, 2016.

[BSO20] BSON Specification Contributors. BSON. http://bsonspec.org, 2020. (Last accessed: 10.02.2020).

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[Cat20a] Catapult. Clearsky T6. https://www.catapultsports.com/products/clearsky-t6, 2020. (Last accessed: 02.04.2020).

[Cat20b] Catapult. Optimeye S5. https://www.catapultsports.com/products/optimeye-s5, 2020. (Last accessed: 02.04.2020).

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 03)*, pages 668–668, San Diego, CA, USA, 2003. DOI: 10.1145/872757.872857.

[Cha20] Chart.js Contributors. Chart.js. https://www.chartjs.org/, 2020. (Last accessed: 05.02.2020).

[CFL+14] Sheng Chen, Zhongyuan Feng, Qingkai Lu, Behrooz Mahasseni, Trevor Fiez, Alan Fern, and Sinisa Todorovic. Play Type Recognition in Real-World Football Video. In *Proceedings of the 2014 IEEE Winter Conference on Applications of Computer Vision (WACV 14)*, pages 652–659, Steamboat Springs, CO, USA, 2014. DOI: 10.1109/WACV.2014.6836040.

[Chy20a] ChyronHego Corporation. Coach Capture. https://chyronhego.com/products/broadcast-graphics/coach-capture/, 2020. (Last accessed: 01.04.2020).

[Chy20b] ChyronHego Corporation. TRACAB GO. https://chyronhego.com/products/sports-tracking/zxy-go-wearable-tracking/, 2020. (Last accessed: 02.04.2020).

[Chy20c] ChyronHego Corporation. TRACAB Optical Tracking. https://chyronhego.com/products/sports-tracking/tracab-optical-tracking/, 2020. (Last accessed: 02.04.2020).

[Chy20d] ChyronHego Corporation. TRACAB RF. https://chyronhego.com/products/sports-tracking/zxy-arena-wearable-tracking/, 2020. (Last accessed: 02.04.2020).

[Cod70] Edgar Frank Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970. DOI: 10.1145/362384.362685.

[Com20] Commons Codec Contributors. Apache Commons Codec. https://commons.apache.org/proper/commons-codec, 2020. (Last accessed: 27.01.2020).

[CV95] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995. DOI: 10.1007/BF00994018.

[CM09] Gianpaolo Cugola and Alessandro Margara. Raced: an adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware (ARM 09)*, pages 1–6, Urbana Champaign, IL, USA, 2009. DOI: 10.1145/1658185.1658190.

[CM10] Gianpaolo Cugola and Alessandro Margara. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the 4th ACM International Conference on Distributed and Event-based Systems (DEBS 10)*, pages 50–61, Cambrdige, UK, 2010. DOI: 10.1145/1827418.1827427.

[CM12a] Gianpaolo Cugola and Alessandro Margara. Complex event processing with T-REX. *Journal of Systems and Software*, 85(8):1709–1728, 2012. DOI: 10.1016/j.jss.2012.03.056.

[CM12b] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: from data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):1–62, 2012. DOI: 10.1145/2187671.2187677.

[CM13] Gianpaolo Cugola and Alessandro Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013. DOI: 10.1007/s00607-012-0217-9.

[Dar20] Dartfish. myDartfish Live S. http://www.dartfish.com/live_S, 2020. (Last accessed: 01.04.2020).

[DZ83] John D Day and Hubert Zimmermann. The OSI Reference Model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983. DOI: 10.1109/PROC.1983.12775.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI 04)*, San Francisco, CA, USA, 2004.

[DGP$^+$07] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR 07)*, pages 412–422, Asilomar, CA, USA, 2007.

[DD09] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer, 2009. ISBN: 9783642002335.

[Ecl20] Eclipse Foundation. Eclipse Jetty. https://www.eclipse.org/jetty, 2020. (Last accessed: 27.01.2020).

[Eid08] John Eidson. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Standard 1588-2008, IEEE Standards Association, 2008. https://standards.ieee.org/standard/1588-2008.html.

[Esp20] EsperTech, Inc. Esper. http://www.espertech.com/esper/, 2020. (Last accessed: 31.03.2020).

[Fac20] Facebook, Inc. RocksDB. https://rocksdb.org, 2020. (Last accessed: 08.01.2020).

[Fit20] Fitbit, Inc. Fitbit. https://www.fitbit.com, 2020. (Last accessed: 15.04.2020).

[FR07] Michael Fleischman and Deb Roy. Unsupervised Content-Based Indexing of Sports Video. In *Proceedings of the International Workshop on Multimedia Information Retrieval (MIR 07)*, pages 87–94, Augsburg, Germany. ACM, 2007. DOI: 10.1145/1290082.1290097.

[Fli18] Flink Contributors. Apache Flink Documentation (version 1.5) - Event Time Overview. https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/event_time.html, 2018. (Last accessed: 27.07.2018).

[FZ98] Michael Franklin and Stan Zdonik. "Data in your face": Push Technology in Perspective. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD 98)*, pages 516–519, Seattle, WA, USA. ACM, 1998. DOI: 10.1145/276304.276360.

[Fri17] David Frische. Revolution im Fussball dank Schweizer Technik? *20 Minuten*, 2017. https://www.20min.ch/schweiz/basel/story/Revolution-im-Fussball-dank-Schweizer-Technik–26553547 (Last accessed: 17.04.2020).

[GKS+13] Avigdor Gal, Sarah Keren, Mor Sondak, Matthias Weidlich, Hendrik Blom, and Christian Bockermann. Grand Challenge: The TechniBall System. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 319–324, Arlington, TX, USA. ACM, 2013. DOI: 10.1145/2488222.2488282.

[GZC13] Francis Galiegue, Kris Zyp, and Gary Court. JSON Schema: core definitions and terminology. Internet-Draft draft-zyp-json-schema-04, IETF Secretariat, 2013. https://tools.ietf.org/html/draft-zyp-json-schema-04.

[Gha08] Sherif Ghali. *Introduction to Geometric Computing*. Springer, 2008. ISBN: 9781848001145.

[Goo19] Google LLC. Protocol Buffers. https://developers.google.com/protocol-buffers, 2019. (Last accessed: 03.09.2019).

[Goo20] Google LLC. Google Fit. https://www.google.com/fit, 2020. (Last accessed: 15.04.2020).

[Gra72] Ronald L. Graham. An Efficient Algorith[sic] for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972. DOI: 10.1016/0020-0190(72)90045-2.

[GFW+11] Thomas von der Grün, Norbert Franke, Daniel Wolf, Nicolas Witt, and Andreas Eidloth. A Real-Time Tracking System for Football Match and Training Analysis. In *Microelectronic Systems*, pages 199–212. Springer, 2011. DOI: 10.1007/978-3-642-23071-4_19.

[Güt94] Ralf Hartmut Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994. DOI: 10.1007/BF01231602.

[Guz19] Tomasz (tgrez) Guz. Answer to "Is Kafka timestamp order corresponding to the offset?" https://stackoverflow.com/a/55259573, 2019. (Last accessed: 21.01.2020).

[Had19] Hadoop Contributors. Apache Hadoop. https://hadoop.apache.org, 2019. (Last accessed: 09.12.2019).

[Har69] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[HSS+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014. DOI: 10.1145/2528412.

[Hud20] Hudl. Sportscode. https://www.hudl.com/products/sportscode, 2020. (Last accessed: 01.04.2020).

[HKJ+10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC 10)*, Boston, MA, USA, 2010.

[Inm20] Inmotio Object Tracking B.V. Football Performance Analysis. https://inmotio.eu/football-performance-analysis/, 2020. (Last accessed: 14.02.2020).

[ISO88] ISO 8601:1988. Data elements and interchange formats – Information interchange – Representation of dates and times. Standard, International Organization for Standardization, Geneva, Switzerland, 1988.

[JA20] Caleb Jacob and Louis Ameline. Tooltipster. http://iamceege.github.io/tooltipster/, 2020. (Last accessed: 05.02.2020).

[JMR+13] Hans-Arno Jacobsen, Kianoosh Mokhtarian, Tilmann Rabl, Mohammad Sadoghi, Reza Sherafat Kazemzadeh, Young Yoon, and Kaiwen Zhang. Grand Challenge: The Bluebay Soccer Monitoring Engine. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 295–300, Arlington, TX, USA. ACM, 2013. DOI: `10.1145/2488222.2488278`.

[JDN+13] Martin Jergler, Christoph Doblander, Mohammedreza Najafi, and Hans-Arno Jacobsen. Grand Challenge: Real-time Soccer Analytics Leveraging Low-Latency Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 307–312, Arlington, TX, USA. ACM, 2013. DOI: `10.1145/2488222.2488280`.

[Jos06] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, 2006, pages 1–18. DOI: `10.17487/RFC4648`.

[jQu20a] jQuery Foundation. jQuery. https://jquery.com/, 2020. (Last accessed: 05.02.2020).

[jQu20b] jQuery selectBox Contributors. jQuery selectBox. http://marcj.github.io/jquery-selectBox/, 2020. (Last accessed: 05.02.2020).

[Kaf16] Kafka Contributors. Apache Kafka Documentation (version 0.10.1). https://kafka.apache.org/0101/documentation.html, 2016. (Last accessed: 21.01.2020).

[KK15] Martin Kleppmann and Jay Kreps. Kafka, Samza and the Unix Philosophy of Distributed Data. *IEEE Data Engineering Bulletin*, 38(4):4–14, 2015.

[KNR11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB 11)*, Athens, Greece. ACM, 2011.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. DOI: 10.1145/357172.357176.

[Lan18] Alice Lanzke. Wird Fussball zum Computer-Spiel? *Spick*, 425:32–33, 2018.

[Lob17] Rufus Lobo. *Query-by-Sketch in Automatically Annotated Football Matches*. Master's thesis, University of Basel, 2017.

[MSZ13] Kasper Grud Skat Madsen, Li Su, and Yongluan Zhou. Grand Challenge: MapReduce-Style Processing of Fast Sensor Data. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 313–318, Arlington, TX, USA. ACM, 2013. DOI: 10.1145/2488222.2488281.

[MZS16] Kasper Grud Skat Madsen, Yongluan Zhou, and Li Su. Enorm: Efficient Window-Based Computation in Large-Scale Distributed Stream Processing Systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS 16)*, pages 37–48, Irvine, CA, USA. ACM, 2016. DOI: 10.1145/2933267.2933315.

[Mic14] Timo Michelsen. Data Stream Processing in Dynamic and Decentralized Peer-to-Peer Networks. In *Proceedings of the 2014 ACM SIGMOD PhD Symposium (SIGMOD 14 PhD Symposium)*, pages 1–5, Snowbird, UT, USA. ACM, 2014. DOI: 10.1145/2602622.2602629.

[MBC+15] Timo Michelsen, Michael Brand, Carsten Cordes, and Hans-Jürgen Appelrath. Demo: Herakles – Real-time Sport Analysis using a Distributed Data Stream Management System. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS 15)*, pages 356–359, Oslo, Norway. ACM, 2015. DOI: 10.1145/2675743.2776775.

[MMB+10] David Mills, Jim Martin, Jack Burbank, and William Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, RFC Editor, 2010, pages 1–110. DOI: 10.17487/RFC5905.

[Mon17a] MongoDB, Inc. MongoDB Documentation (version 3.6) - Create a 2d Index. https://docs.mongodb.com/v3.6/tutorial/build-a-2d-index/, 2017. (Last accessed: 12.02.2020).

[Mon17b] MongoDB, Inc. MongoDB Documentation (version 3.6) - Indexes. https://docs.mongodb.com/v3.6/indexes, 2017. (Last accessed: 10.02.2020).

[Mon17c] MongoDB, Inc. MongoDB Documentation (version 3.6) - Insert Documents. https://docs.mongodb.com/v3.6/tutorial/insert-documents/index.html, 2017. (Last accessed: 11.02.2020).

[Mon17d] MongoDB, Inc. MongoDB Documentation (version 3.6) - $jsonSchema. https://docs.mongodb.com/v3.6/reference/operator/query/jsonSchema, 2017. (Last accessed: 10.02.2020).

[Mon17e] MongoDB, Inc. MongoDB Documentation (version 3.6) - Query Documents. https://docs.mongodb.com/v3.6/tutorial/query-documents/index.html, 2017. (Last accessed: 29.03.2020).

[Mon20] MongoDB, Inc. MongoDB. https://www.mongodb.com, 2020. (Last accessed: 10.02.2020).

[Mov20] Movio Limited. Samza Prometheus Exporter. https://github.com/movio/samza-prometheus-exporter, 2020. (Last accessed: 25.03.2020).

[MZJ13] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. The DEBS 2013 Grand Challenge. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 289–294, Arlington, TX, USA. ACM, 2013. DOI: 10.1145/2488222.2488283.

[Nah20] Christoph Nahr. Tektosyne. https://github.com/kynosarges/tektosyne, 2020. (Last accessed: 03.02.2020).

[NBN20] NBN23. NBN23 Performance. https://www.nbn23.com/en/performance-tracking, 2020. (Last accessed: 02.04.2020).

[Net20] Netdata Contributors. Netdata. https://github.com/netdata/netdata, 2020. (Last accessed: 28.01.2020).

[NPP+17] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017. DOI: `10.14778/3137765.3137770`.

[Nut18] Daniel Nutz. Opta Match Experience: Wie werden Daten und Statistiken beim Fußball erfasst? https://www.goal.com/de/meldungen/opta-match-experience-wie-werden-daten-statistiken-fussball/1edqf26j61ajg1ni3kpe21r2mv, 2018. (Last accessed: 21.03.2020).

[Obr16] Marco Obradovic. *Highly-Parameterizable Data Generation and Modularized Real-Time Data Stream Analysis in Forest Fire Fighting Scenarios*. Master's thesis, University of Basel, 2016.

[Opa79] Jaroslav Opatrny. Total ordering problem. *SIAM Journal on Computing*, 8(1):111–114, 1979. DOI: `10.1137/0208008`.

[Opt18] Opta Sports. Opta's event definitions. https://www.optasports.com/news/opta-s-event-definitions/, 2018. (Last accessed: 20.03.2020).

[Opt20a] Opta Sports. Opta data feeds. https://www.optasports.com/services/data-feeds/, 2020. (Last accessed: 09.03.2020).

[Opt20b] Opta Sports. Unsere Feeds im Detail. https://www.optasports.com/de/produkte/feeds/unsere-feeds-im-detail/, 2020. (Last accessed: 09.03.2020).

[Pag19] Page, John. "Polygon" From Math Open Reference. https://www.mathopenref.com/polygon.html, 2019. (Last accessed: 28.06.2019).

[PGR+19] Eduard Pons, Tomás García-Calvo, Ricardo Resta, Hugo Blanco, Roberto López del Campo, Jesús Díaz García, and Juan José Pulido. A comparison of a GPS device and a multi-camera video technology during official soccer matches: Agreement between systems. *PLoS ONE*, 14(8), 2019. DOI: `10.1371/journal.pone.0220729`.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer, 1985. ISBN: 0387961313.

[Pro12] Lukas Probst. *Routing-Optimierung für verteilte Datenverwaltung und Prozessausführung auf Mobilgeräten mit OSIRIS-SR*. Bachelor's thesis, University of Basel, 2012.

[Pro14] Lukas Probst. *PAN – A P2P Approach for Scalable Complex Event Detection in Distributed Data Streams*. Master's thesis, University of Basel, 2014. https://dbis.dmi.unibas.ch/downloads/theses/MSc_Thesis_Lukas_Probst.pdf.

[PAL+18] Lukas Probst, Ihab Al Kabary, Rufus Lobo, Fabian Rauschenbach, Heiko Schuldt, Philipp Seidenschwarz, and Martin Rumo. SportSense: User Interface for Sketch-Based Spatio-Temporal Team Sports Video Scene Retrieval. In *Proceedings of the 1st Workshop on User Interface for Spatial and Temporal Data Analysis (UISTDA 18)*, Tokyo, Japan. CEUR-WS, 2018. http://ceur-ws.org/Vol-2068/uistda3.pdf.

[PBS+17] Lukas Probst, Frederik Brix, Heiko Schuldt, and Martin Rumo. Demo: Real-Time Football Analysis with StreamTeam. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS 17)*, pages 319–322. ACM, 2017. DOI: `10.1145/3093742.3095089`.

[PGS16a] Lukas Probst, Ivan Giangreco, and Heiko Schuldt. PAN – Distributed Real-Time Complex Event Detection in Multiple Data Streams. In *Proceedings of the 16th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 16)*, pages 189–195, Heraklion, Crete, Greece. Springer, 2016. DOI: `10.1007/978-3-319-39577-7_15`.

[PGS16b] Lukas Probst, Ivan Giangreco, and Heiko Schuldt. Pull-based Real-Time Complex Event Detection in Multiple Data Streams – the PAN Approach. Technical report, University of Basel, 2016.

https://courses.cs.unibas.ch/techreports/_Downloads/cs-2016-003.pdf.

[PRS+18] Lukas Probst, Fabian Rauschenbach, Heiko Schuldt, Philipp Seidenschwarz, and Martin Rumo. Integrated Real-Time Data Stream Analysis and Sketch-Based Video Retrieval in Team Sports. In *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data 18)*, pages 548–555. IEEE, 2018. DOI: 10.1109/BigData.2018.8622592.

[PSS13] Lukas Probst, Nenad Stojnić, and Heiko Schuldt. COMPASS – Latency Optimal Routing in Heterogeneous Chord-based P2P Systems. Technical report, University of Basel, 2013. https://courses.cs.unibas.ch/techreports/_Downloads/cs-2013-004.pdf.

[Pro20] Prometheus Contributors. Prometheus. https://prometheus.io, 2020. (Last accessed: 25.03.2020).

[Rau17] Fabian Rauschenbach. *SportSense Ice Hockey: Sketch-Based Search in Sport Videos*. Bachelor's thesis, University of Basel, 2017.

[Rau19] Fabian Rauschenbach. *A Zooming UI for Team Sport Analysis*. Master's project, University of Basel, 2019.

[RBR+16] Keven Richly, Max Bothe, Tobias Rohloff, and Christian Schwarz. Recognizing Compound Events in Spatio-Temporal Football Data. In *Proceedings of the International Conference on Internet of Things and Big Data (IoTBD 16)*, pages 27–35, Rome, Italy. SCITEPRESS, 2016. DOI: 10.5220/0005877600270035.

[RMS17] Keven Richly, Florian Moritz, and Christian Schwarz. Utilizing Artificial Neural Networks to Detect Compound Events in Spatio-Temporal Soccer Data. In *Proceedings of the 2017 SIGKDD Workshop MiLeTS (MiLeTS 17)*, Halifax, Canada, 2017.

[RZG+18] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. Probabilistic Management of Late Arrival of Events. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS 18)*, pages 52–63, Hamilton, New Zealand. ACM, 2018. DOI: 10.1145/3210284.3210293.

[RM19] Henriette Röger and Ruben Mayer. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Computing Surveys (CSUR)*, 52(3):36:1–36:37, 2019. DOI: 10.1145/3303849.

[Roh16] Till Rohrmann. Answer to "Apache Flink's ingestion time, which wall lock?" https://stackoverflow.com/a/35701510, 2016. (Last accessed: 22.11.2018).

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. DOI: 10.1038/323533a0.

[Rum20] Martin Rumo. Human vs Artificial Intelligence in Sports Coaching: The Case of Football. *Presentation at the Spring Workshop on Mining and Learning 2020 (SMiLe 20)*, 2020.

[Sam17a] Samza Contributors. Apache Samza Documentation (version 0.13) - API Overview. http://samza.apache.org/learn/documentation/0.13/api/overview.html, 2017. (Last accessed: 09.12.2019).

[Sam17b] Samza Contributors. Apache Samza Documentation (version 0.13) - Architecture. http://samza.apache.org/learn/documentation/0.13/introduction/architecture.html, 2017. (Last accessed: 09.12.2019).

[Sam17c] Samza Contributors. Apache Samza Documentation (version 0.13) - Checkpointing. http://samza.apache.org/learn/documentation/0.13/container/checkpointing.html, 2017. (Last accessed: 04.02.2020).

[Sam17d] Samza Contributors. Apache Samza Documentation (version 0.13) - Concepts. http://samza.apache.org/learn/documentation/0.13/introduction/concepts.html, 2017. (Last accessed: 26.03.2019).

[Sam17e] Samza Contributors. Apache Samza Documentation (version 0.13) - Configuration Reference. http://samza.apache.org/learn/documentation/0.13/jobs/configuration-table.html, 2017. (Last accessed: 23.03.2020).

[Sam17f] Samza Contributors. Apache Samza Documentation (version 0.13) - Event Loop. http://samza.apache.org/learn/documentation/0.13/container/event-loop.html, 2017. (Last accessed: 20.06.2019).

[Sam17g] Samza Contributors. Apache Samza Documentation (version 0.13) - Metrics Reference. http://samza.apache.org/learn/documentation/0.13/container/metrics-table.html, 2017. (Last accessed: 25.03.2020).

[Sam17h] Samza Contributors. Apache Samza Documentation (version 0.13) - Samza Container. http://samza.apache.org/learn/documentation/0.13/container/samza-container.html, 2017. (Last accessed: 20.06.2019).

[Sam17i] Samza Contributors. Apache Samza Documentation (version 0.13) - Serialization. http://samza.apache.org/learn/documentation/0.13/container/serialization.html, 2017. (Last accessed: 09.12.2019).

[Sam17j] Samza Contributors. Apache Samza Documentation (version 0.13) - State Management. http://samza.apache.org/learn/documentation/0.13/container/state-management.html, 2017. (Last accessed: 08.01.2020).

[Sam17k] Samza Contributors. Apache Samza Documentation (version 0.13) - Streams. http://samza.apache.org/learn/documentation/0.13/container/streams.html, 2017. (Last accessed: 04.06.2019).

[Sam17l] Samza Contributors. Apache Samza Documentation (version 0.13) - Windowing. http://samza.apache.org/learn/documentation/0.13/container/windowing.html, 2017. (Last accessed: 24.06.2019).

[Sam20] Samza Contributors. Apache Samza Documentation (version 1.4.0) - High Level Streams API. http://samza.apache.org/learn/documentation/1.4.0/api/high-level-api, 2020. (Last accessed: 03.04.2020).

[SMB+17] Adrià Arbués Sangüesa, Thomas B Moeslund, Chris H Bahnsen, and Raul Benıtez Iglesias. Identifying Basketball Plays from Sensor Data; towards a Low-Cost Automatic Extraction of Advanced Statistics. In *Proceedings of the 2017 IEEE International Conference on Data Mining Workshops (ICDMW 17)*, pages 894–901, New Orleans, LA, USA. IEEE, 2017. DOI: 10.1109/ICDMW.2017.123.

[SGR15] Stephan Schmid, Eszter Galicz, and Wolfgang Reinhardt. Performance investigation of selected SQL and NoSQL databases. In *Proceedings of the 18th AGILE International Conference on Geographic Information Science (AGILE 15)*, Lisbon, Portugal, 2015.

[Sch84] Fred Barry Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems (TOCS)*, 2(2):145–154, 1984. DOI: `10.1145/190.357399`.

[SJP+20] Philipp Seidenschwarz, Adalsteinn Jonsson, Michael Plüss, Martin Rumo, Lukas Probst, and Heiko Schuldt. The SportSense User Interface for Holistic Tactical Performance Analysis in Football. In *Proceedings of the 25th International Conference on Intelligent User Interfaces Companion (IUI 20 Companion)*, pages 45–46, Cagliari, Italy. ACM, 2020. DOI: `10.1145/3379336.3381473`.

[SJR+19] Philipp Seidenschwarz, Adalsteinn Jonsson, Fabian Rauschenbach, Martin Rumo, Lukas Probst, and Heiko Schuldt. Combining Qualitative and Quantitative Analysis in Football with SportSense. In *Proceedings of the 2nd International Workshop on Multimedia Content Analysis in Sports (MMSports 19)*, pages 34–41, Nice, France. ACM, 2019. DOI: `10.1145/3347318.3355519`.

[SRP+19] Philipp Seidenschwarz, Martin Rumo, Lukas Probst, and Heiko Schuldt. A Flexible Approach to Football Analytics: Assessment, Modeling and Implementation. In *Proceedings of the 12th International Symposium on Computer Science in Sport (IACSS 19)*, pages 19–27, Moscow, Russia. Springer, 2019. DOI: `10.1007/978-3-030-35048-2_3`.

[Shi07] Paul L. Shick. *Topology: Point-Set and Geometric*. John Wiley & Sons, 2007. ISBN: 9780470096055.

[Sta17] Christine Staehelin. Mit Software zum Champions-League-Sieg: Basler Forscher arbeiten an wegweisender Match-Analyse. http://barfi.ch/News-Basel/Mit-Software-zum-Champions-League-Sieg-Basler-Forscher-arbeiten-an-wegweisender-Match-Analyse, 2017. (Last accessed: 17.04.2020).

[Sta20a] Stats Perform. SportVU. https://www.statsperform.com/team-performance/football/optical-tracking/, 2020. (Last accessed: 02.04.2020).

[Sta20b] Stats Perform. Starts Perform GPS. https://www.statsperform.com/team-performance/football/athlete-monitoring/, 2020. (Last accessed: 02.04.2020).

[SJL+18] Manuel Stein, Halldor Janetzko, Andreas Lamprecht, Thorsten Breitkreutz, Philipp Zimmermann, Bastian Goldlücke, Tobias Schreck, Gennady Andrienko, Michael Grossniklaus, and Daniel A Keim. Bring it to the Pitch: Combining Video and Movement Data to Enhance Team Sport Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):13–22, 2018. DOI: 10.1109/TVCG.2017.2745181.

[SPF04] Andreas Stelzer, Klaus Pourvoyeur, and Alexander Fischer. Concept and Application of LPM – A Novel 3-D Local Position Measurement System. *IEEE Transactions on Microwave Theory and Techniques*, 52(12):2664–2669, 2004. DOI: 10.1109/TMTT.2004.838281.

[SPS13] Nenad Stojnić, Lukas Probst, and Heiko Schuldt. COMPASS – Optimized Routing for Efficient Data Access in Mobile Chord-Based P2P Systems. In *Proceedings of the 14th IEEE International Conference on Mobile Data Management (MDM 13)*, volume 1, pages 46–55, Milan, Italy. IEEE, 2013. DOI: 10.1109/MDM.2013.15.

[SGS14] Fabio Sulser, Ivan Giangreco, and Heiko Schuldt. Crowd-based Semantic Event Detection and Video Annotation for Sports Videos. In *Proceedings of the 2014 International ACM Workshop on Crowdsourcing for Multimedia (CrowdMM 14)*, pages 63–68, Orlando, FL, USA. ACM, 2014. DOI: 10.1145/2660114.2660119.

[Sza17] Janosh Szabo. Innovation aus einmaliger Position. *Bieler Tagblatt*, 2017. https://www.bielertagblatt.ch/nachrichten/biel/innovation-aus-einmaliger-position (Last accessed: 17.04.2020).

[TSM18] Quoc-Cuong To, Juan Soto, and Volker Markl. A survey of state management in big data processing systems. *The VLDB Journal*, 27(6):847–872, 2018. DOI: 10.1007/s00778-018-0514-9.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm @Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 14)*, pages 147–156. ACM, 2014. DOI: 10.1145/2588555.2595641.

[Tre20] Diego Tremper. rgb-color. https://github.com/diegotremper/rgb-color, 2020. (Last accessed: 05.02.2020).

[Twi20] Twitter, Inc. Twitter. https://twitter.com, 2020. (Last accessed: 18.03.2020).

[Uni17] University of Basel. StreamTeam: Taktikanalyse im Fussball live auf dem Tablet. https://www.unibas.ch/de/Aktuell/News/Uni-Research/StreamTeam–Taktikanalyse-im-Fussball-live-auf-dem-Tab let-.html, 2017. (Last accessed: 17.04.2020).

[VMD+13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing (SOCC 13)*, Santa Clara, CA, USA. ACM, 2013. DOI: 10.1145/2523616.2523633.

[VRA18] Soroush Vosoughi, Deb Roy, and Sinan Aral. The spread of true and false news online. *Science*, 359(6380):1146–1151, 2018. DOI: 10.1126/science.aap9559.

[Wal17] Samuel Waldis. Auch die Uni Basel will führend sein im Fussballgeschäft. *TagesWoche*, 17(51/52):32–33, 2017.

[WZ16] Kuan-Chieh Wang and Richard Zemel. Classifying NBA Offensive Plays Using Neural Networks. In *Proceedings of MIT Sloan Sports Analytics Conference*, Boston, MA, USA, 2016.

[Wie20] Patrick Wied. heatmap.js. https://www.patrick-wied.at/static/heatmapjs/, 2020. (Last accessed: 05.02.2020).

[Win17] Wintergreen Research. Sports Player Tracking and Analytics: Market Shares, Strategies, and Forecasts, Worldwide, 2017 to 2023. https://wintergreenresearch.com/sports-analytics, 2017. (Last accessed: 11.04.2020).

[Wir20] Daniel Wirtz. protobuf.js. https://github.com/protobufjs/proto buf.js, 2020. (Last accessed: 24.01.2020).

[WEG87] Svante Wold, Kim Esbensen, and Paul Geladi. Principal Component Analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(1-3):37–52, 1987. DOI: 10.1016/0169-7439(87)80084-9.

[WMT13] Yingjun Wu, David Maier, and Kian-Lee Tan. Grand Challenge: SPRINT Stream Processing Engine as a Solution. In *Proceedings of the 7th ACM International Conference on Distributed and Event-based Systems (DEBS 13)*, pages 301–306, Arlington, TX, USA. ACM, 2013. DOI: `10.1145/2488222.2488279`.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 12)*, Lombard, IL, USA, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP 13)*, pages 423–438, Famington, PA, USA. ACM, 2013. DOI: `10.1145/2517349.2522737`.

[ZR11] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[Zum19] Patrick Zumsteg. *Complex Event Detection and Event Based Retrieval in eSports*. Master's thesis, University of Basel, 2019.

Curriculum Vitae

Name	Lukas Probst
	Pestalozzistr. 43, 79540 Lörrach, Germany
Date of Birth	22.05.1990
Birthplace	Laufenburg, Switzerland
Citizenship	German

Education

since Aug. 2015	Ph. D. in Computer Science under the supervision of Prof. Dr. Heiko Schuldt, Databases and Information Systems research group, University of Basel, Switzerland
Sept. 2012 – Aug. 2014	M. Sc. in Computer Science, University of Basel, Switzerland
Sept. 2009 – Aug. 2012	B. Sc. in Computer Science, University of Basel, Switzerland

Employment

since Aug. 2015	Research and teaching assistant, Databases and Information Systems research group, University of Basel, Switzerland
Oct. 2014 – Jul. 2015	Research and teaching assistant, Distributed Systems research group, University of Stuttgart, Germany
Sept. 2012 – Dec. 2013	Student research assistant, Databases and Information Systems research group, University of Basel, Switzerland
Feb. 2011 – Dec. 2013	Student teaching assistant, Department of Mathematics and Computer Science, University of Basel, Switzerland

Publications

2020

– Philipp Seidenschwarz, Adalsteinn Jonsson, Michael Plüss, et al. The SportSense User Interface for Holistic Tactical Performance Analysis in Football. In *Proceedings of the 25th International Conference on Intelligent User Interfaces Companion (IUI 20 Companion)*, pages 45–46, Cagliari, Italy. ACM, 2020. DOI: 10.1145/3379336.3381473.

2019

– Philipp Seidenschwarz, Martin Rumo, Lukas Probst, et al. A Flexible Approach to Football Analytics: Assessment, Modeling and Implementation. In *Proceedings of the 12th International Symposium on Computer Science in Sport (IACSS 19)*, pages 19–27, Moscow, Russia. Springer, 2019. DOI: 10.1007/978-3-030-35048-2_3.

– Philipp Seidenschwarz, Adalsteinn Jonsson, Fabian Rauschenbach, et al. Combining Qualitative and Quantitative Analysis in Football with SportSense. In *Proceedings of the 2nd International Workshop on Multimedia Content Analysis in Sports (MMSports 19)*, pages 34–41, Nice, France. ACM, 2019. DOI: 10.1145/3347318.3355519.

2018

– Lukas Probst, Fabian Rauschenbach, Heiko Schuldt, et al. Integrated Real-Time Data Stream Analysis and Sketch-Based Video Retrieval in Team Sports. In *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data 18)*, pages 548–555. IEEE, 2018. DOI: 10.1109/BigData.2018.8622592.

– Lukas Probst, Ihab Al Kabary, Rufus Lobo, et al. SportSense: User Interface for Sketch-Based Spatio-Temporal Team Sports Video Scene Retrieval. In *Proceedings of the 1st Workshop on User Interface for Spatial and Temporal Data Analysis (UISTDA 18)*, Tokyo, Japan. CEUR-WS, 2018. http://ceur-ws.org/Vol-2068/uistda3.pdf.

2017

- Lukas Probst, Frederik Brix, Heiko Schuldt, et al. Demo: Real-Time Football Analysis with StreamTeam. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS 17)*, pages 319–322. ACM, 2017. DOI: `10.1145/3093742.3095089`.

2016

- Lukas Probst, Ivan Giangreco, and Heiko Schuldt. Pull-based Real-Time Complex Event Detection in Multiple Data Streams – the PAN Approach. Technical report, University of Basel, 2016. https://courses.cs.unibas.ch/techreports/_Downloads/cs-2016-003.pdf.
- Lukas Probst, Ivan Giangreco, and Heiko Schuldt. PAN – Distributed Real-Time Complex Event Detection in Multiple Data Streams. In *Proceedings of the 16th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 16)*, pages 189–195, Heraklion, Crete, Greece. Springer, 2016. DOI: `10.1007/978-3-319-39577-7_15`.

2014

- Lukas Probst. *PAN – A P2P Approach for Scalable Complex Event Detection in Distributed Data Streams*. Master's thesis, University of Basel, 2014. https://dbis.dmi.unibas.ch/downloads/theses/MSc_Thesis_Lukas_Probst.pdf.

2013

- Lukas Probst, Nenad Stojnić, and Heiko Schuldt. COMPASS – Latency Optimal Routing in Heterogeneous Chord-based P2P Systems. Technical report, University of Basel, 2013. https://courses.cs.unibas.ch/techreports/_Downloads/cs-2013-004.pdf.
- Nenad Stojnić, Lukas Probst, and Heiko Schuldt. COMPASS – Optimized Routing for Efficient Data Access in Mobile Chord-Based P2P Systems. In *Proceedings of the 14th IEEE International Conference on Mobile Data Management (MDM 13)*, volume 1, pages 46–55, Milan, Italy. IEEE, 2013. DOI: `10.1109/MDM.2013.15`.

2012

– Lukas Probst. *Routing-Optimierung für verteilte Datenverwaltung und Prozessausführung auf Mobilgeräten mit OSIRIS-SR*. Bachelor's thesis, University of Basel, 2012.

Advised Theses

– Patrick Zumsteg. *Complex Event Detection and Event Based Retrieval in eSports*. Master's thesis, University of Basel, 2019.

– Fabian Rauschenbach. *A Zooming UI for Team Sport Analysis*. Master's project, University of Basel, 2019.

– Fabian Rauschenbach. *SportSense Ice Hockey: Sketch-Based Search in Sport Videos*. Bachelor's thesis, University of Basel, 2017.

– Rufus Lobo. *Query-by-Sketch in Automatically Annotated Football Matches*. Master's thesis, University of Basel, 2017.

– Marco Obradovic. *Highly-Parameterizable Data Generation and Modularized Real-Time Data Stream Analysis in Forest Fire Fighting Scenarios*. Master's thesis, University of Basel, 2016.

– Frederik Brix. *Complex Event Detection in Real Time DataStreams*. Bachelor's thesis, University of Basel, 2016.