January 2015

# Efficient Aggregated Deliveries with Strong Guarantees in an Event-based Distributed System

Gregory Aaron Wilkin
*Purdue University*

**PURDUE UNIVERSITY**
**GRADUATE SCHOOL**
**Thesis/Dissertation Acceptance**

This is to certify that the thesis/dissertation prepared

By Gregory Aaron Wilkin

Entitled
Efficient Aggregated Deliveries with Strong Guarantees in an Event-based Distributed System

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Patrick T. Eugster
Chair

Sonia Fahmy

Xiangyu Zhang

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Patrick T. Eugster

Approved by: William Gorman — 10/15/2015

Head of the Departmental Graduate Program — Date

# EFFICIENT AGGREGATED DELIVERIES WITH STRONG GUARANTEES IN EVENT-BASED DISTRIBUTED SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Gregory Aaron Wilkin

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2015

Purdue University

West Lafayette, Indiana

# TABLE OF CONTENTS

Page



[1] Published in JPDC 2012
Authors: G. A. Wilkin and P. Eugster

[2]Published in MIDDLEWARE 2011
Authors: G. A. Wilkin, K. R. Jayaram, P. Eugster and A. Khetrapal

[3]Published in ToIT 2013
Authors: G. A. Wilkin, K. R. Jayaram and P. Eugster

Page

# LIST OF FIGURES

# LIST OF TABLES

## ABSTRACT

Wilkin, Gregory Aaron PhD., Purdue University, December 2015. Efficient Aggregated Deliveries with Strong Guarantees in Event-based Distributed Systems. Major Professor: Patrick Eugster.

A popular approach to designing large scale distributed systems is to follow an event-based approach. In an event-based approach, a set of software components interact by producing and consuming events. The event-based model allows for the decoupling of software components, allowing distributed systems to scale to a large number of components. Event correlation allows for higher order reasoning of events by constructing complex events from single, consumable events. In many cases, event correlation applications rely on centralized setups or broker overlay networks. In the case of centralized setups, the guarantees for complex event delivery are stronger, however, centralized setups create performance bottlenecks and single points of failure. With broker overlays, the performance and fault tolerance are improved but at the cost of weaker guarantees.

The goal of this dissertation is to develop an efficient middleware for event correlation while still providing strong guarantees. First, we show what is necessary for strong guarantees in asynchronous distributed event-based systems that perform event correlation. Secondly, we provide the main deliverable of this dissertation: a generic middleware system, FAIDECS, which utilizes event types to efficiently correlate individually multicast events while providing strong guarantees for asynchronous event-based distributed systems. We then provide semantic alternatives to those provided in FAIDECS, showing what strong guarantees are able to be provided given certain operators.

# 1 INTRODUCTION

An event-based system consists of a set of software components that interact using event notifications. In this context, an event is any happening of interest, which is typically a change in state of some component within a system. Examples of events include mouse clicks, keyboard events, timers, OS and I/O interrupts, sensor readings, stock quotes and news articles. Events contain data attributes, where a typed event consists in an ordered set of attributes, each of which may be a simple or complex type respectively. For example, a weather sensor reading might contain attributes such as location data (may be represented by several individual attributes or a single complex attribute, which contains individual elements), temperature readings (a simple floating point value), barometer readings, etc., and may contain more complex data such as satellite readings. The event handler, also called a reaction, is executed when an event occurs, often a method call, and is executed asynchronously to the caller. An increasing number of applications detect and react to patterns of events, called complex events or composite events, and event correlation is the detection of complex events.

Many software applications utilize the event-based programming paradigm. Example applications include operating systems, graphical interfaces, news dissemination, algorithmic stock/commodity training, weather prediction and detection, network management, intrusion detection, etc. Many mainstream programming languages support simple event handling of singleton events. Examples include Java's JFC/Swing, RTSJ's AsynchEvents and C's POSIX condition variables.

Due to newer concepts such as pervasive computing, or the increasing connectivity of several components (complex or simple) and the overall size of distributed systems in general, a number of requirements emerge, the most basic being the availability of scalable interaction mechanisms which are crucial for building and maintaining a broad range of event based systems. Many such systems not only must support increasingly larger numbers of components, ranging from hundreds to thousands, but also face complex application environments which require strong guarantees across these components, or even subsets of components.

Further, automation of data processing is becoming more of the norm rather than the traditional interactive user request/reply architecture. This gives rise to data-/information-driven distributed applications which react to data according to a set of predefined, programmer specified rules. The stock market is nearly revolutionized by automated trades by detecting events and trends in the market and reacting at the nano-second level. Large office buildings on or near Wall Street, once full of humans trading equities, are now being emptied and filled with large, powerful machines to perform trades due to the close proximity to the source tickers where every nano-second counts. Inventory in stores may now be efficiently monitored, and low supplies quickly and automatically trigger orders for replenishment. For any such computation to be automated, components must be provided with the necessary data to constantly check for such conditions, where these conditions often take the form of events.

## 1.1 Request-reply Interaction

Traditional distributed applications have worked under the assumption that data and/or services are stored in a collection of objects or databases, and retrieval of this data is through a request-reply interaction. Client-server architectures have risen as a result where process roles are clearly defined, and a system component actively retrieves data for processing. An example of this architecture is the Remote Proceduer Call (RPC), where similar techniques exist. These techniques have been optimized through a successful history of engineering experience where the principles are well understood. Many applications as a result naturally fit into this paradigm, making the request-reply interaction model a very suitable choice.

However, for many distributed applications, where the environment is far more dynamic and data must be communicated as events occur rather than in a predefined interval, the request-reply interaction model has great limitations. Often, in this model, communication is synchronous, and only between any two machines at a time which enforces a tight coupling of components and greatly hampers system scalability. Clients periodically poll remote data sources, and must balance the tradeoffs of resources such as network bandwidth, processing power, etc., for accuracy and timely relevance of data. Polling for data by the clients too often results in better accuracy of data, but wastes many resources. On the other hand, not polling often enough results in stale or irrelevant data, and may result in higher update latencies. Further, as the needs evolve from such applications, software extensibility is greatly restricted in the request-reply model. Control flow is encoded in application components, which couples the system configuration with the application logic of various components.

In contrast to the traditional request-reply model of interaction in distributed systems, event-based design provides a better alternative for many applications. This is largely due to the fact that event-based design inherently decouples system components, improving extensibility and scalability of the software, particularly for large scale distributed systems. This may often improve the reasoning and even design process of such systems since individual components may be designed independently of others.

## 1.2 Event-based Distributed Systems

In event-based distributed systems, components communicate through event notifications which are produced, transmitted and received by interested components. Middleware systems allow for the sources (event producers/publishers) to be decoupled from the sinks (event consumers/subscribers) since the middleware handles the transfer of events from the former to the latter. Sources and sinks need not even have a priori knowledge of each other. Decoupling avoids name binding of components, which yeilds modules that are largely independent; thus, new components may be deployed into a running application without the need for system reconfiguration. Communication using this specialized middleware is greatly improved in terms of efficiency and scalability by aggregating and sharing traffic among components when possible.

The event-based model also allows for more straightforward higher order reasoning of data within an application. As event notifications convey a particular happening of interest, multiple event notifications, when taken together as a single entity, may convey a more meaningful occurence of interest. Often, when events occur in a particular order, if certain events occur within a certain time relative to other events, or even if a number of different events ever occur at some point in a system, more may be inferred than by simply viewing the individual events. For example, monitoring weather systems requires the constant monitoring of individual weather events. However, in order to determine certain future events, e.g., a storm, multiple events must be correlated with other events in time and space to determine if the conditions are conducive for bad weather. Such systems demand event-based programming models since engineers may take advantage of their autonomic, reactive and asynchronous nature.

## 1.3 Engineering of Event-based Distributed Software

Consider an event-based distributed application such as an algorithmic stock trading application. For such an application, the interacting components are typically applications at the stock exchanges, commodity exchanges, brokerage firms and high frequency traders. Events which are produced and consumed in such a system include stock quotes, commodity price quotes, analyst reports, trading volumes, annual reports quarterly earnings statements, etc. Using a traditional request-reply design for a high frequency trading component at a brokerage to periodically poll the stock exchange component for stock quotes severely hampers scalability and ability to react to changing events as quickly as they occur. Thus, such a system is greatly improved by the use of a middleware layer which allows producers to offload events immediately as they occur, and then consumers may receive these events through the middleware using a push protocol. Thus, this middleware layer decouples producers and consumers and allows for much greater scalability. Thus, most systems consist of the following:

1. Producers and consumers, where the application/business logic (e.g., decisions regarding buying or selling stocks, posting a weather warning, etc.) is typically programmed in a language such as C, C++, Java, ML, etc.

2. Event transmission middleware, which are typically dedicated high bandwidth communication links or publish/subscribe systems (topic-based, type-based or content-based), achieving a form of multicast.

3. Event correlation middleware responsible for detecting complex events by correlating or matching singleton events.

Both 2 and 3 may be combined in the same middleware. Systems like Gryphon [85], PADRES [60] and Hermes [66] are such examples.

### 1.3.1 Engineering of Event-based Middleware

In these scenarios, more than one process is receiving and possibly aggregating messages at a time, implying that the middleware implement some form of multicast. Currently, most middleware which offers complex event detection either utilize centralized setups, which hampers performance and yields a single point of failure, or focus more on efficiency and complexity of matching or on the number of possible aggregations and thus yield only best-effort guarantees.

Thus, middleware that merges both 2 and 3 above while providing strong guarantees is highly desirable. Such middleware would then be responsible for the transmission/dissemination of both singleton events, when desired by a consumer, as well as the dissemination and correlation of complex events when also desired by any number of consumers.

**Problem Statement** Many complex event processing applications require strong guarantees. However, no such system exists which provides these strong guarantees in an efficient, fault tolerant, fully distributed manner. Current trends are thus to trade efficiency to provide these guarantees in a centralized setup or sacrifice strong guarantees to gain efficiency in a distributed setting.

## 1.4 Thesis Statement

*We propose an efficient model for distributed complex event processing which provides strong guarantees in the face of process failures. We first prove what is necessary to achieve these strong guarantees in a fault tolerant manner, proving certain fundamental impossibilities. We then propose a model which allows for an efficient implementation by introducing the proper determinism necessary to achieve stronger guarantees. We then provide a practical, concrete system application of that model, we call FAIDECS. Lastly, we investigate the relationships between our model/system and more expressive languages, outlining the trade-offs of the expressiveness of various operations.*

## 1.5 Contributions

The contributions of this thesis are as follows: First, theoretical examination of strong guarantees for complex event processing in a distributed setting will be provided, where we prove what is necessary to meet these strong guarantees. Next, this thesis will present FAIDECS, which is a decentralized event correlation middleware that provides strong guarantees efficiently by utilizing event types to provide a total order on subsets of event types of a system. Lastly, we explore (in)feasibilities of additional strong guarantees and how the addition of certain operators to FAIDECS would affect any such guarantees by exploring operators from other event-based systems. The technical and theoretical contributions of this thesis are thus:

1. We prove that in order to achieve agreement on correlated events among processes with exact, or similar, interests, a total order on individually multicast events is re-

quired. We enumerate a number of strong guarantees, providing (in)feasibilities of each.

2. FAIDECS, a "Fair Decentralized Event Correlation System" which achieves the above mentioned strong guarantees by providing a total order on subsets of events of interest utilizing event types. We prove that FAIDECS meets each of the presented strong guarantees shown to be feasible and empirically evaluate its performance against other systems which can provide the same guarantees.

3. We explore further guarantees, mixing different matching semantics and event disposal semantics, proving which may be met in FAIDECS and which may not. We contrast these guarantees with other systems providing trade-offs demonstrating the scalability of our decentralized algorithms and exploring overall performance benefits and tradeoffs by comparing two different Java implementations of FAIDECS with three different implementations of a global total order of which two are fault tolerant.

## 1.6 Roadmap

Chapter 2 presents proofs that a total order on individually multicast events is necessary to achieve even simple forms of agreement in the context of correlation. Chapter 3 presents FAIDECS, a fair decentralized event correlation system, which is the deliverable and implementation of the concepts presented in Chapter 2. And Chapter 4 presents alternative semantics to FAIDECS and (in)feasibilities of meeting each of the guarantees with these new semantics with empirical evaluation/comparison of FAIDECS to other systems.

## 2 MULTICASTING IN THE PRESENCE OF AGGREGATED DELIVERIES [1]

Several fundamental models of distributed systems leverage relationships among many events [57], and an increasingly large number of distributed applications are explicitly built on a form of message *correlation*. A traditional use of correlation consists in the verification of safety conditions for intrusion detection [56]. Network monitoring, more generally [54], also enables the improvement of resource usage, e.g., in data centers. Workflow monitoring and production chain management are further application scenarios [26, 60]. More recent application environments for correlation include embedded and pervasive systems [38], and sensor networks [72].

The semantics of correlation in decentralized asynchronous systems prone to failures remain, however, under-addressed. Seminal investigations of message correlation were conducted in the context of *active databases* [17, 34, 35]. These attempted to formalize different options in syntax and semantics of elementary correlation. A message $m_l^k$ in the following represents a message of *type* $T_k$. A sequence of messages $m_1^1 \cdot m_2^1 \cdot m_1^2$ can be matched by a "subscription" correlating instances of message types $T_1$ and $T_2$ as $[m_1^1, m_1^2]$ (*first received* first) or as $[m_2^1, m_1^2]$ (*most recent* first). However, such work, just like *stream processing* [8, 24], considers events to be *uni*cast, or focuses on individual processes, centralized setups, or synchronous systems. The more recent StreamCloud [40] strives for ordering *across* nodes, but this ordering is, however, achieved based on timestamps assuming synchronized clocks.

Message aggregation has also been investigated in the context of *content-based publish/subscribe* systems [15], which focus on *multi*cast. Several systems have been extended to support some form of correlation, broadening their scope from, say, the canonical stock quote dissemination example for publish/subscribe systems to expressive algorithmic stock

---

[1] PUBLISHED IN JPDC 2012
AUTHORS: G. A. WILKIN AND P. EUGSTER

trading. Examples of such systems are Gryphon [85], PADRES [60] and Hermes [66]. However, most such extensions focus on efficiency and complexity of matching or on the number of possible aggregations and thus yield only best-effort guarantees on message delivery unless relying on centralized rendezvous nodes [66].

In summary, the above-mentioned approaches exhibit the following limitations: (1) no guarantees on messages delivered or (2) no support for multicast and thus no guarantees *across* individual processes; (3) no consideration of failures or (4) use of specific architectural setups with centralized components assumed to be reliable.

The absence of guarantees or the violation of expectations due to failures can have drastic effects [74]. Consider, for example, monitoring a network to decide which one of two gateways to route certain traffic through. If the first gateway receives the sequence $m_1^1 \cdot m_2^1 \cdot m_1^2$ outlined above, but the second one receives the sequence $m_1^2 \cdot m_2^1 \cdot m_1^1$ instead, each gateway might consider itself to be responsible for routing. Worse even, each can consider the other to be responsible. Of course, individual systems can be designed to deal with some of these issues (e.g., by using a proxy process to merge and multiplex streams to replicas), but corresponding solutions are hardly generic and can easily introduce bottlenecks to performance and dependability.

While several kinds of properties have been proposed and rigorously investigated for single message delivery scenarios (e.g., agreed delivery [42], probabilistic delivery [13], ordering properties [33]), the *feasibility* of guarantees in the presence of atomic, *aggregated* deliveries of *sets* of messages which are *multicast* in asynchronous systems remains unexplored. This paper thus makes the following contributions:

- A simple model of multicast with aggregated message delivery and properties are proposed for the crash-stop failure model. The model includes a basic predicate grammar for subscriptions of processes supporting message correlation. We term this specification Conjunction Multi-Delivery Multicast (C-MDMcast).

- We show that to achieve agreement on delivered messages (message aggregates) among processes subscribed with identical conjunctions, total order on individual messages, or an equivalent oracle, is both useful (as conveyed by the example above) and *necessary*. We show this by exhibiting an algorithm *FRIP* implementing C-MDMcast on top of Total Order Broadcast (TOBcast) and vice-versa with a majority of correct processes. This is opposed to single message deliveries where (total) order and agreement can be separated.[2]

- We specify a stronger agreement property on conjunctions, which formalizes the intuition that the aggregated messages delivered in response to a first subscription, which "covers" a second subscription, should include the set of messages delivered to the latter one. Such *subsumption* is trivial in single-message deliveries (and in fact is paramount to scalability in publish/subscribe systems [4]) but more involving when the delivery of a message depends on others, and does thus not simply boil down to predicate inclusion. We prove that FRIP implements this stronger agreement property.

- We add *dis*junctions and introduce corresponding properties, defining the problem of Disjunction Multi-Delivery Multicast (D-MDMcast). We exhibit a derivation D-FRIP of our algorithm FRIP, which implements D-MDMcast.

- We formulate total order properties for conjunctions and disjunctions: we can leverage the total order required on *individual* messages to achieve agreement on aggregated deliveries in order to establish a total order on *aggregated* deliveries.

- We similarly propose ordering properties capturing the order in which messages are produced (FIFO order) and capturing causal dependencies (causal order).

Note that the goal of this paper is not to exhibit the *weakest failure detector* [19] for correlation or to propose efficient algorithms. The intent is to show that some total ordering (or equivalent oracle) is required to achieve agreement (not ordering) on aggregated deliveries, suggesting that system builders think of such order at the core of systems and not simply by layering it atop. Thus, our algorithm implementing C-MDMcast with TOBcast,

[2]Many agreement and (total) order properties are unnecessarily intertwined with liveness [10].

for instance, is inefficient. More specialized algorithms achieving the same guarantees efficiently with pragmatic fault tolerance assumptions without going through TOBcast are the topic of a companion paper [82]. Recent work by others [84], also motivated by solving agreed correlation, actually proposes a more generic yet inefficient TOBcast primitive for publish/subscribe systems. Note also that total order on messages is not a panacea: we describe feasible properties as well as infeasible ones for our algorithms.

In contrast to our initial work [80] this paper presents (a) proofs for relationships between primitives, (b) a more expressive subscription grammar with content-based predicates and (c) more general corresponding properties, (d) a stronger agreement property for subsumption relationships on subscriptions, and (e) ordering guarantees.

**Roadmap.** Section 2.1 presents background information. Section 2.2 then introduces C-MDMcast. Section 2.3 investigates relationships between TOBcast and C-MDMcast. Section 2.4 discusses coverage. Section 2.5 introduces disjunctions. Section 2.6 discusses total order. FIFO and causal order are addressed in Section 2.7. Section 2.8 presents related work. Section 2.9 concludes with final remarks.

## 2.1 Preliminaries

### 2.1.1 System Model

We assume a system $\Pi$ of *processes*, $\Pi$=$\{p_1$, ..., $p_u\}$, interconnected pairwise by reliable channels [11] with primitives to SEND messages and receive (RECV) these messages. We consider a crash-stop failure model [29], i.e., a faulty process may stop prematurely and does not recover. Further, we assume the existence of a discrete global clock to which processes do not have access and that an algorithm run $R$ consists in a sequence of events on processes. That is, similar to other models [5], one process performs an action per clock tick which is either of (a) a protocol action (e.g., RECV), (b) an internal action, or (c) a "no-op". In the following sections, we may augment this model at times with certain primitives (e.g., TOBcast, see below) for comparison.

A failure pattern $F$ is a function mapping clock times to processes, where $F(t)$ yields all the processes that crashed by time $t$. Let $crashed(F)$ be the set of all processes $\in \Pi$ that have crashed during $R$. Thus, for a correct process $p_i$, $p_i \in correct(F)$ where $correct(F) = \Pi - crashed(F)$ [20].

### 2.1.2 Properties and Total Order Broadcast

For brevity and clarity, we adopt in the following a more formal notation for properties than common. Consider for instance the well-known problem of Total Order Broadcast (TOBcast) [42] defined over primitives $\text{TO-BCAST}(m)$ and $\text{TO-DLVR}(m)$, which will be used for comparison later on. We denote $\text{TO-DLVR}^i(m)_t$ as the TO-delivery of message $m$ by process $p_i$ at time $t$, and similarly, $\text{TO-BCAST}^i(m)_t$ denotes the TO-broadcasting of $m$ by $p_i$ at time $t$. We elide any of $i, t$, or $m$ when not germane to the context. We write $\exists e$ for an event $e$ such as a SEND or TO-BCAST as a shorthand for $\exists e \in R$. The specification of Uniform TOBcast thus becomes (where SDM stands for *Single-Delivery Multicast*):

<u>SDM No Duplication</u> $\exists \text{TO-DLVR}^i(m)_t \Rightarrow \nexists \text{TO-DLVR}^i(m)_{t'} \mid t' \neq t$

<u>SDM No Creation</u> $\exists \text{TO-DLVR}(m)_t \Rightarrow \exists \text{TO-BCAST}(m)_{t'} \mid t' < t$

<u>SDM Validity</u> $\exists$TO-BCAST$^i(m) \ \wedge \ p_i \in correct(F) \Rightarrow \exists$TO-DLVR$^i(m)$

<u>SDM Agreement</u> $\exists$TO-DLVR$^i(m) \ \Rightarrow \ \forall p_j \in correct(F) \backslash \{p_i\}$ $\exists$TO-DLVR$^j(m)$

<u>SDM Total Order</u> $\exists$TO-DLVR$^i(m)_{t_i}$, TO-DLVR$^i(m')_{t'_i}$, TO-DLVR$^j(m)_{t_j}$, TO-DLVR$^j(m')_{t'_j}$ $\Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

SDM Agreement is visibly a *uniform* property. Property SDM Total Order corresponds to Strong Uniform Total Order (SUTO) in the categorization of Baldoni et al. [10].

## 2.2 Conjunction Multi-Delivery Multicast (C-MDMcast)

In this section, we present a specification of multicast with message conjunctions.

### 2.2.1 Predicate Grammar

Sets of delivered messages — *relations* — are messages aggregated according to specific subscriptions. Such subscriptions are combinations of predicates on messages expressed in disjunctive normal form (DNF) according to the following grammar:

*Subscription* $\Psi$ ::= $\Phi \mid \Phi \vee \Psi$ *Predicate* $\rho$ ::= $T[i].a\ op\ v \mid T[i].a\ op\ T[i].a \mid T[i] \mid \top$

*Conjunction* $\Phi$ ::= $\rho \mid \rho \wedge \Phi$ *Operation* *op* ::= $< \mid > \mid \leq \mid \geq \mid = \mid \neq$

A type $T$ is characterized by an ordered set of attributes $[a_1, ..., a_n]$ each of which has a type of its own – typically a scalar type such as `Integer` or `Float`. A message $m$ of type $T$ is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of $T$. $T[i].a$ denotes an attribute $a$ of the $i$-th *instance* of type $T$ ($T[i]$). $v$ is a value. As syntactic sugar, we can allow predicates to refer to just $T.a$, which can be automatically translated to $T[1].a$. We may use this in examples for simplicity.

A predicate that compares a single message attribute to a value or compares two message attributes on the *same* message, i.e., on the same instance of a same type (e.g., $T_k[i].a\ op\ T_k[i].a'$) is referred to as a *unary* predicate. When two distinct messages (two distinct types or different instances of the same type) are involved in a predicate, we speak of a *binary* predicate ($T_k[i].a\ op\ T_l[j].a',\ k \neq l \vee i \neq j$). To simplify properties, we also introduce the *empty* predicate $\top$ which trivially yields *true*. Predicates comparing an attribute of a type instance to itself ($T_k[i].a\ op\ T_k[i].a$) constitute useless operations and are prohibited. We also allow *wildcard* predicates of the form $T$ (or $T_1$) to be specified; such predicates simply specify a desired type $T$ of messages of interest. $T[i]$ implicitly also declares $T[k]\ \forall k \in [1..i-1]$ if these are not already explicitly declared as part of other predicates in the same subscription.

A process $p_j$'s subscription is referred to as $\Psi(p_j)$. By abuse of notation but unambiguously, we sometimes handle disjunctions or conjunctions as sets (of conjunctions and predicates respectively). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$, or $\Phi_r \in \Psi \Leftrightarrow \Psi = \Phi_1 \vee ... \vee \Phi_n$ with $r \in [1..n]$. For simplicity, we first consider a subscription to consist in a *single* conjunction in the context of C-MDMcast.

An example subscription $\Psi_S$ for an increase in three successive stock quotes after a quarterly earnings report in the above grammar is expressed as follows:

$$\begin{aligned}\Psi_S \;=\; & \texttt{StockQuote}[0].\texttt{time} > \texttt{EarningsReport}[0].\texttt{time} \wedge \\ & \texttt{StockQuote}[1].\texttt{value} > \texttt{StockQuote}[0].\texttt{value} \wedge \\ & \texttt{StockQuote}[2].\texttt{value} > \texttt{StockQuote}[1].\texttt{value}\end{aligned}$$

Our grammar is expressive enough to model concrete ones by capturing message streams (via windows $T[i]$), joining of multiple streams/sources (represented by different types $T_k$), and attribute-based filtering ($T.a$), without however introducing specialized syntax to support *several different* semantic choices for these (e.g., first received vs. most recent matching, tumbling windows vs. sliding windows).

### 2.2.2 Predicate Types and Evaluation

We assume a deterministic order $\prec_N$ within subscriptions based on the names of message types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on message types, and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly for presentation simplicity.

The number of messages involved in a subscription is given by the number of types and corresponding instances involved. More precisely, the types involved in a subscription are represented as *sequences*. As alluded to by the index $i$ in $T[i]$, a same type can be admitted multiple times. Such sequences can be viewed as predicate *signatures*:

$$\begin{array}{lcllcl}\mathbb{T}(\Phi \vee \Psi) & = & \mathbb{T}(\Phi) \uplus \mathbb{T}(\Psi) & \mathbb{T}(T[i].a \; op \; v) & = & \mathbb{T}(T[i]) \\ \mathbb{T}(\rho \wedge \Phi) & = & \mathbb{T}(\rho) \uplus \mathbb{T}(\Phi) & \mathbb{T}(\top) & = & \emptyset \\ \mathbb{T}(T_1[i].a_1 \; op \; T_2[j].a_2) & = & \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) & = & [\underbrace{T, ..., T}_{i\times}]\end{array}$$

$$\emptyset \uplus [T, ...] = [T, ...] \qquad\qquad [T, ...] \uplus \emptyset = [T, ...]$$

$$[\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] = \begin{cases} [\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec_N T_2 \\ [\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec_N T_1 \\ [\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2 \end{cases}$$

Above, $\oplus$ represents simple concatenation and $\uplus$ stands for in-order union of sequences. In the previous example, the types involved may thus be [`EarningsReport`, `StockQuote`, `StockQuote`, `StockQuote`].

Any subscription $\Phi$ thus involves a sequence of message types $\mathbb{T}(\Phi)$=$[T_1, ..., T_n]$ where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j]\ T_k = T_i = T_j$, that is, a subsequence of identical types. These represent a *stream* of messages of the respective type of length $j - i + 1$. A subscription is evaluated for an ordered set of messages $[m_1, ..., m_n]$, where $m_i$ is of type $T_i$. We assume that types of values in predicates are checked statically with respect to the types of messages. $T(m)$ returns the type of a given message $m$. Note that we do not introduce a set of uniquely identified types $\{T_1, T_2, ...\}$. This allows for the set of types to be unbounded which does not violate our assumptions or guarantees, and keeps notation more brief in that we can use [$T_1$, ..., $T_k$] to refer to a sequence of $k$ *arbitrary* types, as opposed to, e.g., [$T_{i_1}$, ..., $T_{i_k}$].

The evaluation of a conjunction $\Phi$ on a relation is written as $\Phi[m_1, ..., m_n]$. For evaluation of an attribute $a$ on a message $m_i$, we write $m_i.a$. Evaluation semantics for predicates are thus defined as follows:

$$(\Phi \vee \Psi)[m_1, ..., m_n] = \Phi[m_1, ..., m_n] \vee \Psi[m_1, ..., m_n] \qquad (T)[m_1, ..., m_n] = \textit{true}$$

$$(\rho \wedge \Phi)[m_1, ..., m_n] = \rho[m_1, ..., m_n] \wedge \Phi[m_1, ..., m_n] \qquad (\top)[m_1, ..., m_n] = \textit{true}$$

$$\begin{array}{l}(T[i].a\ op\ v)\\ \quad [m_1, ..., m_n]\end{array} = \begin{cases} m_{k+i-1}.a\ op\ v & T(m_k) = T\ \wedge\ (T(m_{k-1}) \neq T \\ & \quad \vee (k-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\begin{array}{l}(T_1[i].a_1\ op\ T_2[j].a_2)\\ \quad [m_1, ..., m_n]\end{array} = \begin{cases} m_{k+i-1}.a_1\ op\ m_{l+j-1}.a_2 & T(m_k) = T_1 \wedge \\ & (T(m_{k-1}) \neq T_1\ \vee (k-1) = 0) \\ & \wedge\ T(m_l) = T_2 \wedge \\ & (T(m_{l-1}) \neq T_2\ \vee (l-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

Parentheses are used for clarity. For brevity we write simply $\Phi[...]$ for $\Phi[...] = \textit{true}$. The DLVR primitive to be generically typed, i.e., for delivering a relation $[m_1, ..., m_n]$, we write $\text{DLVR}_\Phi([m_1, ..., m_n])$ where $m_i$ is of type $T_i$ such that $\mathbb{T}(\Phi)$=[$T_1, ..., T_n$]. Analogous to TOBcast, $\text{DLVR}^i_\Phi([..., m, ...])_t$ defines the delivery event of a message $m$ on process $p_i$ in response to $\Phi$ at time $t$ and $\text{MCAST}^i(m)_t$ defines the multicasting of a message $m$ by $p_i$ at time $t$. $i, t$ etc. may be omitted when not germane to the context.

### 2.2.3 Properties

Conjunction Multi-Delivery Multicast (C-MDMcast) is defined over primitives MCAST and DLVR, where DLVR is parameterized by a subscription $\Phi$ and delivers *ordered sets* of messages. In the remainder of this paper, *deliver* refers to DLVR (while *TO-deliver* refers to TO-DLVR), and *multicast* refers to MCAST (vs. *TO-broadcast*).

Basic Safety Properties

We define three basic safety properties for C-MDMcast:

MDM NO DUPLICATION $\exists$ DLVR$^i_\Phi([..., m, ...])_t \Rightarrow \nexists$DLVR$^i_\Phi([..., m, ...])_{t'} \mid t' \neq t$

MDM NO CREATION $\exists$DLVR$_\Phi([..., m, ...])_t \Rightarrow \exists$MCAST$(m)_{t'} \mid t' < t$

MDM ADMISSION $\exists$DLVR$^i_\Phi([m_1, ..., m_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow \Phi \in \Psi(p_i) \wedge$
$\Phi[m_1, ..., m_n] \wedge \forall k \in [1..n] : T(m_k) = T_k$

The MDM NO DUPLICATION property implies that a same message is delivered at most once on any single process for a conjunction, which may be opposed to certain systems that allow a same message to be correlated multiple times. Our property could be substituted to allow a delivery for *every instance* of a type in a conjunction which would, however, make the guarantees and proofs more complicated without affecting the feasibilities explored in this paper.

Liveness

MDM ADMISSION can trivially hold while not performing any deliveries. We have to be careful about providing strong delivery properties on *individually* multicast messages though, as messages may depend on others to match a given conjunction. We propose the two following complementary liveness properties:

MDM CONJUNCTION NON-TRIVIALITY $\exists$MCAST$(m^k_l), k \in [1..n], l \in [1..\infty] \wedge$
$p_i \in correct(F) \wedge \exists\Phi \in \Psi(p_i) \mid \Phi[m^1_l, ..., m^n_l] \Rightarrow \exists$DLVR$^i_\Phi([...])_{t_j} \mid j \in [1..\infty]$

MDM MESSAGE NON-TRIVIALITY $\exists$MCAST$^i(m^x)$, MCAST$^{k,l}(m^k_l), k \in [1..n]\backslash x$,
$l \in [1..\infty] \mid \{p_i, p_j, p_{k,l}\} \subseteq correct(F) \wedge \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \wedge \forall z \in [w..y]$,
$T_z = T(m^x) \wedge \nexists(T(m^x)[x - w + 1].a_1 \; op \; T[r].a_2) \in \Phi \mid (T \neq T(m^x) \vee r \neq x - w + 1) \wedge$
$\Phi[m^1_l, ..., m^{x-1}_l, m^x, m^{x+1}_l, ..., m^n_l] \Rightarrow \exists$DLVR$^j_\Phi([..., m^x, ...])$

These two properties deal with the two possible cases that can arise. The first property deals with dependencies across messages and can be paraphrased as follows: "If for a correct process $p_i$, there is an infinite number of relations of matching messages that are successfully multicast, then $p_i$ will deliver infinitely many such relations." This property is reminiscent of the FINITE LOSSES PROPERTY of fair-lossy channels [11]. It allows matching algorithms to discard *some* messages for practical purposes or for agreement and ordering, yet ensures that when matching messages are continuously multicast, a corresponding process will continuously deliver.

MDM MESSAGE NON-TRIVIALITY provides a property analogous to validity for single-message deliveries (e.g., TOBcast): If a message is multicast by a correct process $p_i$, and its delivery in response to a conjunction on some correct process $p_j$ is not conditioned by binary predicates with other message types, then the message must be delivered by $p_j$ if messages of all other types matching each other are continuously multicast. This latter condition is necessary because the delivery of the message even in the absence of binary predicates requires the *existence* of other messages.

The condition also ensures that any unary predicates on the respective message type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force a message to be delivered more than once as the position of the message is not fixed in the implied delivery. The example in Section 2.2.1 does not contain a unary predicate, and thus is not affected by this property. If the subscription $\Psi_S$ were extended to trigger only if the value of the U.S. dollar is below some value `v` as in $\Psi'_S$ = $\Psi_S \wedge$ `USDollar.value < v`, then any message matching this predicate will be delivered with the entire relation given by $\Psi_S$.

Note that none of these properties is impacted by the presence of multiple instances of a same type in a conjunction. An infinite flow of messages of some type implies multiple (a finite number of) infinite flows of that type.

Agreement

We now turn to a stronger property for relations delivered across processes:

MDM CONJUNCTION AGREEMENT $\exists \text{DLVR}^i_\Phi([m_1, ..., m_n]) \Rightarrow \forall p_j \in correct(F)\backslash\{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DLVR}^j_\Phi([m_1, ..., m_n])$

The uniform MDM CONJUNCTION AGREEMENT property ensures that two correct processes $p_i$ and $p_j$ with identical subscriptions expressed by the conjunction $\Phi$ must deliver the same relation, *without constraining the respective orders of such deliveries*.

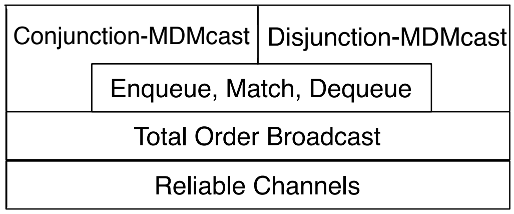



Figure 2.1.: Layered structure.

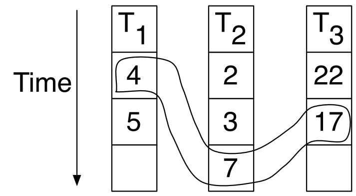



Figure 2.2.: Message queue match.

## 2.3 Comparison of C-MDMcast with Total Order Broadcast

In this section, we show that by augmenting our system model with the TOBcast primitive defined in Section 2.1.2, we can implement C-MDMcast and vice-versa with a majority of correct processes. This substantiates the intuition that a total order on messages or an equivalent oracle is not only useful to achieve agreement on conjoined messages, but also necessary.

### 2.3.1 C-MDMcast Using TOBcast

We present FRIP (*F*irst-*r*eceived matching with *i*nfix&*p*refix disposal), an algorithm implementing C-MDMcast using TOBcast. FRIP exploits the total order on messages created by TOBcast for agreement on relations. Fig. 2.1 represents a layered structure for MDMcast. D-MDMcast is an extension of C-MDMcast presented later.

#### Algorithm

Our FRIP algorithm (Alg. 2.1) can be broken down into several components namely (1) the buffering of TO-delivered messages (ENQUEUE), (2) the actual matching of messages (MATCH), and (3) the disposal of messages after matching (DEQUEUE). Every process $p_i$ has a subscription of one conjunction $\Phi$. A process $p_i$ maintains exactly one queue $Q$ per message type appearing in its conjunction (regardless of the number of instances of that

Executed by every process $p_i$

Initialisation:
  $\Psi \leftarrow \Phi$
  $\Phi \leftarrow \rho_1 \wedge ... \wedge \rho_m$ {*Composite,* $|\Phi| = m$}
  $Q[T] \leftarrow \emptyset$ {*Msg queues by type* $T$}
To MCAST$(m)$:
  TO-BCAST$(m)$
**procedure** DEQUEUE$([m_1, ..., m_l], Q)$ {*GC*}
  **for all** $Q[T(m_k)] = ... \oplus m_k \oplus m \oplus ...,$ $k \in [1..l]$ **do**
    $Q[T(m_k)] \leftarrow m \oplus ...$
**function** ENQUEUE $(m, \Phi, Q)$ {*Q mngmnt*}
  $win \leftarrow max(j \mid \exists(...T(m)[j].a...) \in \Phi)$
  **if** $\forall j = 1..win$ $(\exists(T(m)[j].a\ op\ v) \in \Phi \mid \neg(m.a\ op\ v) \vee \exists(T(m)[j].a\ op\ T(m)[j].a') \in \Phi \mid \neg(m.a\ op\ m.a'))$ **then**
    **return** false {*Useless bc unary preds*}
  **else**
    $Q[T(m)] \leftarrow Q[T(m)] \oplus m$
    **return** true
**upon** TO-DLVR$(m)$ **do**
  **if** $T(m) \in \mathbb{T}(\Phi)$ **then**
    **if** ENQUEUE$(m, \Phi, Q)$ **then**
      $[m_1, ..., m_l] \leftarrow$ MATCH$(\emptyset, \Phi, Q)$
      **if** $l > 0$ **then** {*Not an empty set*}
        DEQUEUE$([m_1, ..., m_l], Q)$
        DLVR$_\Phi([m_1, ..., m_l])$
**function** MATCH $([m'_1, ..., m'_n], \Phi, Q)$
  $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
  $l \leftarrow max(j \mid Q[T] = m_1 \oplus ... \oplus m_j \oplus ...) \mid m_j = m'_k$ {*Last*}
  **for all** $k = (l+1)..h \mid Q[T] = m_1 \oplus ... \oplus m_h$ **do**
    **if** $|\mathbb{T}(\Phi)| = n + 1$ **then**
      **if** $\Phi[m'_1, ..., m'_n, m_k]$ **then** {*match*}
        **return** $[m'_1, ..., m'_n, m_k]$
    **else if** $E =$ MATCH$([m'_1, ..., m'_n, m_k], \Phi, Q) \neq \emptyset$ **then**
      **return** $E$
  **return** $\emptyset$

Alg. 2.1.: First-Received matching with Infix&Prefix disposal (FRIP) algorithm.

type in its subscription). When TO-delivering a message, $p_i$ first checks whether the type of the message is in its subscription and, if so, attempts to ENQUEUE it. $Q[T(m)] \oplus m$ denotes appending a message $m$ to the queue of $m$'s type $T(m)$. The ENQUEUE primitive returns *true* if the message has been ENQUEUEd, indicating it satisfies all *unary* predicates on the respective type in the conjunction. This tells the algorithm to MATCH, as any received message can complete a relation.

It is important that this matching is triggered deterministically on every process and that the matching itself is deterministic. The procedure attempts to find the *first instance* of the *first type* in $\Phi$ for which there are messages of the remaining types with which all predicates in $\Phi$ are satisfied. Among all such possibilities, if any, the algorithm recursively seeks for a match with the *first instance* of the *second type* in $\Phi$, etc., until a match is found or no more possibilities exist. In the case of messages of a same type, a first instance of that type is recursively matched with the *first follow-up instance* of the *same type* until the number of messages needed for that type are matched, or until all possibilities in the queue for that type are exhausted. Thus in Alg. 2.1, on Line 11, $l$ denotes the last matched instance of the currently matched type. These semantics can be termed *first-received* matching semantics. Consider the example of Section 2 where messages of two types $T_1$ and $T_2$ are matched with wildcard predicates of the respective types. A message $m_l^k$ in the following represents a message of type $T_k$. A sequence $m_1^1 \cdot m_2^1 \cdot m_1^2$ received by a process $p_i$ will lead to the match $[m_1^1, m_1^2]$ with the above matching semantics, while a permutation of the sequence, $m_2^1 \cdot m_1^1 \cdot m_1^2$ will lead to the match $[m_2^1, m_1^2]$. A simple permutation across processes can thus lead to delivery of distinct relations, which intuitively conveys the need for total order.

The described matching algorithm performs an exhaustive search and is thus not efficient; however, it suffices to illustrate the relevant properties and can be represented concisely. More elaborate and efficient matching algorithms exist, offering the same semantics. A common approach consists in storing *partial matches* in specialized data-structures for matching a given message effectively (e.g., [50]). The goal of this paper is not to give guidelines on *how* exactly correlation-enabled multicast systems should be devised but to explore the foundations.

Upon a successful match, our FRIP algorithm in Alg. 2.1 discards not only consumed, matched messages, but also predating buffered ones. We refer to these semantics as *infix&prefix* disposal. More precisely, upon a successful match $[m_1, ..., m_n]$, for each message $m_i$, all messages of the same type received prior to $m_i$ are discarded with $m_i$ via the garbage collection mechanism DEQUEUE. This algorithm, thus, achieves agreement since it is triggered deterministically and also behaves deterministically. Fig. 2.2 shows such an example for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_1$ and $\rho_2 = T_3.a_1 < 20$ (recall that $T_1.a_1$, $T_2.a_1$ and $T_3.a_1$ are shorthand for $T_1[1].a_1$, $T_2[1].a_1$ and $T_3[1].a_1$ respectively). The marked line shows a matched relation. The latest message received is of type $T_2$ with value 7. All messages in the respective queues *in front of* the matched messages are DEQUEUEd.

Correctness of FRIP with Respect to C-MDMcast

**Lemma 1** *FRIP ensures* MDM NO DUPLICATION.

**Proof** SDM NO DUPLICATION ensures that a message cannot be TO-delivered and thus enqueued more than once. If the message results in a successful match, the corresponding message is removed from the queue in the procedure DEQUEUE (Lines 33 - 35 in Alg. 2.1) and, therefore, will not be delivered more than once. Line 11 further ensures that for each matching instance of a same type, after the instance $l$, each subsequent instance message is also delivered and dequeued only once. ■

**Lemma 2** *FRIP ensures* MDM NO CREATION.

**Proof** SDM NO CREATION ensures that a message will only be TO-delivered if it has been TO-broadcast. A message is only TO-broadcast if multicast by Lines 5 - 6. A message may therefore only be delivered if it has been TO-delivered. ■

**Lemma 3** *FRIP ensures* MDM ADMISSION.

**Proof** The function ENQUEUE (Lines 26 - 32) filters out all messages which do not satisfy the unary predicates in the subscription $\Phi$. MATCH (Lines 8 - 19) iterates through the queues to find the *first* instance of the first type in $\Phi$ for which there are messages of the remaining types (or further messages of the same type in such cases) with which *all* predicates in $\Phi$ are satisfied. Hence, any relation $[m_1,\ldots,m_n]$ that is delivered matches the subscription $\Phi$. ■

**Lemma 4** *FRIP ensures* MDM MESSAGE NON-TRIVIALITY.

**Proof** For a given type $T$ of a matching message $m$ which is not dependent on any other type in a conjunction through a binary predicate, given an infinite number of messages of each of the conjoined types, if $m$ is TO-broadcast, it will eventually be TO-delivered by all correct processes. Further, $m$ will not be DEQUEUEd by some later message being matched prior since MATCH (Lines 8-19) looks for the *first* found instance of a type which satisfies the conjunction. $m$, as part of only a unary predicate, will always be a first found instance; even when multiple messages of the same type such as $m$ belong to a predicate, each message will be matched according to the order in the queue and none will be DEQUEUEd due to some later message being matched. ■

**Lemma 5** *FRIP ensures* MDM CONJUNCTION NON-TRIVIALITY.

**Proof** If for any process's conjunction, infinitely many matching messages are multicast, MDM CONJUNCTION NON-TRIVIALITY is ensured. Every multicast namely leads to a TO-broadcast, and since DEQUEUE is only called after a match, it cannot keep an infinite subset of matching TO-broadcast messages from being correlated and matched. Every time messages are discarded from the buffer, including those not delivered, there will still be an infinite number of matching messages TO-broadcast in the future. ■

**Lemma 6** *FRIP ensures* MDM CONJUNCTION AGREEMENT.

**Proof** The underlying Total Order Broadcast guarantees that no two (correct or faulty) processes TO-deliver the same two messages in different orders through SDM AGREEMENT and SDM TOTAL ORDER. Hence, no two processes with the same subscription $\Phi$ have message queue contents which diverge in time with respect to their (identical) streams of TO-delivered messages. The deterministic matching of messages performed in the MATCH function (Lines 8 - 19) ensures that the same relations are delivered at all processes with subscription $\Phi$. ■

**Theorem 2.3.1** *FRIP implements C-MDMcast.*

**Proof** By Lemmas $1 - 6$. ■

### 2.3.2 Total Order Broadcast Using Conjunction Multi-Delivery Multicast

Is total order on messages *necessary* for solving C-MDMcast After all, TOBcast can be used to implement Causal Order Broadcast or FIFO Order Broadcast [42] (just like C-MDMcast), but going the other way is not possible. Alg. 2.2 describes an algorithm $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ to implement Total Order Broadcast over C-MDMcast assuming a majority $\lceil \frac{n+1}{2} \rceil$ of correct processes.

Together with Alg. 2.1, $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ establishes the equivalence between C-MDMcast and Total Order Broadcast in the system and failure model considered. Note that Total Order Broadcast itself is unimplementable in this model; it is equivalent to Consensus [19], which is unsolvable [29]. Thus, implementing the necessary total order requires an oracle such as a failure detector or a more specific *ordering oracle* [65].

Algorithm

In short, the algorithm uses a single type of multicast message $MIP$, which contains the actual application message of type $M$, the sending process's current sequence number as an `Integer` ($I$), as well as the process's identifier of type $P$. Each process is interested in conjunctions consisting in a number of instances of $MIP$ equal to the size of the majority partition of processes in the system. That is, $\Phi = \bigwedge_{i=1..\lceil\frac{n+1}{2}\rceil} MIP[i]$, or more simply $MIP[\lceil\frac{n+1}{2}\rceil]$.

We must ensure a total order among all processes, so each process proceeds in lock-step manner. More precisely, every process at every time has a message that is "under correlation", i.e., a message it has multicast but not yet delivered as part of a relation. This is ensured by SENDER. If a process does not have any pending TO-broadcast messages (a TO-BCAST message is simply added to a queue $broadcasts$ of messages to be broadcast), it simply uses an empty message $\perp$. This is necessary to ensure non-triviality (i.e., an infinite sequence of messages) while a single process only multicasts a single message at a time – less than a majority of processes might be TO-broadcasting.

Since a process can very well deliver several relations that do not contain any of its own messages, and these relations are not necessarily delivered by the underlying C-MDMcast layer in the same order on all processes, they are stored in a buffer upon arrival. The internal messages of the relation are only TO-delivered by the RECEIVER task when certain conditions hold, i.e., the next relation of messages to be TO-delivered must contain messages for which each message sequence number is next in sequence for each respective process. In fact, it is easy to see that any two relations must respectively contain a message from at least one common process – only one message of a given process is under correlation at a time and every relation contains $\lceil\frac{n+1}{2}\rceil$ messages. Those sequence numbers are used to break ties. Given that at any point in time there is only one message per process under correlation, we cannot have two relations with messages from two processes with inverse respective sequence number orders. This argument can be extended to any number of transitively connected relations.

Executed by every process $p_i$

Initialisation:
  $broadcasts \leftarrow \emptyset$ {*Output message buffer*}
  $tbdelivered \leftarrow \emptyset$ {*To be delivered*}
  $seq \leftarrow 0$ {*Last own message created*}
  $last\_sent \leftarrow 0$ {*Last own message sent*}
  $last\_recv \leftarrow 0$ {*Last own msg recved*}
  $last\_delivered[] \leftarrow 0$ {*Seq # hash by PID*}
**task** SENDER
  **if** $last\_recv = last\_sent$ **then** {*no $p_i$ msg*}
    **if** $broadcasts \neq \emptyset$ **then**
      C-MDMCAST$([m, seq', p_i]) \mid seq' = \min(seq'' \mid [..., seq''] \in broadcasts)$
      $broadcasts \leftarrow broadcasts \backslash \{[m, seq']\}$
      $last\_sent \leftarrow seq'$
    **else**
      $seq \leftarrow seq + 1$
      C-MDMCAST$([\perp, seq, p_i])$
      $last\_sent \leftarrow seq$
To TO-BCAST$(m)$:
  $seq \leftarrow seq + 1$
  $broadcasts \leftarrow broadcasts \cup \{[m, seq]\}$
**upon** C-MDMDLVR$_{\Phi=MIP\wedge\ldots\wedge MIP}([m, seq', p_j]_{1..\lceil\frac{n+1}{2}\rceil})$ **do**
  $tbdelivered \leftarrow tbdelivered \cup \{[m, seq', p_j]_{1..\lceil\frac{n+1}{2}\rceil}\}$
  **if** $\exists k \in [1..\lceil\frac{n+1}{2}\rceil] \mid p_{j_k} = p_i$ **then**
    $last\_recv \leftarrow seq'_k$
**task** RECEIVER
  **if** $\exists\{[m, seq', p_j]_{1..\lceil\frac{n+1}{2}\rceil}\} \in tdbelivered \mid \forall k \in [1..\lceil\frac{n+1}{2}\rceil]: seq' = last\_delivered[p_{j_k}] + 1$ **then**
    $tbdelivered \leftarrow tbdelivered \backslash \{[m, seq', p_j]_{1..\lceil\frac{n+1}{2}\rceil}\}$
    **for all** $k = 1..\lceil\frac{n+1}{2}\rceil$ **do**
      $last\_delivered[p_{j_k}] \leftarrow last\_delivered[p_{j_k}] + 1$
      **if** $m_k \neq \perp$ **then**
        TO-DLVR$(m_k)$

Alg. 2.2.: Algorithm $\mathcal{T}_{C-MDMcast\rightarrow TOBcast}$ implementing TOBcast with C-MDMcast.

Intuitively, this may be explained using Fig. 2.3. The graph in Fig. 2.3, for processes $p_1$, $p_2$ and $p_3$, starts where the queues of TO-BCAST messages already contain a number to be C-MDMcast. Each process C-MDMcasts the first message in its queue and updates its expected sequence number. After some time, the C-MDMcast layer matches $m_1^1$ and $m_1^2$, (where an ellipse covers the processes from which the messages were matched), and eventually MDM-delivers the relation $[m_1^1, m_1^2]$ to all processes. Note, that in the underlying C-MDMcast layer, *relations* are not guaranteed to be MDM-delivered in the same order to all processes as seen later by $p_1$. Since for $p_3$, this relation does not contain a message it has previously C-MDMcast, it does not C-MDMcast another message. Since the relation has been MDM-delivered to $p_2$ and $p_3$, each may TO-deliver the respective messages since each is next in sequence. In the figure, TO-DLVR$(m_1^1)$ $(m_1^2)$ denotes the events TO-DLVR$(m_1^1)$ followed by TO-DLVR$(m_1^2)$. Process $p_2$ may C-MDMcast another message since it has received one it has previously C-MDMcast.

Later, the C-MDMcast layer matches another relation $[m_2^2, m_1^3]$ and eventually MDM-delivers this relation to all processes. After this relation is MDM-delivered to both $p_2$ and $p_3$, they may also TO-deliver the respective messages since the messages once again are next in the expected sequence.

For process $p_1$, the relations have been MDM-delivered in a different order than for the other processes. The first relation that is MDM-delivered to $p_1$ contains a message that is out of sequence (i.e., message $m_2^2$) from what is expected, so this relation is buffered. Then, the relation $[m_1^1, m_1^2]$ is MDM-delivered to $p_1$. Now, since every message within this relation is next in expected sequence, the respective messages may be TO-delivered and the next expected sequence numbers may be updated. Also, since the message that $p_1$ has previously C-MDMcast is in this relation, $p_1$ may now C-MDMcast another message. Finally, $p_1$ may TO-deliver the messages from the relation $[m_2^2, m_1^3]$ since each message is now next in sequence.

Thus, the intuition is that for any two relations $r_1$ and $r_2$ containing messages from a majority $\lceil \frac{n+1}{2} \rceil$ of processes, each contain at least one message from the same process. Among those two messages, one message will have a lower sequence number, and thus the relation $r_1$ which contains that message may be ordered before the relation $r_2$ containing the message with the higher sequence number. Inversely, it is not possible to have two relations which contain messages from two respective processes with inverse sequence numbers. Suppose a relation $r_1$ contains a message with sequence number $s_1^i$ from process $p_i$, and another relation $r_2$ contains a message with sequence number $s_2^i$ (where $s_2^i > s_1^i$) from the same process. The very existence of the relation $r_2$ indicates that $r_1$ has already been MDM-delivered to $p_1$, which then C-MDMcasts a second message of sequence $s_2^i$ since processes only C-MDMcast further messages after having received its previous C-MDMcast message. It would thus be impossible for $r_1$ to contain a message with sequence number $s_2^j$ from process $p_j$ and $r_2$ to contain a message with sequence number $s_1^j$ (where $s_2^j > s_1^j$). For this to have happened, either $p_i$ must have sent $s_1^i$ and $s_2^i$ before ever receiving a relation, or process $p_j$ would have had to send $s_2^j$ before $s_1^j$. Because processes proceed in lock-step manner and sequence numbers are monotonically increasing, neither case described is possible, demonstrating the intuition.

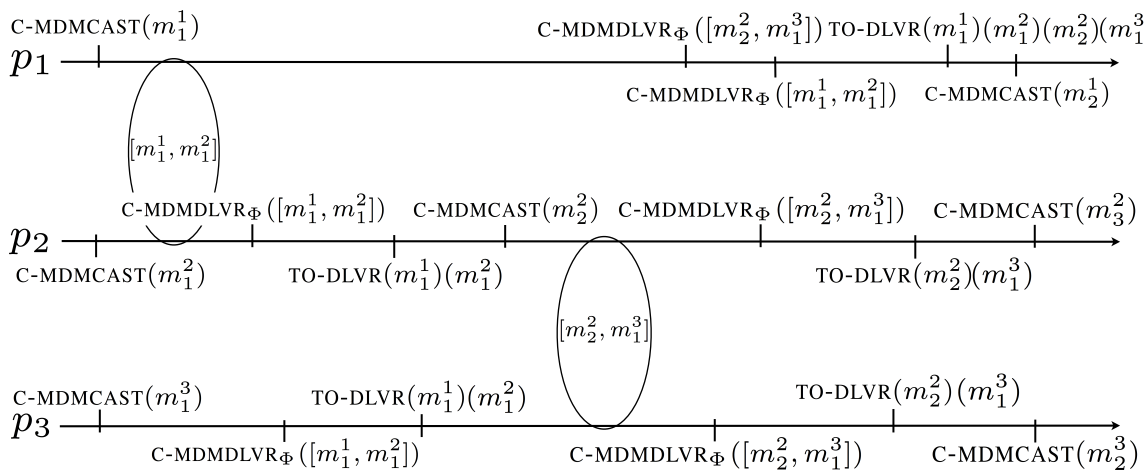



Figure 2.3.: Example demonstrating the total order of message delivery from Alg. 2.2 where TO-DLVR$(m_1)\ldots(m_n)$ summarizes events TO-DLVR$(m_1)\ldots$ TO-DLVR$(m_n)$.

Correctness of $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ (Alg. 2.2) with Respect to TOBcast

**Lemma 7** $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ *ensures* SDM NO DUPLICATION.

**Proof** MDM NO DUPLICATION ensures that no message can be delivered more than once. Each message multicast is added to $tbdelivered$ at most once. Thus, each message is TO-delivered at most once since once a message is TO-delivered, the relation containing that message is removed from $tbdelivered$. ■

**Lemma 8** $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ *ensures* SDM NO CREATION.

**Proof** MDM NO CREATION ensures a message is delivered only if multicast. Each message is only multicast once it is placed in $broadcasts$ by Lines 18 and 20 and correspondingly placed in $tbdelivered$ if delivered within a relation. Only messages (except $\perp$ messages) in $tbdelivered$ are TO-delivered. ■

**Lemma 9** $\mathcal{T}_{C-MDMcast \rightarrow TOBcast}$ *ensures* SDM VALIDITY.

**Proof** The proof is in two steps. First, it will be shown that relations will be delivered by correct processes in a lock-step manner, which assures that messages of some form are delivered. Then, it will be shown that a relation containing message $m$ which a process $p_i$ has multicast will eventually be delivered by $p_i$ and thus TO-delivered.

Each process will multicast application (TO-broadcast) messages (Line 11 of Alg. 2.2) when present, or $\perp$ messages (Line 16 of Alg. 2.2) when there are no application messages to send. Because there is a majority $\lceil \frac{n+1}{2} \rceil$ of correct processes, there will always be at least $\lceil \frac{n+1}{2} \rceil$ messages which may be correlated at any given time. Since each process only multicasts one message at a time, each message a process receives of its own that is delivered in a relation will be a message in sequence; and that process may therefore multicast another message. If a process delivers a relation containing any number of in-sequence messages, there may be other messages in the same relation that are out-of-sequence from the respective processes. However, a process will not TO-deliver any messages in a relation unless *all* the messages are next in respective sequence by Line 26 of Alg. 2.2. Between any two relations, there will always be at least one message of a same process $p_k$ in each of the relations. There is, therefore, transitively an order that may be determined by the relation that $p_k$ delivered first. Therefore, there are further relations that contain the messages which precede each of the out-of-sequence messages that the corresponding processes have already delivered. By MDM CONJUNCTION AGREEMENT, those relations will eventually be delivered and all the internal messages will therefore be TO-delivered.

When a correct process $p_i$ multicasts a message $m$, $m$ may only be delivered when it is correlated with $\lfloor \frac{n}{2} \rfloor$ additional messages. By MDM MESSAGE NON-TRIVIALITY, since the subscription $\Phi$ has no unary (or binary) predicates on any of the messages, $m$ will eventually be delivered. When the relation containing $m$ is delivered, there may be other messages in that relation that are out of sequence from the respective processes. Since the relations containing those messages are guaranteed to be delivered and thus all the preceding in-sequence messages TO-delivered (as shown above), $m$ will thus be TO-delivered ensuring SDM VALIDITY. ■

**Lemma 10** $\mathcal{T}_{C-MDMcast \to TOBcast}$ *ensures* SDM AGREEMENT.

**Proof** All processes have the same subscription $\Phi$. By SDM VALIDITY, if one process $p_i$ multicasts a message $m$, $m$ will be correlated with $\lfloor \frac{n}{2} \rfloor$ other messages, matching $\Phi$, and thus be TO-delivered by $p_i$. By MDM CONJUNCTION AGREEMENT, all processes will deliver the relation containing $m$ and place that relation in $tbdelivered$. As was first shown for SDM VALIDITY, if for some process, there are messages out-of-sequence in the same relation as $m$, the in-sequence messages will eventually be TO-delivered so that $m$ and all the messages in the same relation may also be TO-delivered by that process. By MDM CONJUNCTION AGREEMENT, all processes will eventually deliver all the same relations. Through the deterministic order (as shown in Lemma 9) in which relations are delivered, all messages in the respective delivered relations will eventually be TO-delivered, thus, Alg. 2.2 ensures SDM AGREEMENT. ■

**Lemma 11** $\mathcal{T}_{C-MDMcast \to TOBcast}$ *ensures* SDM TOTAL ORDER.

**Proof** Correct processes deliver the same relations (by Lemma 10), and these can be ordered deterministically (cf. Lemma 9). SDM TOTAL ORDER holds as the messages within these relations are TO-delivered deterministically (Lines 26 - 31 of Alg. 2.2). ■

**Theorem 2.3.2** $\mathcal{T}_{C-MDMcast \to TOBcast}$ *implements Total Order Broadcast.*

**Proof** By Lemmas $7 - 11$. ■

## 2.4 Subsumption

This section discusses a stronger agreement property, capturing the intuition that subscriptions can include others, and transposing it to the respective delivered relations.

### 2.4.1 Motivation

*Subscription subsumption*, i.e., the recognition of inclusion or covering relationships among subscriptions, is an important concept in publish/subscribe systems [4, 15, 77]. It is used both for scaling, in terms of time needed to match a message against subscriptions (first matching a message against the broadest subscription before matching it, only if the match succeeds, to any covered subscriptions, etc.), as well as in terms of space (by using intermediate nodes and covering subscriptions to abstract many subscriptions or nodes). The same intuition — that any message matching a given subscription is delivered also to any subscription covering the former one — can be applied to multi-message delivery scenarios, yet a precise definition of corresponding properties and their implementation is much more involving when the delivery of a message depends on others.

### 2.4.2 Property

We now introduce MDM COVERING CONJUNCTION AGREEMENT, a stronger property than MDM CONJUNCTION AGREEMENT presented previously in Section 2.2.3.

Formalizing such a property is not trivial because one would also want to retain agreement on (sub-)relations, i.e., that messages delivered *together* as part of the more specific subscription are delivered *together* as well for the more generic one. This leads to fundamental limitations. MDM COVERING CONJUNCTION AGREEMENT only holds for conjunctions which are respectively "*extended to the right*" with respect to the subscription order $\prec_N$, and the condition on disjointness of the sets of types, e.g., between $\Phi$ and $\Phi'$, makes the sub-conjunctions *independent*:

<u>MDM COVERING CONJUNCTION AGREEMENT</u> $\exists$DLVR$^i_{\Phi\wedge\Phi'}([m_1,...,m_{n,...}])$ |
$((\mathbb{T}(\Phi) = [T_1,...,T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \Rightarrow \forall p_j \in correct(F)\backslash\{p_i\} \mid \Phi \in \Psi(p_j):$
$\exists$DLVR$^j_{\Phi}([m_1,...,m_n])$

MDM COVERING CONJUNCTION AGREEMENT is not defined as a symmetric implication (with $\Phi(p_j)$=$\Phi \wedge \Phi''$). The presence of a matching set of messages for a sub-relation given by $\Phi'$ namely does not imply a timely or even eventual occurrence of a matching set for $\Phi''$ conjoined by $p_j$ with $\Phi$, not even by MDM CONJUNCTION NON-TRIVIALITY. MDM COVERING CONJUNCTION AGREEMENT becomes trivially symmetric if $\Phi' = \top$ (thus subsuming MDM CONJUNCTION AGREEMENT).

Also note that not only must the types of the conjunction $\Phi$ be equal, but the predicates must also be equivalent, i.e., no process may extend $\Phi$ with another predicate of the same respective types. Consider that process $p_j$ has defined a predicate $\Phi_j = T_1 \wedge T_2$ which could simply mean to deliver the first found instance of a message of type $T_1$ with the first instance of a message of type $T_2$. Second, a process $p_i$ has defined a predicate $\Phi_i = \Phi_j \wedge T_2.a_1 < 3$. Now suppose, as shown in Fig. 2.4, that a sequence of messages of types $T_1$ and $T_2$ arrive in the following order: $m_1^1 \cdot m_2^1 \cdot m_1^2 \cdot m_2^2$. It is clear that $\Phi_j[m_1^1, m_2^1]$ holds, but assume that $m_1^2.a_1 = 4$ ($> 3$) and $m_2^2.a_1 = 2$ ($< 3$). Process $p_j$ would then deliver $[m_1^1, m_1^2]$ followed by $[m_2^1, m_2^2]$ but process $p_i$ would deliver $[m_1^1, m_2^2]$. Since $m_2^2$ is matched with different messages in both cases, neither the agreement property of Section 2.2.3 nor MDM COVERING CONJUNCTION AGREEMENT is met.

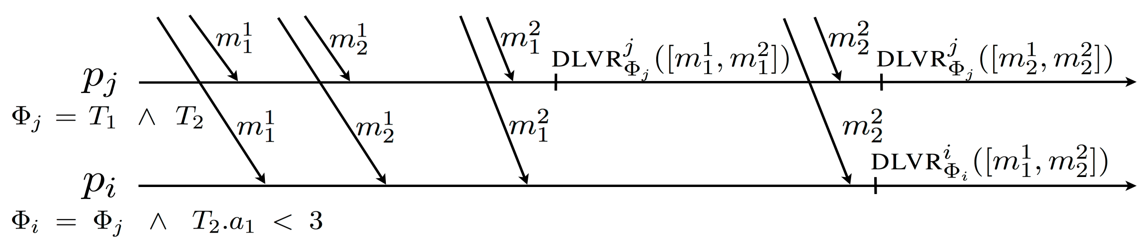



Figure 2.4.: Graph illustrating the order of reception of messages (e.g., $m_1^1$) vs. when they are delivered as part of a relation (e.g., $[m_1^1, m_1^2]$).

Thus, by example, if process $p_j$ defines a conjunction $\Phi_j = T_1.a_1 = v$ and a second process $p_i$ wishes to extend the conjunction $\Phi_j$ with another predicate, it could be such that $\Phi_i = \Phi_j \wedge T_2.a_2 = v'$ but not be of the form (a) $\Phi_i = T_1.a_1 = v' \wedge \Phi_j$, (b) $\Phi_i = \Phi_j \wedge T_2.a_2 = T_1.a_1$, or (c) $\Phi_i = T_1.a_1 \leq v$. (a) is impossible since matching on several message types at any given process must proceed in a deterministic order, and any choice for a given type will affect all the choices for subsequent types. (b) and (c) would require all processes to know of the subscriptions of all other processes (and many messages to be discarded), which we deem overconstraining.

The example subscriptions $\Psi_S$, as defined in Section 2.2.1, and $\Psi'_S$, defined in Section 3.3.3, would exhibit the necessary conditions for MDM COVERING AGREEMENT. That is, the common predicates over the `EarningsReport` and `StockQuote` types would yield the same (sub)-relations for $\Psi_S$ and $\Psi'_S$, where $\Psi'_S$ would deliver relations containing the above with an additional message of type `USDollar`.

### 2.4.3 Correctness of FRIP with Respect to MDM COVERING CONJUNCTION AGREEMENT

**Theorem 2.4.1** *FRIP ensures* MDM COVERING CONJUNCTION AGREEMENT.

**Proof** MDM COVERING CONJUNCTION AGREEMENT is provided as messages of individual types are handled independently by the matching in Alg. 2.1. If two processes $p_i$ and $p_j$ define conjunctions $\Phi \wedge \Phi_i$ and $\Phi$ respectively, as long as $\Phi_i$ is disjoint with $\Phi$ (thus messages that match with $\Phi$ are independent of messages matching $\Phi_i$), then if a match is found for $p_i$, there is a subset $s$ of the relation for which $\Phi$ is true.

Because of SDM AGREEMENT and SDM TOTAL ORDER, no two processes enqueue the same two messages in different orders. Thus, for every type in $\Phi$, both $p_i$ and $p_j$ will have queue contents which remain identical since any messages received by $p_i$ of any type in $\Phi_i$ will be placed in different queues. Thus, if $p_i$ delivers a relation, one of two possibilities occur; either the last message received that triggered the match on $p_i$ is in $s$, thus of a type in $\Phi$, or the last message received is not in $s$, thus of a type in $\Phi_i$.

If the last message received by $p_i$ is in $s$, due to SDM AGREEMENT and SDM TOTAL ORDER, then $p_i$ and $p_j$'s queues over the set of types for $\Phi$ were identical before the message was received by either $p_i$ or $p_j$. Further, by SDM AGREEMENT, if a message is received by $p_i$, $p_j$ will also receive that message, making the queues identical again. Because of the deterministic matching on Lines 8-19 of Alg. 2.1, $p_j$ will also deliver $s$.

Conversely, if the last message received by $p_i$ is not in $s$, then there are messages already in the queues for the types of $\Phi$ which match $s$. Thus, by SDM AGREEMENT and SDM TOTAL ORDER, $p_j$ will have already received the messages in $s$ which would have triggered a match on $p_j$. Messages matching $\Phi_i$ do not affect the order of matching or cause any of the messages in $s$ to be dequeued on $p_i$ when delivering messages corresponding to $\Phi \wedge \Phi_i$. Thus, MDM COVERING CONJUNCTION AGREEMENT holds. ■

## 2.5 Disjunction Multi-delivery Multicast (D-MDMcast)

We now extend C-MDMcast to support disjunctions, thus defining the problem of Disjunction Multi-delivery Multicast (D-MDMcast) over primitives MCAST and DLVR.

### 2.5.1 Predicate Grammar

We now consider lifting the limitation made so far on the number of conjunctions in a disjunction, allowing the full grammar of Section 3.3.1 to be used. For simplicity we however rule out the case of a disjunction that contains several identical conjunctions, i.e., $\forall \Psi = \Phi_1 \vee ... \vee \Phi_n$, $l, k \in [1..n]$: $\Phi_k = \Phi_l \Rightarrow k = l$. In practice, we can remove all but one copy.

DLVR is still parameterized by a conjunction $\Phi_k$ for a given invocation, which can be, however, any $\Phi_k \in \Psi(p_j)$ for a given process $p_j$'s subscription $\Psi(p_j)$.

Note at this point that $\vee$ is not interpreted as an *eXclusive* OR. Our non-triviality and agreement properties introduced in Section 2.2.3 as well as the stronger property introduced in Section 2.4 thus remain valid for disjunctions since conjunctions within a disjunction are handled independently with respect to messages deliveries.

Executed by every process $p_i$. Reuses ENQUEUE, DEQUEUE, and MATCH from FRIP.

**init**
  $\Psi \leftarrow \Phi_1 \vee \ldots \vee \Phi_o$
  $\Phi_l \leftarrow \rho_1 \wedge \ldots \wedge \rho_m$
  $Q_l[T] \leftarrow \emptyset$   {*MQs by type $T$ for $\Phi_l$*}
To MCAST$(m)$:
  TO-BCAST$(m)$

**upon** TO-DLVR$(m)$ **do**
  **for all** $\Phi_l \in \Psi$ in order **do**
    **if** $T(m) \in \mathbb{T}(\Phi_l)$ **then**
      **if** ENQUEUE$(m, \Phi_l, Q_l)$ **then**
        $[m_1, ..., m_k] \leftarrow$ MATCH$(\emptyset, \Phi_l, Q_l)$
        **if** $k \neq 0$ **then**   {*Not an empty set*}
          DEQUEUE$([m_1, ..., m_k], Q_l)$
          DLVR$_{\Phi_l}([m_1, ..., m_k])$

Alg. 2.3.: D-FRIP algorithm implementing D-MDMcast using TOBcast.

### 2.5.2 Algorithm

We now present an algorithm D-FRIP (*D*isjunction-FRIP) to implement the properties provided by D-MDMcast using TOBcast. This algorithm (see Alg. 2.3) reuses the auxiliary functions ENQUEUE, DEQUEUE, and MATCH from FRIP in Alg. 2.1. In D-FRIP, however, every process maintains one message queue per message type *per* conjunction (queues could trivially be shared across conjunctions). For example, for a disjunction $\Psi = \Phi_1 \vee \Phi_2$ where $\mathbb{T}(\Phi_1)$=$\mathbb{T}(\Phi_2)$=$[T_1, T_2]$, $\Phi_1 = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 < 20$ and $\Phi_2 = \rho_3 \wedge \rho_4$ where $\rho_3 = T_1.a_1 > T_2.a_2$ and $\rho_2 = T_2.a_1 < 20$, a process maintains two queues for type $T_1$ and $T_2$, one each for $\Phi_1$ ($Q_1[T_1]$ and $Q_1[T_2]$) and $\Phi_2$ ($Q_2[T_1]$ and $Q_2[T_2]$).

The primary change with respect to FRIP consists in a new response to a TO-delivery. The new primitive dispatches a message to conjunctions in a deterministic order, as a same message can now lead to multiple MATCHes and DLVRies.

### 2.5.3 Correctness of D-FRIP with Respect to D-MDMcast

**Lemma 12** *D-FRIP ensures* MDM NO DUPLICATION.

**Proof** From Lemma 1, no message will be enqueued, delivered and/or dequeued more than once for any conjunction. D-FRIP holds a separate queue per conjunction. The primitives ENQUEUE and DLVR are each called at most once per message, per conjunction in

Lines 8 - 14 in Alg. 2.3 and DEQUEUE is called per conjunction only after a match. Therefore, D-FRIP ensures MDM NO DUPLICATION for each conjunction. ■

**Lemma 13** *D-FRIP ensures* MDM NO CREATION.

**Proof** No message is TO-broadcast and hence TO-delivered unless multicast, and Alg. 2.3 only delivers TO-delivered messages. ■

**Lemma 14** *D-FRIP ensures* MDM ADMISSION.

**Proof** By Lemma 3, FRIP ensures MDM ADMISSION for any one conjunction. Since ENQUEUE (Line 10) and MATCH (Line 11) are called per conjunction upon the reception of a message in Alg. 2.3, only valid relations that match at least one conjunction in a subscription are delivered in D-FRIP. ■

**Lemma 15** *D-FRIP ensures* MDM MESSAGE NON-TRIVIALITY

**Proof** By Lemma 4, FRIP ensures MDM MESSAGE NON-TRIVIALITY for any one conjunction. Since individual messages are matched in a first-received order for a conjunction, and since D-FRIP calls the match in Line 11 of Alg. 2.3 for each conjunction in a subscription, D-FRIP ensures MDM MESSAGE NON-TRIVIALITY. ■

**Lemma 16** *D-FRIP ensures* MDM CONJUNCTION NON-TRIVIALITY.

**Proof** By Lemma 5, as long as infinitely many matching messages are multicast, FRIP ensures MDM CONJUNCTION NON-TRIVIALITY for any single conjunction. The proofs for D-FRIP follows from a match for each conjunction within a subscription being independently triggered upon receiving matching messages in Line 11 of Alg. 2.3. ■

**Lemma 17** *D-FRIP ensures* MDM COVERING CONJUNCTION AGREEMENT.

**Proof** By Theorem 2.4.1, FRIP ensures MDM COVERING CONJUNCTION AGREEMENT. If two processes $p_i$ and $p_j$ define conjunctions $\Phi \wedge \Phi_i$ and $\Phi$ respectively, then since MATCH is reused from Alg. 2.1 and is called *deterministically* in Alg. 2.3, the conjunctions will be evaluated independently and thus deliver matching messages when they are received. Since these conjunctions are matched independently of one another, and since FRIP ensures MDM COVERING CONJUNCTION AGREEMENT per conjunction, D-FRIP ensures MDM COVERING CONJUNCTION AGREEMENT. ■

**Theorem 2.5.1** *D-FRIP implements D-MDMcast.*

**Proof** By Lemmas $12 - 17$. ■

## 2.6 Total Order

Section 2.3 showed that total order is required on single *messages* to achieve some form of agreement on relations in C-MDMcast. The same mechanisms for achieving such ordering might, however, help provide total order properties for *relations*.

### 2.6.1 Properties

We define three total order properties for MDMcast below:

<u>MDM TYPE TOTAL ORDER</u> $\exists \text{DLVR}^i_\Phi([..., m, ...])_{t_i}, \text{DLVR}^i_\Phi([..., m', ...])_{t'_i}, \text{DLVR}^j_{\Phi'}([..., m, ...])_{t_j}, \text{DLVR}^j_{\Phi'}([..., m', ...])_{t'_j} \,|\, T(m) = T(m') \Rightarrow (t_i < t'_i \Leftrightarrow \neg(t'_j < t_j))$

<u>MDM CONJUNCTION TOTAL ORDER</u> $\exists \text{DLVR}^i_{\Phi\wedge\Phi'}([m_1, ..., m_n, ...])_{t_i}, \text{DLVR}^i_{\Phi\wedge\Phi'}([m'_1, ..., m'_n, ...])_{t'_i}, \text{DLVR}^j_{\Phi\wedge\Phi''}([m_1, ..., m_n, ...])_{t_j}, \text{DLVR}^j_{\Phi\wedge\Phi''}([m'_1, ..., m'_n, ...])_{t'_j} \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \wedge (\mathbb{T}(\Phi) \cap \mathbb{T}(\Phi'')) = \emptyset \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

<u>MDM DISJUNCTION TOTAL ORDER</u> $\exists \text{DLVR}^i_\Phi([m_1, ..., m_n])_{t_i}, \text{DLVR}^i_{\Phi'}([m'_1, ..., m'_m])_{t'_i}, \text{DLVR}^j_\Phi([m_1, ..., m_n])_{t_j}, \text{DLVR}^j_{\Phi'}([m'_1, ..., m'_m])_{t'_j} \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

None of the properties includes any of the others. MDM TYPE TOTAL ORDER ensures that there is a total (sub-)order on the messages of a same type. MDM CONJUNCTION TOTAL ORDER ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces MDM CONJUNCTION TOTAL ORDER, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure MDM TYPE TOTAL ORDER. Perhaps more obvious is that, inversely, MDM TYPE TOTAL ORDER does not imply MDM CONJUNCTION TOTAL ORDER. MDM DISJUNCTION TOTAL ORDER further sets our model apart from many single-message delivery multicast settings (e.g., traditional publish/subscribe [15]), where subscriptions are conjunctions, and disjunctions are handled independently through multiple conjunctions. Our property strives for total order *across* relations delivered to *distinct* conjunctions in a disjunction.

One might imagine extending MDM CONJUNCTION - and MDM DISJUNCTION TOTAL ORDER to a similar property as below:

<u>MDM JUNCTION TOTAL ORDER</u> $\exists \text{DLVR}^{i}_{\Phi_1 \wedge \Phi'_1}([m_1, ..., m_n, ...])_{t_i}$, $\text{DLVR}^{i}_{\Phi_2 \wedge \Phi'_2}([m'_1, ..., m'_m, ...])_{t'_i}, \text{DLVR}^{j}_{\Phi_1}([m_1, ..., m_n])_{t_j}, \text{DLVR}^{j}_{\Phi_2}([m'_1, ..., m'_m])_{t'_j} \mid$ $((\mathbb{T}(\Phi_1) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi'_1)) = \emptyset \ \wedge \ ((\mathbb{T}(\Phi_2) = [T'_1, ..., T'_m]) \cap \mathbb{T}(\Phi'_2)) = \emptyset$ $\Rightarrow \ (t_i < t'_i \Leftrightarrow t_j < t'_j)$

However, due to the (left-to-right) deterministic order in which disjunctions are evaluated, $p_i$ and $p_j$ could deliver commonly received messages in different orders. If a message $m_1^2$ of type $T_2$ is received by both processes, followed by a message $m_1^1$ of type $T_1$, where $\Phi_1 = T_1 \vee T_2$, then $p_j$ delivers $m_1^2$ before $m_1^1$. However, if message(s) are received by $p_i$ that trigger $\Phi'_1$ before any that satisfy $\Phi'_2$, then $p_i$ will deliver $m_1^1$ before $m_1^2$. Even in a more constraining case, when $\Phi'_1 = \Phi'_2$ as in Fig. 2.5 where $\Phi'_1 = \Phi'_2 = \Phi = T_3$, when the message arrives that triggers $\Phi$ after receiving messages for $\Phi_2$ followed by $\Phi_1$, the matching for $p_i$ is performed left to right and thus, $p_i$ delivers $m_1^1$ before $m_1^2$. Defining the combination of total orders on conjunctions and disjunctions is different from simply ensuring both properties. It is difficult since processes can each have conjunctions which extend conjunctions of the other.

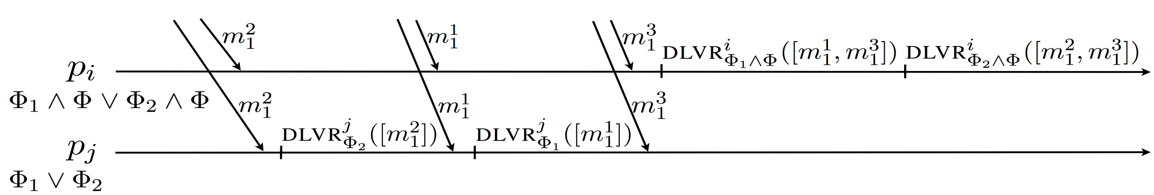



Figure 2.5.: Example showing difficulty/issue of defining generalized MDM JUNCTION TOTAL ORDER: $\Phi_1 = T_1, \ \Phi_2 = T_2, \ \Phi'_1 = \Phi'_2 = \Phi = T_3$.

### 2.6.2 Correctness of FRIP and D-FRIP with Respect to Total Order Properties

**Theorem 2.6.1** *FRIP ensures* MDM TYPE TOTAL ORDER.

**Proof** MDM TYPE TOTAL ORDER is ensured in that TO-BCAST determines a total order for the messages of any specific type, and that first-received matching and infix&prefix disposal retain this order. ■

**Theorem 2.6.2** *FRIP ensures* MDM CONJUNCTION TOTAL ORDER.

**Proof** MDM CONJUNCTION TOTAL ORDER is ensured because the matching deterministically proceeds along types in order of their occurrence in conjunctions and by respecting orders for individual message types. ■

**Theorem 2.6.3** *D-FRIP ensures* MDM TYPE TOTAL ORDER.

**Proof** The proof for MDM TYPE TOTAL ORDER follows that for FRIP as queueing and matching (from FRIP) happen deterministically at each message reception. ■

**Theorem 2.6.4** *D-FRIP ensures* MDM DISJUNCTION TOTAL ORDER.

**Proof** MDM DISJUNCTION TOTAL ORDER is ensured when two processes, $p_i$ and $p_j$, define two separate but equivalent conjunctions. D-FRIP ensures that after receiving each message, both processes deterministically perform matching on those respective conjunctions in the same order (left to right). Since it has been shown that FRIP ensures MDM CONJUNCTION TOTAL ORDER, and D-FRIP reuses MATCH (Line 11) from FRIP per conjunction, D-FRIP, therefore, ensures MDM DISJUNCTION TOTAL ORDER. ■

## 2.7 FIFO and Causal Order

This section investigates the two other ordering properties which are common in the context of Reliable/Total Order Broadcast, namely *FIFO order* and *causal order* [42].

### 2.7.1 FIFO Order

In Total Order Broadcast [42], (uniform) FIFO order may be defined as follows:

<u>SDM FIFO ORDER</u> $\exists \text{TO-BCAST}^i(m)_{t_i}, \text{TO-BCAST}^i(m')_{t'_i}, \text{TO-DLVR}^j(m)_{t_j},$
$\text{TO-DLVR}^j(m')_{t'_j} \mid t_i < t'_i \Rightarrow t_j < t'_j$

Similarly to MDM TYPE TOTAL ORDER, the following property's depends on the equivalence of message types among ordered messages:

<u>MDM TYPE FIFO ORDER</u> $\exists \text{MCAST}^i(m)_{t_i}, \text{MCAST}^i(m')_{t'_i}, \text{DLVR}^j_\Phi([..., m, ...])_{t_j},$
$\text{DLVR}^j_\Phi([..., m', ...])_{t'_j} \mid T(m) = T(m') \wedge t_i < t'_i \Rightarrow t_j \leq t'_j$

This property differs from SDM FIFO ORDER in two ways. Note, that the delivery of $m$ does not imply the delivery of $m'$ within a relation. If $m$ were to be delivered, the only implication is that $m$ matches all predicates for conjunction $\Phi$, but $m'$ may contain attributes which do no match all predicates in $\Phi$; thus, the property may only specify the necessary conditions when both $m$ and $m'$ are delivered.

Firstly, the types $T(m)$ and $T(m')$ must be identical. Secondly, because messages of a same type may be delivered together as part of a stream, the property allows $m$ and $m'$ to be delivered *at the same time*, i.e., in the same relation.

**Theorem 2.7.1** *If TOBcast ensures* SDM FIFO ORDER*, FRIP ensures* MDM TYPE FIFO ORDER.

**Proof** Messages are queued as they are received in the respective type queues by Line 15 of Alg. 2.1. If two messages of the same type are received, they will appear in the queue in the order received. If $m'$ is delivered before $m$, then by infix&prefix disposal, $m$ will be discarded since it appears earlier in the queue for $T(m)$ and $T(m')$. Thus, if both $m$ and $m'$ are delivered, since Alg. 2.1 uses first-received matching, either $m$ must have been delivered before $m'$ or they are delivered in the same relation. ■

The matching semantics and garbage collection can have a direct effect on meeting MDM TYPE FIFO ORDER. The use of last-received matching semantics, for example, would *not* violate this property when infix&prefix disposal is still used. However, if last-received matching were used with, say, infix (only) disposal, then it is possible to violate MDM TYPE FIFO ORDER. In fact, even first-received matching with infix (only) disposal can violate MDM TYPE FIFO ORDER due to the nature of binary predicates: A message $m_i^x$ may currently not match with any other currently received messages. However, some later received message of type $T_x$ from the same sender, e.g., $m_j^x$, might match with other current messages and thus be delivered. With infix (only) disposal, $m_i^x$ will remain in the queue. Later, upon the reception of some new message $m_k^y$ from any sender, $m_i^x$ and $m_k^y$ may now match a binary predicate and thus be delivered as part of a relation, violating FIFO order between $m_i^x$ and $m_j^x$.

One could imagine an alternative guarantee MDM CONJUNCTION FIFO ORDER that allows for any single source to multicast messages of any types and maintain FIFO order between *any* two subsequent messages. It is easy to see that such a guarantee is not implemented in FRIP. To illustrate this, suppose that some process has a conjunction $\Phi$ over two types $T_1$ and $T_2$. Further, suppose that the queue for $T_1$ is empty but the queue for $T_2$ contains many received messages that have not yet been delivered as part of a relation. If another process were to multicast a message $m_i^2$ of type $T_2$ followed by a second message $m_j^1$ of type $T_1$, the messages will be received in the same order they were multicast. However, upon the reception of $m_i^2$, it is queued and no match is found since there are no messages of type $T_1$ in the queue to complete a match. Then, once $m_j^1$ is received, the matching is performed. Since first-received matching is used, it is possible that $m_j^1$ is matched with an earlier received message in the queue for type $T_2$. Further, it is possible that another message of type $T_1$ is later received that is matched and delivered with $m_i^2$. Since relations containing the messages $m_i^2$ and $m_j^1$ were delivered in such a manner that the two messages were delivered in a different order than they were multicast, MDM CONJUNCTION FIFO ORDER is violated.

It is possible to modify FRIP to implement MDM CONJUNCTION FIFO ORDER. A possible solution would be to include tags or sequence numbers in messages and when a process performs a match, it assures that no message is about to be delivered such that after garbage collection, other messages with lower sequence numbers or tags are left. However, this can drastically increase the matching complexity. Another implementation would be to discard *all* messages in every queue upon a match, but this is impractical for most scenarios. While the above example scenario would be avoided by using most recent matching, there are other, more complex scenarios in using most recent matching that could be constructed that still violate MDM CONJUNCTION FIFO ORDER even while using infix&prefix disposal.

### 2.7.2 Causal Order

Causal order can be expressed as the combination of SDM FIFO ORDER and the following SDM LOCAL ORDER property [42]:

SDM LOCAL ORDER $\exists \text{TO-DLVR}^i(m)_{t_i}, \text{TO-BCAST}^i(m')_{t'_i}, \text{TO-DLVR}^j(m)_{t_j}, \text{TO-DLVR}^j(m')_{t'_j} \mid t_i < t'_i \Rightarrow t_j < t'_j$

We propose the following property which, combined with MDM TYPE FIFO ORDER, yields a type-specific form of causal order for relations:

MDM TYPE LOCAL ORDER $\exists \text{DLVR}^i_{\Phi}([..., m, ...])_{t_i}, \text{MCAST}^i(m')_{t'_i}, \text{DLVR}^j_{\Phi'}([..., m, ...])_{t_j}, \text{DLVR}^j_{\Phi'}([..., m', ...])_{t'_j} \mid T(m) = T(m') \wedge t_i < t'_i \Rightarrow t_j \leq t'_j$

This property again brings to surface a number of issues that do not appear in Total Order Broadcast. Here, a message must be delivered as part of a *relation* before the multicast of another message such that MDM TYPE LOCAL ORDER holds. This can be one necessary condition for a causal relationship. However, there may be other forms of causality that might be considered that still relate to this form. For instance, consider a process that is not delivering sets of messages, but rather is waiting for a single message before it multicasts another. This case, although seemingly different, is still covered by the above property if the predicate for a relation is looking for a single message of a single type that satisfies certain conditions. Although the relation is a single message, this scenario is supported by our model. This illustrates that the properties presented in this paper are more general than those for Total Order Broadcast.

As with MDM TYPE FIFO ORDER, the delivery of $m$ does not imply the delivery of $m'$ within a relation. The types $T(m)$ and $T(m')$ must be equivalent here as well. The reasons are slightly different than for FIFO order, however. If a message $m$ is delivered and causes the multicast of another message $m'$, then it is clear that before $m'$ may be multicast, $m$ must have been received and delivered by the sending process and thus $m$ has been received first by all processes in the presence of total order. Further, as with MDM TYPE FIFO ORDER, since the types are equal, the messages will appear in the same queue in the correct order. Thus, by first-received matching and infix&prefix disposal, either both $m$ and $m'$ will be delivered as part of relations in a correct order, or one or both will be discarded.

**Lemma 18** *FRIP implements* MDM TYPE LOCAL ORDER.

**Proof** Because FRIP is implemented over Total Order Broadcast, then if a process $p_i$ has received and delivered a message $m$, another process $p_j$ will have at least received and queued $m$. Then, any message $m'$ of the same type as $m$ that $p_i$ may multicast will be received after $m$ and thus placed in the queue after $m$ by process $p_j$. Thus, if $p_j$ has delivered both $m$ and $m'$ within relations, then because of first-received matching and infix&prefix disposal, $m$ must have been either delivered first or with $m'$ in the same relation, or $m$ would have been discarded if $m'$ was delivered first. ■

**Theorem 2.7.2** *If TOBcast ensures* SDM FIFO ORDER *FRIP implements* MDM TYPE CAUSAL ORDER.

**Proof** By Theorem 2.7.1 and Lemma 18 ■

As with MDM CONJUNCTION FIFO ORDER, a more general MDM CONJUNCTION LOCAL ORDER property could be defined. Its implementation, albeit not impossible, would be yet more constrained than that of MDM CONJUNCTION FIFO ORDER.

Note that an application that requires such FIFO or causal order properties ranging across types could always achieve those in our framework by mapping the desired sets of types to single *union* types (a.k.a. *algebraic* types), e.g., $T_1 +$ *etc.* $+ T_k$.

## 2.8 Related Work

Many early approaches for message aggregation are based on **active databases** that employ fully *centralized* detection of *unicast* messages [17]. An aggregated message is a pattern of messages that a subscriber may be interested in. A composite subscription is a pattern describing the interests of the subscriber for each individual subscription. In the Ode object database [35], a composite subscription can be specified using a regular expression type language and detection is performed using finite state automata. The SAMOS database [34] employs colored Petri Nets for message aggregation.

Message aggregation has been vigorously investigated in the context of **content-based publish/subscribe** systems. Most content-based publish/subscribe systems rely on a *broker network* responsible for routing messages to the subscribers. *Advertisements* are typically used to form routing trees in order to avoid flooding of subscriptions throughout the broker network. Upon receiving a message $m$, a broker determines the subset of parties (subscribers, brokers) with matching interests, and forwards $m$ to them. Well-known examples of such systems include SIENA [15] and Gryphon [85]. A broker network can be used to gather all publications, e.g., conjunctions, for the individual subscriptions and match. A successful match results in the generation of a composite event in the broker network, which needs to be delivered to the interested subscribers. Typically, no guarantees are provided on correlation in this case. If two subscribers correlate the same types of messages, which are published by two producers respectively, then unless the subscribers are connected to a same edge broker, they may receive the messages through different routes. This leads to different orders among the messages and, consequently, to different matching outcomes even if the two subscribers have the same composite events. Hermes, REBECCA, PADRES and Gryphon are all examples of publish/subscribe systems where recent works ( [60, 66, 79, 85] respectively) provide extensions for correlation. None of these provide agreement properties with multicast, failures, and a decentralized implementation.

Recent work by Zhang et. al. [84] proposes an extension to existing broker network infrastructure in content-based publish/subscribe systems to achieve the equivalent of total order broadcast as a module for PADRES [60]. Several total order properties are discussed, outlying the advantages of total order in current systems. The implemented total order assures that subscribers belonging to the same destination group for any pair of messages are guaranteed not to receive messages in conflicting orders by brokers holding conflicting messages indefinitely until predating messages have been forwarded. While the benefits of total order are demonstrated, stronger properties such as uniform agreement (when processes may fail) are not achieved, and some processes may receive messages that other processes do not in order to assure safety.

**Stream processing**, which denotes a form of aggregation on streams of data, has been the object of intense research. Examples of corresponding systems are Borealis [8] and Cayuga [24]. However, most work considers events to be *uni*cast, or focuses on individual processes and centralized setups in attempt to provide best-effort guarantees. This becomes even more apparent through widely adopted *load shedding* techniques, which constitute a pragmatic attempt of maintaining timely delivery while sacrificing strong delivery guarantees. Ordering guarantees are discussed in StreamCloud [40]. A proposed guarantee is to ensure that operations being split into sub-operations executing in parallel behave equivalently to non-parallel ones. Ordering is achieved based on timestamps assuming well-synchronized clocks implying a *synchronous* system, and some form of placeholder messages are also employed in the absence of application messages which allows redundant stream operations to agree.

## 2.9 Conclusions

As we show in this paper, ordering and agreement are intertwined in aggregated deliveries unlike in single message deliveries. Alas, total order on individual messages is a prerequisite for agreement on delivery of relations; this order can, however, be exploited to order relations. While specific correlation and stream processing models have more expressive subscription grammars, our feasibility results are generic, and apply to more specific models. Indeed, a number of *deterministic* grammar extensions such as arithmetic operators as in $\Phi = T_1.a_1 < T_2.a_1 + 5$ straightforwardly increase expressiveness yet do not contradict our findings or properties.

In practice, using a Consensus-based TO-Broadcast to implement correlation yields high availability yet is very expensive; inversely, a pragmatic sequencer-based approach exposes a single point of failure and a performance bottleneck. The findings presented in this paper have guided the design of FAIDECS (FAir Decentralized Event Correlation System) [82]: a pragmatic scalable correlation-specific total order approach based on a distributed hash-table that determines *merger* processes which handle specific conjunctions or disjunctions among given message types. These merger processes are interconnected in a way which is fundamentally geared at achieving total order, and are replicated to achieve some degree of fault tolerance which is weaker but far less expensive than that achieved by solving Total Order Broadcast in a peer-based manner. The properties provided by FAIDECS include those presented herein for FRIP and D-FRIP, including all per-type ordering properties. Supporting any discussed *intra*-type (as opposed to *per*-type) FIFO and causal ordering properties are likely to lead to more substantial performance penalties. As mentioned though, union types can be used in the model to achieve such properties at the desired granularity.

# 3 FAIDECS: FAIR DECENTRALIZED EVENT CORRELATION [1]

The abstraction of application *events* is useful not only for reasoning about distributed systems [57], but also for building such systems [15, 85].

**Events: Composition and correlation** Event *correlation* [24] enables higher-level reasoning about interactions by supporting the assembly of *composite* events from elementary events [60, 66]. Traditional uses of correlation include intrusion detection [56]; network monitoring [54] enables the improvement of resource usage, e.g., in data centers. More recent application scenarios for correlation include embedded and pervasive systems [38], and sensor networks [72]. *Complex event processing* (CEP) is a computing paradigm based on event correlation, with applications to business process management and algorithmic trading.

**Challenges for event correlation middleware** Reasoning about event composition is, however, far from non-trivial. Early work in active databases [17] explored syntax and semantics of correlation, pinpointing options. Consider a sequence of events $e_1^1 \cdot e_2^1 \cdot e_1^2$, where $e_l^k$ is a the $l$-th received event (instance) of event type $T_k$. This sequence can be matched by a "subscription" correlating two event types $T_1$ and $T_2$ as $[e_1^1, e_1^2]$ (*first received* first) or as $[e_2^1, e_1^2]$ (*most recent* first). However, corresponding systems are centralized and consider events to be *uni*cast.

Many theoretical and practical efforts on event correlation in publish/subscribe systems [15] consider decentralized setups and *multi*cast but focus on efficiency or the number of aggregations, yielding only best-effort guarantees on event delivery. Consider an online auction where the bidding price of a product or advertisement slot is event-driven. If two

[1]PUBLISHED IN MIDDLEWARE 2011
AUTHORS: G. A. WILKIN, K. R. JAYARAM, P. EUGSTER AND A. KHETRAPAL

processes participating in the auction observe the same events in different orders (e.g., one receives the sequence above, the second one receiving $e_1^2 \cdot e_2^1 \cdot e_1^1$), then the event correlation middleware might be unfair to the first process if $e_1^2$ has information that is critical to placing an optimal bid. Or, consider assembly line surveillance through two monitors for fault tolerance. If they observe events differently, they might yield contradicting reports or alarms. During decentralized event correlation, one might not only expect that processes with identical subscriptions deliver identical sets of events, but also that if the subscription of a first process $p_i$ "covers" that of a second process $p_j$, then $p_i$ would deliver anything that $p_j$ does. In production chains, the same complex events triggering alarms can be combined with further events for taking actions further down the chain or triggering more specific alarms. Such *subsumption* is natural in publish/subscribe systems and even key to scalability [15]. Of course, correlation-based systems can currently be designed individually to achieve such properties, e.g., by using proxy processes to merge and multiplex event streams to replicas agreement; corresponding solutions are hardly generic though, and can introduce bottlenecks to performance and dependability.

**Contributions** This paper presents FAIDECS (a *FAI*r *D*ecentralized *E*vent *C*orrelation *S*ystem – pronounced "Fedex"), a middleware system for *fair* decentralized correlation of events *multicast* among processes. Our exact contributions are:

- We present clear and feasible *properties* for *aggregated* deliveries of *sets* of events based on a concise and generic event correlation sub-grammar in FAIDECS. While in single event (message) delivery scenarios, several families of properties have been proposed and investigated (e.g., agreed delivery [42], probabilistic delivery [13], ordering properties [33]), corresponding properties for better understanding correlation-based systems and ensuring "*logical correctness and integrity*" [69] are namely still lacking. Our properties provide *fairness* in the face of failures of processes responsible for merging events: either all or none of the depending processes cease to receive the desired events, while common overlays (e.g., [60]) might continue to deliver differing sets of events to subsets of interested processes. Our properties also include a notion of subsumption on correlation patterns.

- We introduce novel pragmatic algorithms implementing our delivery properties. For illustration purposes, we first describe simple algorithms based on a group broadcast black box. Then we present decentralized solutions implemented in FAIDECS based on a distributed hash-table (DHT), and present the use of lightweight redundancy mechanisms used for fault tolerance.

- An implementation of our algorithms in FAIDECS is evaluated under different workloads. We quantify the benefits of our decentralized approach by comparing them with sequencer-based and token-based total order broadcast protocols providing comparable properties.

**Roadmap** Section 3.1 presents related work. Section 3.2 introduces the system model and assumptions. Section 3.3 presents our correlation model and properties. Section 3.4 proposes corresponding algorithms. In Section 3.5 we empirically evaluate FAIDECS. Section 3.6 concludes with final remarks.

## 3.1 Related Work

Many early approaches for composite event detection are based on *active data-bases* that employ *centralized* detection of events (e.g., [17]). A composite event is a pattern of events that a subscriber may be interested in. A composite *subscription* is a pattern describing the interests of the subscriber.

Event correlation has been vigorously investigated in the context of *content-based publish/subscribe systems*. Most such systems rely on a *broker network* for routing events to the subscribers (e.g., SIENA [15] and Gryphon [4]). *Advertisements* are typically used to form routing trees in order to avoid propagating subscriptions by flooding the broker network. Upon receiving an event $e$, a broker determines the subset of parties (subscribers and brokers) with matching interests, and forwards $e$ to them. Subscription *subsumption* [15] is used to summarize subscriptions and avoid redundant matching on brokers and redundant traffic among them. If any event $e$ that matches a first subscription also matches a second one, then the latter subscription subsumes the former one.

A broker network can be used to gather all publications for the elementary subscriptions and perform correlation matching. A successful match yields a composite event which is delivered to interested subscribers, where no guarantees are typically provided on correlation. If the events matching a composite subscription shared by two subscribers are produced by several publishers, then unless the subscribers are connected to a same edge broker, they may receive the events through different routes. This leads to different orders among the events and consequently to different composite events for the two subscribers. PADRES [60] performs composite event detection for each subscription at the first broker that accumulates all the individual subscriptions, providing no global properties. In the context of Hermes [66], *complex event detectors* using an interval timestamp model are proposed as a generic extension for existing middleware architectures. Hermes uses a DHT to determine rendezvous nodes for publishers and subscribers; however, this can create single points of failure. The framework we propose is inspired by Hermes in that our framework uses specific merger nodes for specific combinations of types, determined by a DHT. However, we replicate the mergers for availability and connect them such as to ensure agreement, ordering and subsumption on composite events.

*Stream processing* is a paradigm closely related to event correlation and much investigated in the last few years. Research around database-backed systems like Aurora [3] or Borealis [76] has led the path. These systems, however, focus on correlation over streams of events with respect to single destinations and do not consider multicasting. Straightforwardly merging two same streams at two different nodes leads to different outcomes. StreamBase[2] is a commercial offspring of these efforts. Cayuga [24] is a generic correlation engine supporting correlation across streams and is based on a very expressive language but is centralized. The Gryphon publish/subscribe systems has similarly added support for streams [85]. Again, the focus is efficiency, leaving properties unclear.

[2]`http://www.streambase.com/`.

## 3.2 Preliminaries

We assume a system $\Pi$ of *processes*, $\Pi$={$p_1$, ..., $p_u$} connected pairwise by reliable channels [11] offering primitives to SEND (non-blocking) and receive (RECEIVE) messages. We consider a crash-stop failure model [42], i.e., a *faulty* process may stop prematurely and does not recover. We assume the existence of a discrete global clock to which processes do not have access and that an algorithm run $R$ consists in a sequence of events on processes. That is, one process performs an action per clock tick which is either of a (a) protocol action (e.g., RECEIVE), (b) an internal action, or (c) a "no-op". A process is faulty in a run $R$ if it fails during $R$, otherwise *correct*.

A failure pattern $F$ is a function mapping clock times to processes, where $F(t)$ gives all the crashed processes at time $t$. Let $crashed(F)$ be the set of all processes $\in \Pi$ that have crashed during $R$. Thus, for a correct process $p_i$, $p_i \in \mathit{correct}(F)$ where $correct(F) = \Pi - crashed(F)$ [42].

For brevity and clarity, we adopt in the following a more formal notation for properties than common. Consider, for instance, the well-known problem of Total Order Broadcast (TOBcast) [42] defined over primitives TO-BROADCAST and TO-DELIVER, which will be used for comparison later on. We denote $\text{TO-DELIVER}^i(e)_t$ as the TO-delivery of a message conveying an event $e$ by process $p_i$ at time $t$, and similarly, $\text{TO-BROADCAST}^i(e)_t$ denotes the TO-broadcasting of $e$ by $p_i$ at time $t$. We elide any of $i, t$, or $e$ when not germane to the context. We write $\exists a$ for an action $a$ (e.g., SEND, TO-BROADCAST) as a shorthand for $\exists a \in R$. The specification of Uniform TOBcast thus becomes:

TOB-NO DUPLICATION: $\exists\text{TO-DELIVER}^i(e)_t \;\Rightarrow\; \nexists\text{TO-DELIVER}^i(e)_{t'} \mid t' \neq t$

TOB-NO CREATION: $\exists\text{TO-DELIVER}(e)_t \;\Rightarrow\; \exists\text{TO-BROADCAST}(e)_{t'} \mid t' < t$

TOB-VALIDITY: $\exists\text{TO-BROADCAST}^i(e) \;\wedge\; p_i \in \mathit{correct}(F) \Rightarrow \exists\text{TO-DELIVER}^i(e)$

TOB-AGREEMENT: $\exists\text{TO-DELIVER}^i(e) \Rightarrow \forall p_j \in \mathit{correct}(F)\backslash\{p_i\}, \exists\text{TO-DELIVER}^j(e)$

TOB-TOTAL ORDER: $\exists\text{TO-DELIVER}^i(e)_{t_i}, \text{TO-DELIVER}^i(e')_{t'_i}, \text{TO-DELIVER}^j(e)_{t_j},$
$\text{TO-DELIVER}^j(e')_{t'_j} \;\Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

## 3.3 FAIDECS Model

Following, we specify composite events in FAIDECS and the properties achieved for corresponding deliveries (DELIVER) with respect to individually generated (MULTICAST) events. In contrast to traditional settings, DELIVER is parameterized by a "subscription" $\Phi$ and delivers *ordered sets* of typed messages representing events.

### 3.3.1 Predicate Grammar

Sets of delivered events — *relations* — represent events aggregated according to specific subscriptions. Subscriptions are combinations of predicates on events in disjunctive normal form based on the following grammar (extended BNF):

| | | | |
|---|---|---|---|
| *Disjunction* | $\Psi ::= \Phi \mid \Phi \vee \Psi$ | *Operation* | $op ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq$ |
| *Conjunction* | $\Phi ::= \rho \mid \rho \wedge \Phi$ | *Predicate* | $\rho ::= T[i].a\ op\ v \mid T[i].a\ op\ T[i].a \mid T[i] \mid \top$ |

$T[i].a$ denotes an attribute $a$ of the $i$-th *instance* of type $T$ ($T[i]$) and $v$ is a value. As syntactic sugar, we can allow predicates to refer to just $T$, which can be automatically translated to $T[1]$. We may use this in examples for simplicity. A type $T$ is characterized by an ordered set of attributes $[a_1, ..., a_n]$, each of which has a type of its own – typically a scalar type such as `Integer` or `Float`. An event $e$ of type $T$ is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of $T$. We assume that types of values in predicates conform with the types of events (e.g., through static type-checking [27]). $T(e)$ returns the type of a given event $e$. It is important to note that we do *not* introduce a set of uniquely identified types $\{T_1, ..., T_w\}$ as we do for processes. This keeps notation more brief in that we can use $[T_1, ..., T_k]$ to refer to an arbitrary ordered set of $k$ types, as opposed to something of the form $[T_{j_1}, ..., T_{j_k}]$.

To later simplify properties, we introduce the *empty* predicate $\top$, which trivially yields *true*. A predicate that compares a single event attribute to a value or two event attributes on the *same* event, i.e., on the same instance of a same type (e.g., $T_k[i].a\ op\ T_k[i].a'$), is a *unary* predicate. When two distinct events (two distinct types or different instances of the same type) are involved, we speak of *binary* predicates ($T_k[i].a \ op \ T_l[j].a', \ k \neq l \vee i \neq j$). We also allow *wildcard* predicates of the form $T[i]$ to be specified; such predicates simply specify a desired type $T[i]$ of events of interest. $T[i]$ implicitly also declares $T[k] \ \forall k \in [1..i-1]$ if not already explicitly declared as part of other predicates in the subscription.

We assume, for presentation brevity, a single subscription per process. The disjunction representing process $p_i$'s subscription is represented as $\Psi(p_i)$. We also rule out disjunctions with several identical conjunctions. In practice, we can simply remove all but one copy. By abuse of notation but unambiguously, we sometimes handle disjunctions (or conjunctions) as sets of conjunctions (or predicates). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$.

For the following, consider an example subscription $\Psi_S$ for an increase in three successive stock quotes after a quarterly earnings report:

$$\begin{aligned}\Psi_S \quad = \quad & \texttt{StockQuote}[0].\texttt{time} > \texttt{EarningsReport}[0].\texttt{time} \wedge \\ & \texttt{StockQuote}[1].\texttt{value} > \texttt{StockQuote}[0].\texttt{value} \wedge \\ & \texttt{StockQuote}[2].\texttt{value} > \texttt{StockQuote}[1].\texttt{value}\end{aligned}$$

We would probably want to introduce arithmetic operators on values [53] to express, e.g., that the local publication `time` of the first stock quote is within some interval of that of the earnings report. Our grammar can be easily extended by such *deterministic* constructs but is intentionally kept simple for presentation and to illustrate the independence of our algorithms from specific grammars.

### 3.3.2 Predicate Types and Evaluation

We assume that a deterministic order $\prec$ exists within subscriptions based on the names of event types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on event types and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly.

The number of events involved in a subscription is given by the number of its types and corresponding instances. More precisely, the types involved in a subscription are represented as *sequences* as they are ordered, and the same type can be admitted multiple times. Such sequences can be viewed as the *signatures* of predicates, defined as follows:

$$\begin{array}{lcllcl}
\mathbb{T}(\Phi \vee \Psi) & = & \mathbb{T}(\Phi) \uplus \mathbb{T}(\Psi) & \mathbb{T}(T[i].a\ op\ v) & = & \mathbb{T}(T[i]) \\
\mathbb{T}(\rho \wedge \Phi) & = & \mathbb{T}(\rho) \uplus \mathbb{T}(\Phi) & \mathbb{T}(\top) & = & \emptyset \\
\mathbb{T}(T_1[i].a_1\ op\ T_2[j].a_2) & = & \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) & = & [\underbrace{T, ..., T}_{i\times}]
\end{array}$$

$\uplus$ stands for in-order union of sequences defined below:

$$\emptyset \uplus [T, ...] \quad = \quad [T, ...] \qquad\qquad [T, ...] \uplus \emptyset \quad = \quad [T, ...]$$

$$\begin{array}{c} [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \\ \\ \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] \end{array} = \begin{cases} [\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec T_2 \\ [\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec T_1 \\ [\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2 \end{cases}$$

Above, $\oplus$ represents simple concatenation. In the previous example, the types involved are thus [`EarningsReport`, `StockQuote`, `StockQuote`, `StockQuote`].

Any subscription $\Psi$ thus involves a sequence of event types $\mathbb{T}(\Psi)$=$[T_1, ..., T_n]$, where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j] T_k = T_i = T_j$. That is, we can have subsequences of identical types. Such a subsequence represents a stream of events of the respective type of length $j - i + 1$ ($T_k[1], ..., T_k[j - i + 1]$).

A subscription is correspondingly evaluated for an ordered set of events $[e_1, ..., e_n]$, where $e_i$ is of type $T_i$. The evaluation of a conjunction $\Phi$ on a relation is written as $\Phi[e_1, ..., e_n]$. For evaluation of an attribute $a$ on an event $e_i$, we write $e_i.a$. Evaluation semantics for predicates are defined as follows:

$$(\Phi \vee \Psi)[e_1, ..., e_n] \quad = \quad \Phi[e_1, ..., e_n] \vee \Psi[e_1, ..., e_n] \qquad (T)[e_1, ..., e_n] \quad = \quad \textit{true}$$

$$(\rho \wedge \Phi)[e_1, ..., e_n] \quad = \quad \rho[e_1, ..., e_n] \wedge \Phi[e_1, ..., e_n] \qquad (\top)[e_1, ..., e_n] \quad = \quad \textit{true}$$

$$\begin{array}{c}(T[i].a \; op \; v)\\ {[e_1, ..., e_n]}\end{array} = \begin{cases} e_{k+i-1}.a \; op \; v & T(e_k) = T \wedge (T(e_{k-1}) \neq T \\ & \quad \vee (k-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\begin{array}{c}(T_1[i].a_1 \; op \, T_2[j].a_2)\\ {[e_1, ..., e_n]}\end{array} = \begin{cases} e_{k+i-1}.a_1 \; op \; e_{l+j-1}.a_2 & T(e_k) = T_1 \wedge (T(e_{k-1}) \neq T_1 \\ & \quad \vee (k-1) = 0) \wedge \; T(e_l) = T_2 \\ & \quad \wedge (T(e_{l-1}) \neq T_2 \vee (l-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

For brevity we may write simply $\Phi$[...] for $\Phi$[...] $=$ *true*.

A process $p_i$ delivers events in response to its subscription $\Psi(p_i)$ through DELIVER. We consider this primitive to be generically typed, i.e., we write $\text{DELIVER}_{\Phi}([e_1, ..., e_n])$ to deliver a relation $[e_1, ..., e_n]$, where $e_j$ is of type $T_j$ such that $\mathbb{T}(\Phi)$=[$T_1, ..., T_n$]. $\text{DELIVER}^i_{\Phi}([e_1, ..., e_n])_t$ denotes a delivery on process $p_i$ in response to $\Phi$ at time $t$, and $\text{MULTICAST}^i(e)_t$ defines the multicast of an event $e$ by $p_i$ at time $t$. Again $i, t$, etc. may be omitted when not germane to the context.

### 3.3.3 Properties

We now present properties for composite events in FAIDECS defined over primitives MULTICAST and DELIVER. From here on, *deliver* refers to DELIVER (vs. *TO-deliver* for TO-DELIVER), and *multicast* refers to MULTICAST (vs. *TO-broadcast*).

#### Basic safety properties

The basic safety properties for FAIDECS are MDM-NO DUPLICATION, MDM-NO CREATION and ADMISSION as shown below:

MDM-NO DUPLICATION: $\exists \text{DELIVER}^i_\Phi([..., e, ...])_t \Rightarrow \nexists \text{DELIVER}^i_\Phi([..., e, ...])_{t'} \mid t' \neq t$

MDM-NO CREATION: $\exists \text{DELIVER}_\Phi([..., e, ...])_t \Rightarrow \exists \text{MULTICAST}(e)_{t'} \mid t' < t$

ADMISSION: $\exists \text{DELIVER}^i_\Phi([e_1, ..., e_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow \Phi \in \Psi(p_i) \wedge \Phi[e_1, ..., e_n] \wedge \forall k \in [1..n] : T(e_k) = T_k$

The MDM-NO DUPLICATION property implies that a same event is delivered at most once for a given conjunction, which may be opposed to certain systems that allow a same event to be correlated multiple times. Our property could easily be substituted to allow a delivery for *every instance* of a type in a given conjunction. We omit this for simplicity of the presented properties and algorithms. MDM-NO CREATION is similar to TO-broadcast specifications [42] in that an event may only be delivered if multicast. ADMISSION ensures type safety and that all events in a relation match the subscription.

### Liveness

ADMISSION can trivially hold while not delivering anything. We have to be careful about providing strong delivery properties on *individually* multicast events though, as events may depend on others to match a given conjunction. Nonetheless, we want to rule out bogus implementations which simply discard all events. We thus propose the following complementary liveness properties:

CONJUNCTION VALIDITY: $\exists \text{MULTICAST}(e^k_l), k \in [1..n], l \in [1..\infty] \wedge p_i \in correct(F) \wedge \exists \Phi \in \Psi(p_i) \mid \Phi[e^1_l, ..., e^n_l] \Rightarrow \exists \text{DELIVER}^i_\Phi([...])_{t_j} \mid j \in [1..\infty]$

EVENT VALIDITY: $\exists \text{MULTICAST}^i(e^x), \text{MULTICAST}^{k,l}(e^k_l), k \in [1..n] \backslash x,\ l \in [1..\infty]$ $\{p_i, p_j, p_{k,l}\} \subseteq correct(F) \mid \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \wedge \forall z \in [w..y] T_z = T(e^x) \wedge$ $\nexists (T(e^x)[x - w + 1].a_1 \ op\ T[r].a_2) \in \Phi \mid (T \neq T(e^x) \vee r \neq x - w + 1) \wedge$ $\Phi[e^1_l, ..., e^{x-1}_l, e^x, e^{x+1}_l, ..., e^n_l] \Rightarrow \exists \text{DELIVER}^j_\Phi([..., e^x, ...])$

These two properties handle the two possible cases that can arise. The first property deals with dependencies across events and can be paraphrased as follows: "If for a correct process $p_i$, there is an infinite number of relations of matching events that are successfully multicast, then $p_i$ will deliver infinitely many such relations." This property is reminiscent of the FINITE LOSSES property of fair-lossy channels [11]. It allows matching algorithms to discard *some* events for practical purposes such as agreement and ordering, yet ensures that when matching events are continuously multicast, a corresponding process will continuously deliver. From the example presented in Section 3.3.1, as long as events of both types are inifinitely published such that infinitely often, three successive, increasing stock quotes are multicast after an earnings report, there will be an infinite number of delivered relations.

EVENT VALIDITY provides a property analogous to validity for single-message deliveries (e.g., TOBcast): If an event is multicast by a correct process $p_i$, and its delivery in response to a conjunction on some correct process $p_j$ is *not* conditioned by binary predicates with other event types, then the event must be delivered by $p_j$ if matching events of all other types are continuously multicast. This latter condition is necessary because the delivery of the event, even in the absence of binary predicates, requires the *existence* of other events (by nature of correlation). The condition also ensures that any unary predicates on the respective event type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force an event to be delivered more than once as the position of the event is not fixed in the implied delivery. The example in Section 3.3.1 does not present a unary predicate, and thus would not be affected by this property. If the subscription $\Psi_S$ were extended to trigger only if the value of the U.S. dollar is below some value `v` as in $\Psi'_S = \Psi_S \quad \wedge$ `USDollar.value < v`, then any event matching this predicate will be delivered with the entire relation given by $\Psi_S$.

Note also that none of these properties is impacted by the presence of multiple instances of a same type in a conjunction. An infinite flow of events of some type implies multiple (a finite number) of infinite flows of that type.

Agreement

The properties so far ensure that as long as matching events are being multicast, processes will eventually deliver relations. We are, however, interested in stronger properties for these delivered relations, which ensure fairness for relations delivered across processes. We define COVERING AGREEMENT:

COVERING AGREEMENT: $\exists \text{DELIVER}^{i}_{\Phi \wedge \Phi'}([e_1, ..., e_{n,...}]) \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \Rightarrow \forall p_j \in correct(F) \backslash \{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DELIVER}^{j}_{\Phi}([e_1, ..., e_n])$

Subsumption only allows "extending conjunctions to the right" as determinism requires some given order for matching. Intuitively, subsumption in the presence of binary predicates is limited since, when comparing two subscriptions with same types, an event of a first type might match both subscriptions without implying that the same holds for a second event.

Note that COVERING AGREEMENT is not defined in a symmetric way (with $\Phi \wedge \Phi'' \in \Psi(p_j)$), as the presence of a matching set of events for a conjunction $\Phi'$ does not imply a timely or even eventual occurrence of a matching set for another sub-relation $\Phi''$ conjoined by $p_j$ with $\Phi$.

Thus, the example subscriptions $\Psi_S$, as defined in Section 3.3.1, and $\Psi'_S$, defined in 3.3.3, would exhibit the necessary conditions for COVERING AGREEMENT. That is, the common predicates over the `EarningsReport` and `StockQuote` types would yield the same (sub)-relations for $\Psi_S$ and $\Psi'_S$, where $\Psi'_S$ would deliver relations containing the above with an additional event of type `USDollar`.

### 3.3.4 Total Order

Intuitively, and as we will illustrate in the following sections, a total order on individual events can be used to achieve agreement on relations. In fact, it is necessary to do so (see Chapter 2 for a formal proof). On the upside, this can be exploited to provide corresponding relation-level properties. We define three types of total order properties below:

EVENT TOTAL ORDER: $\exists\text{DELIVER}^i_{\Phi}([..., e, ...])_{t_i}, \text{DELIVER}^i_{\Phi}([..., e', ...])_{t'_i}, \text{DELIVER}^j_{\Phi'}([..., e, ...])_{t_j}, \text{DELIVER}^j_{\Phi'}([..., e', ...])_{t'_j} | T(e) = T(e') \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

CONJUNCTION TOTAL ORDER: $\exists\text{DELIVER}^i_{\Phi\wedge\Phi'}([e_1, ..., e_n, ...])_{t_i}, \text{DELIVER}^i_{\Phi\wedge\Phi'}([e'_1, ..., e'_n, ...])_{t'_i}, \text{DELIVER}^j_{\Phi\wedge\Phi''}([e_1, ..., e_n, ...])_{t_j}, \text{DELIVER}^j_{\Phi\wedge\Phi''}([e'_1, ..., e'_n, ...])_{t'_j} \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \wedge (\mathbb{T}(\Phi) \cap \mathbb{T}(\Phi'')) = \emptyset \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

DISJUNCTION TOTAL ORDER: $\exists\text{DELIVER}^i_{\Phi}([e_1, ..., e_n])_{t_i}, \text{DELIVER}^i_{\Phi'}([e'_1, ..., e'_m])_{t'_i}, \text{DELIVER}^j_{\Phi}([e_1, ..., e_n])_{t_j}, \text{DELIVER}^j_{\Phi'}([e'_1, ..., e'_m])_{t'_j} \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

None of the properties includes any of the others. EVENT TOTAL ORDER ensures that there is a total (sub-)order on the events of a same type. CONJUNCTION TOTAL ORDER ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces COVERING CONJUNCTION AGREEMENT, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure EVENT TOTAL ORDER. Perhaps more obvious is that, inversely, EVENT TOTAL ORDER does not imply CONJUNCTION TOTAL ORDER. DISJUNCTION TOTAL ORDER further sets our model apart from many single-event delivery multicast settings (e.g., traditional publish/subscribe), where subscriptions are conjunctions, and disjunctions are viewed as being expressed independently through multiple conjunctions. Our property strives for total order *across* relations delivered to *distinct* conjunctions in a *same* disjunction.

## 3.4 Algorithms

We now present ways to implement the properties proposed in the previous section. For illustration purposes, we first outline an approach relying straightforwardly on a total order across multicast events of *all* types. Then, we present novel decentralized algorithms achieving the same properties, leveraging our notion of subscription subsumption.

### 3.4.1 Total Order Broadcast Black Box

A straightforward solution for deterministic event correlation across all processes is to rely on a Total Order Broadcast "black box," with primitives TO-BROADCAST and TO-DELIVER for individual events, ensuring that all correct processes eventually TO-deliver all TO-broadcast events in the same order. To multicast an event $e$ of any type, a process simply performs TO-BROADCAST$(e)$; a TO-DELIVER$(e)$ is handled in a deterministic manner described shortly. Many implementations exist, tolerating different failure patterns [23].

#### Conjunctions

For simplicity, we first focus on single conjunctions for the algorithm in Figure 3.1 before expounding on generic disjunctions. That is, subscription $\Psi_i$ of process $p_i$ consists in a single conjunction $\Phi_i$. DISJUNCTION TOTAL ORDER, in this case, becomes subsumed by CONJUNCTION TOTAL ORDER.

The algorithm in Figure 3.1 uses *first received* matching semantics and *prefix+infix* disposal. In short, the former means that events are matched on a process in the order received by that process. The latter implies the following: Upon a successful match $[e_1, \ldots, e_n]$, for each event $e_i$, all events of the same type received prior to $e_i$ are discarded via the garbage collection mechanism DEQUEUE. These semantics are further elaborated on below.

Each process $p_i$ maintains one queue $Q$ per event type in its conjunction $\Phi$=$\Psi(p_i)$. For example, for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 < 20$, the subscriber maintains one queue for events of type $T_1$ and one for events

Executed by every process $p_i$

1: **init**
2: $\Psi \leftarrow \Phi_1 \vee \ldots \vee \Phi_o$
3: $\Phi_l \leftarrow \rho_1 \wedge \ldots \wedge \rho_m$
4: $Q_l[T] \leftarrow \emptyset$

5: To MULTICAST($e$):
6: TO-BROADCAST($e$)

7: **function** MATCH ($[e'_1, ..., e'_n], \Phi, Q$)
8: $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
9: $l \leftarrow max(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid \exists k \in [1..n] : e_j = e'_k$
10: **for all** $k = (l+1)..h$ **do**
11: **if** $|\mathbb{T}(\Phi)| = n+1$ **then**
12: **if** $\Phi[e'_1, ..., e'_n, e_k]$ **then**
13: **return** $[e'_1, ..., e'_n, e_k]$
14: **else**
15: $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
16: **if** $E \neq \emptyset$ **then**
17: **return** $E$
18: **return** $\emptyset$

19: **upon** TO-DELIVER($e$) **do**
20: **for all** $\Phi_l \in \Psi \mid T(e) \in \mathbb{T}(\Phi_l)$ in order **do**
21: **if** ENQUEUE$(e, \Phi_l, Q_l)$ **then**
22: $[e_1, ..., e_k] \leftarrow$ MATCH$(\emptyset, \Phi_l, Q_l)$
23: **if** $k \neq 0$ **then**
24: DEQUEUE$([e_1, ..., e_k], Q_l)$
25: DELIVER$_{\Phi_l}([e_1, ..., e_k])$

26: **function** ENQUEUE ($e, \Phi, Q$)
27: $win \leftarrow max(j \mid \exists ... T(e)[j].a ... \in \Phi)$
28: **if** $\forall j = 1..win\ ((\exists \rho = (T(e)[j].a\ op\ v) \in \Phi \mid \neg\rho[e]) \vee (\exists(\rho = T(e)[j].a\ op\ T(e)[j].a') \in \Phi \mid \neg\rho[e]))$ **then**
29: **return** false
30: **else**
31: $Q[T(e)] \leftarrow Q[T(e)] \oplus e$
32: **return** true

33: **procedure** DEQUEUE$([e_1, ..., e_m], Q)$
34: **for all** $Q[T] = ... \oplus e_k \oplus e \oplus ..., k \in [1..m]$ **do**
35: $Q[T] \leftarrow e \oplus ...$

Figure 3.1.: Conjunctions/disjunctions with Total Order Broadcast.

of type $T_2$. When TO-delivering an event, $p_i$ will loop *once* by line 20 and first checks whether the type of the event is in $p_i$'s subscription. If so, $p_i$ attempts to ENQUEUE the event. $Q[T(e)] \oplus e$ denotes the appending of event $e$ to the queue of type $T(e)$. The ENQUEUE primitive returns $true$ if the event has been ENQUEUEd, which means that it satisfies all unary predicates on the respective types in the conjunction. Then $p_i$ proceeds to MATCHing. Any single received event may complete up to one relation. If a match $[e_1, \ldots, e_n]$ is identified, the corresponding events are discarded (DEQUEUE) and for each event $e_i$, all preceding events of the same type are discarded from the respective queue for that type. MATCH iterates through the queues deterministically. The semantics attempt to find the *first* instance of the first type in $\Phi$ for which there are events of the remaining types with which $\Phi$ is satisfied. Among all such possibilities, the algorithm recursively seeks for a match with the *first* instance of the second type in $\Phi$, etc. until a match is found or all possibilities are exhausted. For multiple instances of a same type, a first instance is recursively matched with the *first follow-up instance* in the same queue until the needed number of instances is found for that type or the queue is exhausted.

Assuming that the underlying TOBcast primitive ensures TOB-NO CREATION and TOB-NO DUPLICATION (see Section 3.2), it is easy to see how the algorithm of Figure 3.1 ensures the corresponding MDM-NO CREATION and MDM-NO DUPLICATION properties defined in Section 4.2.5. An event $e$, matching all unary predicates of a conjunction $\Phi$, is successfully added to the corresponding queue $Q[T(e)]$ in ENQUEUE (line 31, Figure 3.1). The only way in which $e$ can be removed (and delivered) is together with a matching set of other events fulfilling $\Phi$ (line 23, Figure 3.1), thus ensuring ADMISSION. If matching sets of such events are continuously TO-broadcast, then a match will eventually be determined at line 13 thus ensuring EVENT VALIDITY. CONJUNCTION VALIDITY holds by a similar line of reasoning. The first matching, together with prefix+infix disposal, and the independent handling of events of distinct types ensures EVENT TOTAL ORDER. If two processes $p_i$ and $p_j$ define conjunctions $\Phi \wedge \Phi'$ and $\Phi$ respectively, as long as $\Phi$ and $\Phi'$ are type-disjoint, then events that match with $\Phi$ are independent of any events that match with $\Phi'$. Thus, if there is a matching relation for $p_i$, there is a subset of the relation for which $\Phi$ is true. Since garbage collection is deterministic and is triggered every time an event of a type in $\mathbb{T}(\Phi)$ is TO-delivered and in the same order on $p_i$ and $p_j$ with respect to those deliveries, $p_i$ and $p_j$ will handle respective events identically, ensuring COVERING AGREEMENT. Similarly, CONJUNCTION TOTAL ORDER holds as all processes TO-deliver all relevant events. When $p_i$ identifies a match for $\Phi \wedge \Phi'$, with $\Phi$ and $\Phi'$ type-disjoint, $p_j$ will have TO-delivered the respective subset of events in $\Phi$ already in the same sub-order and thus DELIVERs the respective sub-relations in the same order with any events identified for a $\Phi''$ type-disjoint with $\Phi$.

Disjunctions

When the subscription is a disjunction of several conjunctions, a process maintains one event queue per event type *per* conjunction. For example, for a disjunction $\Psi = \Phi_1 \vee \Phi_2$ where $\mathbb{T}(\Phi_1)$=$\mathbb{T}(\Phi_2)$=$[T_1, T_2]$, a process maintains two queues for type $T_1$ and then two queues for type $T_2$, one each for $\Phi_1$ ($Q_1[T_1]$ and $Q_1[T_2]$) and for $\Phi_2$ ($Q_2[T_1]$ and $Q_2[T_2]$).

Figure 3.1 supports multiple conjunctions in a single disjunction. The primary distinction is in the response to TO-deliveries. The primitive dispatches events to conjunctions *in order* of subscriptions. In contrast to subscriptions of one conjunction, an event can lead to multiple MATCHes and DELIVERies.

Because the MATCHing is performed deterministically as explained previously for a given conjunction, and all processes ENQUEUE the same sets of events in the same order, COVERING AGREEMENT across any two conjunctions is met for the same reasons as for single conjunctions. This property would also be met by any unordered dispatching for multiple conjunctions. The other properties established for conjunctions remain valid due to the duplication of events appearing in distinct conjunctions of a same subscription.

DISJUNCTION TOTAL ORDER is met as any $p_i$ and $p_j$ defining two identical separate conjunctions TO-deliver the respective events (possibly interleaved by those for other conjunctions in $\Psi(p_i)$ and $\Psi(p_j)$ respectively) in the same order. Thus, the correlation for respective relations occurs in the same order.

A simple optimization of the algorithm for subscriptions containing several conjunctions $\Phi_1$,...,$\Phi_m$ with a common event type $T$, omitted for brevity, consists in sharing the queue for $T$ across conjunctions. An event in a queue is then tagged by the index $k$ of a conjunction $\Phi_k$ to indicate that the event has previously been used in a match and DELIVERed for $\Phi_k$. Earlier events of that type should then also be tagged with $k$. Events with tags $\{1, ..., m\}$ may then be discarded. Also, the portrayed matching algorithm performs an exhaustive search and is thus not efficient; however, it suffices to illustrate the relevant properties and can be represented concisely. More elaborate and efficient matching algorithms exist, which offer the same semantics. A common approach consists in storing *partial matches* in specialized data-structures to avoid matching a given event multiple times with same events (cf. [27]). In our implementation of FAIDECS and all evaluated algorithms, we make use of the Rete [30] matching algorithm.

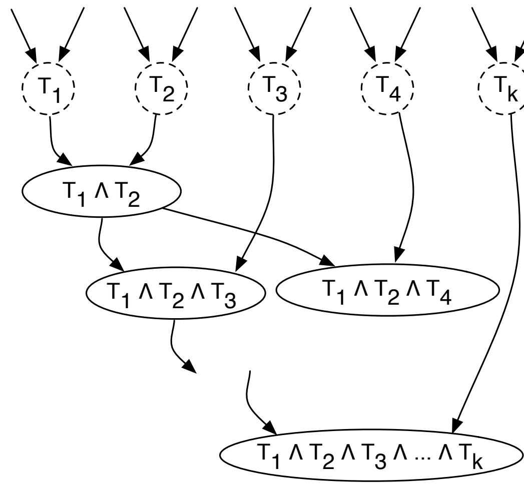



Figure 3.2.: $T_1 \wedge ... \wedge T_j$ denotes the conjunction merger for the respective types $\sqcup[T_1, ...T_j]$ (single instance per type).

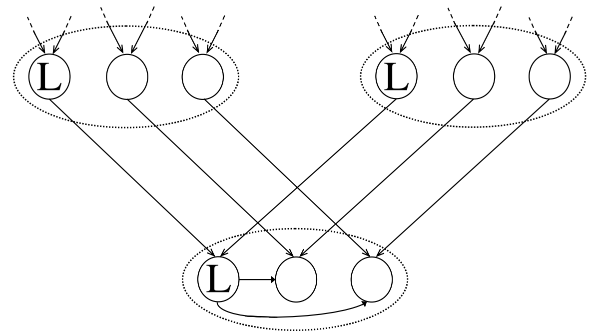



Figure 3.3.: Small-scale FAIDECS merger replication. The dotted ovals are "logical" mergers; circles are processes. $L$ denotes the leader.

### 3.4.2 FAIDECS Decentralized Ordered Merging

One of the simplest and most popular approaches in practice for Total Order Broadcast consists in a sequencer, which orders *all* events. As long as the sequencer remains available (e.g., through replication), the properties presented earlier hold under respective assumptions on failure patterns. A Consensus-based textbook Total Order Broadcast [42] yields the same properties with much better fault tolerance (typically a minority of all processes may fail), yet with a higher overhead. We now present a decentralized solution implementing the same properties, yet with much better scalability characteristics than both and inherently better fault-tolerance than a sequencer-based approach. The solution assumes a distributed hashtable (DHT) or similar mechanism for uniquely identifying a process for a given "role." Lightweight replication mechanisms used for fault-tolerance of such roles are discussed separately thereafter.

Executed by every process $p_i$=PROCESS$(\sqcup[T_1, ..., T_k])$

**init**
  $left \leftarrow$ PROCESS$([\sqcup T_1, ..., T_{k-1}])$
  $right \leftarrow$ PROCESS$([T_k])$
  $subs[p_j]$
  $kids[p_j]$
  INITPARENTS$()$
**procedure** INITPARENTS$()$
  $\Psi' \leftarrow \bigvee_{\Psi \in kids \cup subs} \Psi \backslash \{\rho \in \Psi \,|\mathbb{T}(\rho) \notin \{[T_1], ..., [T_{k-1}]\}\}$
  SEND(CON, $\Psi'$) to $left$
  $\Psi'' \leftarrow \bigvee_{\Psi \in kids \cup subs} \Psi \backslash \{\rho \in \Psi \,|\mathbb{T}(\rho) \neq [T_k]\}$
  SEND(CON, $\Psi''$) to $right$
**upon** RECEIVE(CON, $\Psi$) from $p_j$ **do**
  $kids[p_j] \leftarrow \Psi$
  INITPARENTS$()$
**upon** RECEIVE(SUB, $\Phi$) from $p_j$ **do**
  $subs[p_j] \leftarrow \Phi \backslash \{\rho \in \Phi \mid |\mathbb{T}(\rho)| > 1\}$
  INITPARENTS$()$
**upon** RECEIVE(EV, $e$) **do**
  **for all** $\Psi = kids[p_j]$ **do**
    **if** $\exists l, \Phi \in \Psi \mid \forall \rho = T(e)[l]... \in \Phi : \rho[e]$ **then**
      SEND(EV, $e$) to $p_j$
  **for all** $\Phi = subs[p_j]$ **do**
    **if** $\exists l \mid \forall \rho = T(e)[l]... \in \Phi : \rho[e]$ **then**
      SEND(EV, $e$) to $p_j$

Figure 3.4.: Ordered merging for conjunctions: mergers.

Conjunctions

We first describe an algorithm focusing on single conjunctions, providing the same properties as that of Figure 3.1. All processes with conjunctions on a sequence of event types $[T_1, ..., T_k]$ send their subscriptions to a same process, which is identified as $p_j$ = PROCESS$(\sqcup[T_1, ..., T_k])$, responsible for handling all conjunctions on the involved sequence of types *without duplicates*[3]:

$$\sqcup[T_1, ..., T_1, T_2, ...] \quad = \quad [T_1] \oplus \sqcup[T_2, ...]$$

The function PROCESS relies on a DHT (e.g., a deterministic lookup facility) to deterministically identify such responsible processes, called *mergers*. Lodged at the root of the thereby created overlay network (see Figure 3.2) are mergers responsible for individual event types $T_1$, $T_2$, etc. To ensure the properties with respect to extensions of conjunctions to the right, events undergo an *ordered merge by type* where a merger $p_j$=PROCESS$(\sqcup[T_1, ..., T_k])$ gets events of types $T_1$, ..., $T_k$ from two processes: those identified as PROCESS$(\sqcup[T_1, ..., T_{k-1}])$ and PROCESS$([T_k])$. We term processes in the *role* of subscribers/publishers as *clients*.

[3]We could use different mergers but deduplication simplifies the algorithm.

Executed by every $p_i$. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 3.1

```
init
    Ψ ← Φ
    Φ ← ρ_1 ∧ ... ∧ ρ_m
    Q[T] ← ∅
    SEND(SUB, Φ) to PROCESS(⊔𝕋(Φ))
To MULTICAST(e):
    SEND(EV, e) to PROCESS([T(e)])
upon RECEIVE(EV, e) do
    if ENQUEUE(e, Φ, Q) then
        [e_1, ..., e_l] ← MATCH(∅, Φ, Q)
        if l > 0 then
            DEQUEUE([e_1, ..., e_l], Q)
            DELIVER_Φ([e_1, ..., e_l])
```

Figure 3.5.: Ordered merging for conjunctions: clients.

Figure 3.4 presents the algorithm for merging event types and handling subscriptions corresponding to the merged types. Figure 4.1 presents the algorithm for client processes. Unary predicates are propagated from subscribers to mergers (line 16, Figure 3.4), and from mergers to their ancestor mergers in the form of disjunctions (lines 8-11) since a potential match (i.e., compliant with any unary predicates) for *any* merger or subscriber means a potential match for a parent merger. Forwarding of events received by mergers from their respective parent mergers ($left$) or processes for merged event types ($right$) happens without interruptions by other events and can be achieved by simple local synchronization.

For simplicity, the algorithm in Figure 4.1 handles event queues at clients. The use of shared queues on mergers as described at the end of Section 3.4.1, could lead to savings in global memory overhead by avoiding redundancies. In practice, we have observed that this, however, overburdens mergers, just like a propagation of complete conjunctions instead of only unary predicates to mergers.

Assuming that all subscribers are connected to mergers which are connected to each other before events are multicast, the properties described in Section 3.3.3 are also met by the algorithm in Figures 3.4 and 4.1 thanks to the type-ordered merging of events. COVERING AGREEMENT and CONJUNCTION TOTAL ORDER are ensured as processes with a common "prefix" in their conjunctions, which is type-disjoint with any conjoined predicates, will receive the same events for the prefix and in the same order from the corresponding conjunction merger process.

Executed by every process $p_i$=PROCESS($\sqcup[T_1, ..., T_k]$). Reuses lines 1-11 of Figure 3.4

```
18: upon RECEIVE(EV, e)                          {Rplcs lines 18-24}
19:     for all Ψ = kids[p_j] do
20:         if ∃l, Φ ∈ Ψ | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
21:             SEND(EV, e) to p_j                         {end for}
22:     time ← current time                      {cont frm Line 21}
23:     for all  Φ = subs[p_j] do
24:         if  ∃l | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
25:             SEND(EV, e, time) to p_j
```

Figure 3.6.: Disjunction-enabled ordered merging for conjunctions: mergers.

Executed by every $p_i$. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 3.1

```
1:  init
2:      Ψ ← Φ_1 ∨ ... ∨ Φ_o
3:      Φ_l ← ρ_1 ∧ ... ∧ ρ_m
4:      Q_l[T] ← ∅
5:      R ← ∅
6:      S[T] ← 0
7:      for all Φ_l ∈ Ψ do
8:          SEND(SUB, Φ_l) to PROCESS(⊔𝕋(Φ_l))
9:  To MULTICAST(e):
10:     SEND(EV, e) to PROCESS([T(e)])
11: upon RECEIVE(EV, e, ts) do
12:     if ts > S[T(e)] then
13:         S[T(e)] ← ts
14:         R′ ← {⟨e′, t′⟩ ∈ R | t′ < ts}
15:         R″ ← {⟨e′, t′⟩ ∈ R | t′ > ts}
16:         R ← R′ ∪ {⟨e, ts⟩} ∪ R″
17:         for all ⟨e′, t′⟩ ∈ R ordered on t′ |
              t′ < MIN_T(S[T]) do
18:             for all Φ_l in order do
19:                 if ENQUEUE(e′, Φ_l, Q_l) then
20:                     R ← R\{⟨e′, t′⟩}
21:                     [e_1, ..., e_k] ← MATCH(∅, Φ_l, Q_l)
22:                     if k > 0 then
23:                         DEQUEUE([e_1, ..., e_k], Q_l)
24:                         DELIVER_Φl([e_1, ..., e_k])
```

Figure 3.7.: Ordered merging for conjunctions and disjunctions: clients.

#### Disjunctions

For disjunctions, we essentially need to solve Total Order *Multi*-cast [39] on the event sequences output by conjunction mergers. Using time-stamps and extending the conjunction algorithm of Figures 3.4 and 4.1, order of events is established for clients as needed for disjunctions. More precisely, conjunction mergers following the algorithm of Figure 3.6 timestamps all received messages before passing them to clients which do the actual correlation (Figure 3.7). There is no need for specialized disjunction mergers, which are thus omitted here for simplicity. (If using dedicated disjunction mergers, these can be arbitrarily connected among each other to cover the respective conjunctions.)

If processes send timestamps with events, to achieve order of DELIVERy for relations, an event is only ENQUEUEd (and correspondingly MATCHed) when a receiving process has received events for all other types in its subscription, and the timestamp of that event is less than all the other respective timestamps of other types. As long as all processes which are MULTICASTing events of the respective types continue to do so, for any receiving process, an event will eventually be ENQUEUEd after other events with lower timestamps of other types. This guarantees that all processes receiving the same events over a set of types will ENQUEUE and thus perform a MATCH on them one by one in the same order.

If there are any processes which multicast events at a slower rate than others, then the approach may not be as efficient with the requirement that each event of a type (before being ENQUEUEd) must wait for events of every other type with higher timestamps to be received. To solve this problem for the algorithm in Figure 3.7, if an event has not been received in some time interval by a *conjunction* merging process, then an "empty" event $e_\perp$ may be sent to all processes in $subs[p_j]$, indicating that pending events of other types may be respectively ENQUEUEd. Depending on the targeted scenarios (e.g., publication rate, topology) other information such as rates may be used (additionally).

MDM-NO CREATION and MDM-NO DUPLICATION are met as ENQUEUE and MATCH are only performed on received events, and for a given type, only events with a higher timestamp than the last event of that type are further added to the ordered set $R$ and queue $Q_l$. Since an event is never ENQUEUEd unless its type exists in the process's subscription, and MATCH is performed over every received event, ADMISSION holds. As in Section 3.4.2, EVENT VALIDITY and CONJUNCTION VALIDITY are retained here despite the filtering and discarding of certain events. It is easy to see that the timestamps generated by mergers follow the observed order of event reception, thus respecting CONJUNCTION TOTAL ORDER. Given that events are compared based on timestamps and merged in order of conjunctions, DISJUNCTION TOTAL ORDER is also ensured.

### Joining

The algorithms presented so far *all* rely on a consistent set of event queues across all processes with the same composite subscription if any subscription is issued prior to publications. However, this consistency is violated when two such related processes subscribe to an event stream at different times with respect to the multicasting of events. In order to maintain consistency, we thus employ a simple synchronization algorithm between (a) a joining subscriber process, (b) the corresponding conjunction merger(s), and (c) one of the *existing* subscriber processes with identical conjunctions, if any. This ensures that a joining process starts with a valid state of the respective queues copied from any existing subscriber and does not miss any subsequent events from the merger received also by that existing subscriber after copying the state of its queues.

### Fault tolerance

For presentation simplicity, the algorithms described thus far stipulated single processes returned by function PROCESS() as responsible for given conjunctions, which obviously provides little fault tolerance. In FAIDECS, PROCESS() returns a small fixed number of processes; i.e., the underlying DHT determines a set of replicas for such merger roles. A membership layer monitors the merger processes and ensures that their membership is consistent. Figure 3.3 provides an overview of the replication. A role, or "logical" merger process, is represented by 3 replicas which are contoured by a dotted line. $L$ represents a *leader* process which determines the order between the merged types and communicates that *order* (only) to its peers. These receive the *actual events* independently as depicted in the figure. When a physical merger process (solid circles) $p_i$ fails, its descendant(s) connect to one of $p_i$'s peers. To ensure that no events are missed in the meantime, all replicas regularly acknowledge received and forwarded events to each other; events prior to such acknowledgements are buffered. If a process lags or fails, its peers will attempt to replace it. Using majority-based voting, a minority of (suspected) process failures can typically be tolerated at a time. In addition to benefitting fault tolerance, this small-scale replication also benefits load distribution, in that down-stream processes, including subscribers, distribute uniformly over the replicas.

## 3.5 Evaluation

To demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs, we compare a Java implementation of FAIDECS to the algorithm of Figure 3.1 with 3 different JGroups-based[4] implementations for the Total Order Broadcast black box: (1) a sequencer algorithm, (2) a replicated sequencer (3 replicas) and (3) a token-based algorithm. Figures 3.10, 3.11 and 3.13 summarize our findings.

### 3.5.1 Metrics and Experimental Setup

We used two metrics – *Throughput:* the average number of events delivered per second by a subscriber, and *Latency:* the average delay between the multicasting time of an event and its delivery to a subscriber. The number of subscribers was increased from 10 to 600, and each subscriber had a randomly generated set of subscriptions. Each event consisted of 3 integer attributes with values chosen uniformly at random within $[0..1000]$. All processes were run on 65 nodes in a LAN. Each node is equipped with an Intel Xeon 3.2GHz dual-core processor and 2GB RAM, and runs Linux. A maximum of 15 subscriber processes were run on a single node. The maximum multicast rates varied by setup (e.g., different components became the bottleneck, selectivity of subscriptions varied). We tested scalability of FAIDECS first in terms of conjunctions and then disjunctions.

For conjunctions, we used 3 different distributions of subscriptions, which led to different workloads for actual routing and filtering of events. In scenarios $A$ and $B$, we followed the setup of Figure 3.8, increasing the maximum number of conjoined types (and thus the depth) $k$ from 2 to 4. For scenario $A$, *all* filtering occurred at end nodes rather than in mergers through the selectivity of binary predicates, which differed across conjunctions to achieve the same expected delivery rates at all subscribers in a respective level. This scenario demonstrated the limits of the overlay. In scenario $B$, events were filtered at the mergers through unary predicates propagated upwards from subscriptions, allowing higher aggregate multicast rates than in scenario $A$. Scenario $C$ invariably had 4 event types, and

[4] `http://www.jgroups.org`

subscriptions were over all 6 possible conjunctions ($\binom{4}{2}$). This allowed us to explore the potential of traffic separation. For evaluating scalability with respect to disjunctions, we used scenario $D$, which is the merger overlay shown in Figure 3.9. The maximum level was also varied (from 2 to 4). Subscribers were uniformly distributed across all merger processes and throughput/latency values were averaged for each group of subscribers for a given level.

We expect that the bottleneck in our decentralized algorithms would occur at the merger process(es) which would merge all involved types, limiting throughput consistently for all $k$. All values in Figure 3.10 are normalized with respect to the values obtained with FAIDECS with 10 subscribers connected to a single merger for 2 types in scenario $A$, and with respect to the relations with the largest number of types (independent of the algorithm). Throughput here was approximately 31,400 events/s and latency 150ms. Normalization does not introduce any bias but makes comparison clear, so that values could be reported independent of subscriptions, and so that values may be reported for each level independently.

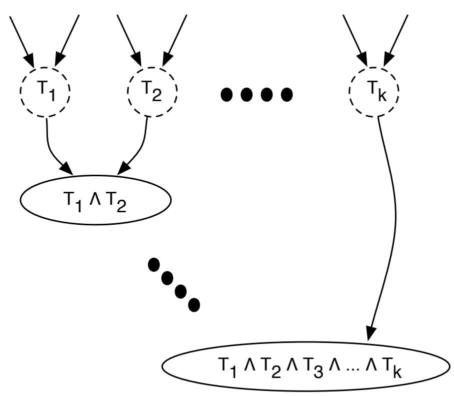



Figure 3.8.: Setup for conjunctions (scenarios $A$ and $B$).

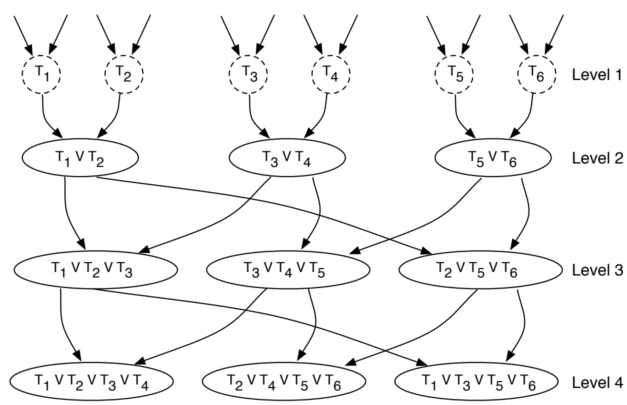
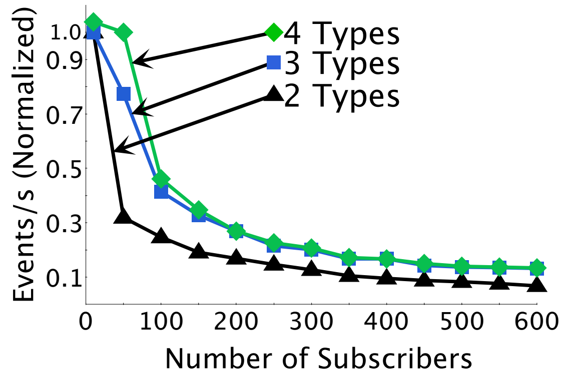
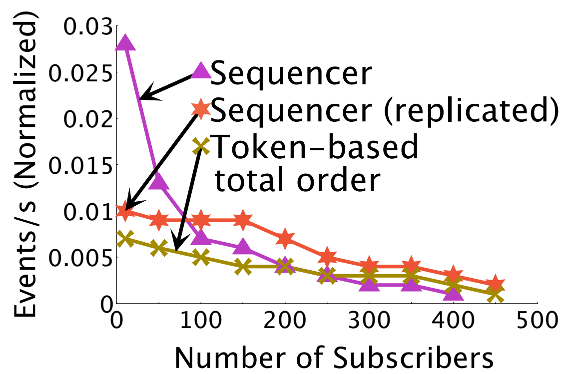


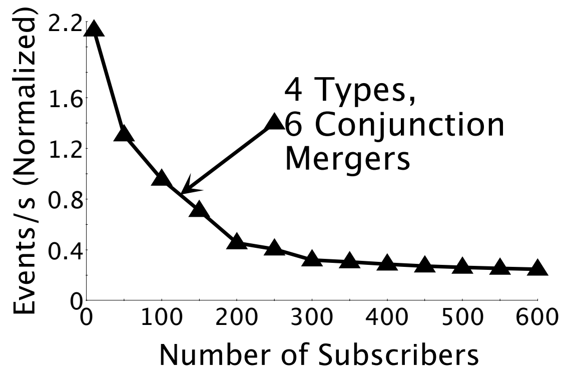

Figure 3.9.: Setup for disjunctions (scenario $D$).



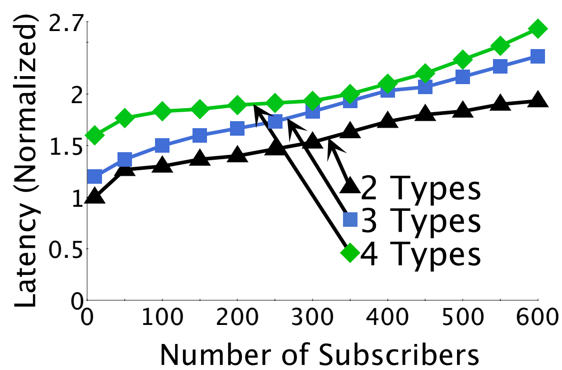

(a) Scenario $A/B$ throughput for conjunctions.



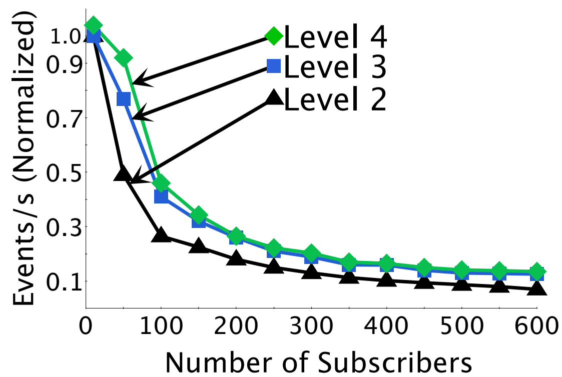

(b) Other total order implementations.



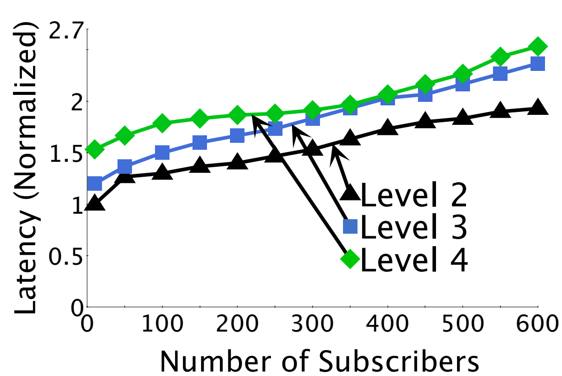

(c) Scenario $C$ throughput for conjunctions.




(d) Scenario $A$ latency for conjunctions.




(e) Scenario $D$ throughput for disjunctions.




(f) Scenario $D$ latency for disjunctions.

Figure 3.10.: Comparing conjunction and disjunction algorithms to a sequencer based approach.

### 3.5.2 Conjunctions

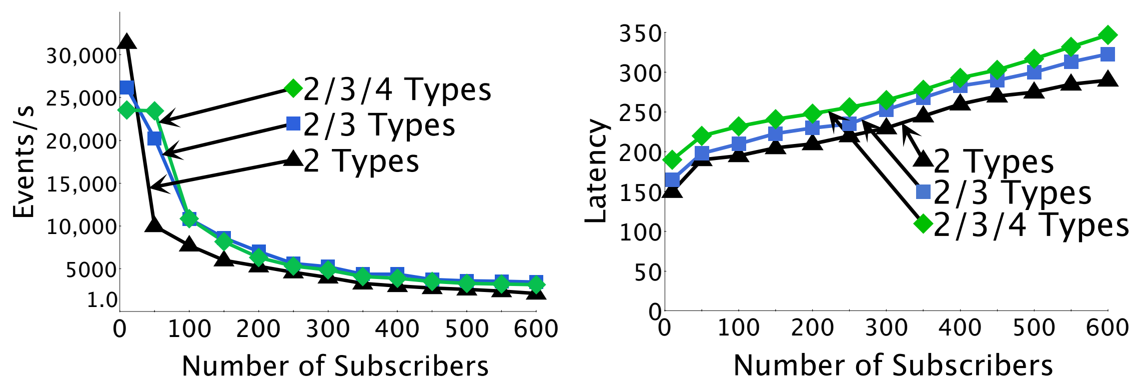
Figure 3.10(a) displays the trend in throughput as the system scales to more subscribers in scenarios $A$ and $B$ with varying number of event types/levels $k$ (see Figure 3.8).

FAIDECS scales very well compared to the approaches shown in Figure 3.10(b), shown separately for a clear relationship among the three implementations since the values start at nearly 3% (about 950 events/s) and remained consistent in all scenarios. Note that IP-multicast was turned off in the test environment which could help throughput for both FAIDECS and the JGroup implementations. In Figure 3.10(b), the token-based algorithm starts with a higher throughput than the sequencer-based one as there were few multicasters competing over the token, but its performance degrades faster due to the inherent cost of its high fault tolerance. Replication helps performance in both FAIDECS and the replicated sequencer due to the load balancing of replicas of a same logical merger process, though less and with an initial cost for the replicated sequencer. The total throughput remained approximately the same in scenarios $A$ and $B$ since propagation of events by mergers was the bottleneck.

Figure 3.10(c) illustrates the scalability and the high throughput of FAIDECS when subscriber interests are in largely disjoint types, following scenario $C$. Thus, FAIDECS scales very well with the addition of an arbitrary number of types to a system, even with transitive correlation across them as in scenario $C$, given enough merger process nodes to support them – the high throughput (about double that of two types for scenario $A$) occurs because every merger only handles relatively few subscribers compared to the other scenarios.

Figure 3.10(d) reports the latency of our algorithms for scenario $A$. As expected, increased depth (conjunctions with increasing number of types) leads to increased latency. Here the "depth" $k$ is fixed to 4, but latency is reported independently at different depths.The observed latency, averaged over all subscribers within each level, was approximately the same with replicated and non-replicated mergers.

Figure 3.11 shows the non-normalized values for conjunctions in Scenarios $A$ and $B$ in FAIDECS. For every level, as shown in Figure 3.8, the values are averaged over all subscribers at the current level as well as the levels with fewer types. That is, by Figure 3.11(a), the throughput values for 4 types are averaged over all subscribers to 4 types as well as with subscribers to 2 and 3 types since the subscribers were evenly distributed across all merger processes. FAIDECS throttles publishers if any merger process becomes saturated. Thus, when publish rates are approximately equal across all publishers, merger processes which merge only a small number of types, as well as the subscribers connected to them, perceive a lower throughput than the merger and subscriber processes interested in more types since the latter mergers tend to become saturated first. Since this was the case in Figure 3.11(a), the throughput for 4 types, when averaged with the throughput for 2 types, resulted in a slightly lower overall average throughput. Figure 3.11(b) shows the averaged latency values. Because the latency values for 4 types were also averaged with the latency values for 2 and 3 types, the line for 2/3/4 Types resulted in a slightly lower overall average latency.




(a) Scenario $A/B$ averaged throughput for conjunctions.

(b) Scenario $A$ averaged latency (ms) for conjunctions.

Figure 3.11.: Conjunction averaged values.

Figure 3.12 shows the latency in milliseconds for the three JGroups implementations for total order. Again, the token-based total order does not perform as well as the other two approaches because of the cost of high fault tolerance.

### 3.5.3 Disjunctions

Figure 3.10(e) compares the scalability of FAIDECS with respect to throughput in scenario $D$. The 3 curves represent different depths of the hierarchy (between 2 to 4 levels). For each curve, the throughput is averaged at the respective level. We observe that the impact on throughput is minimal when the disjunctions are made more complex. As shown

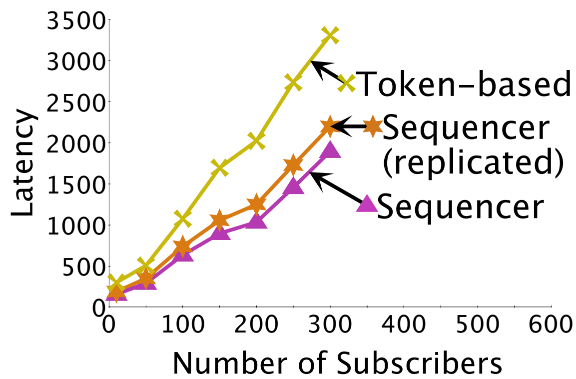
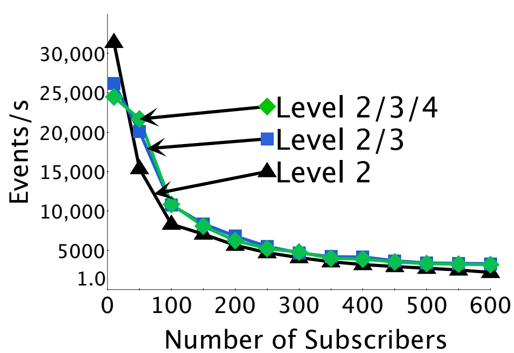


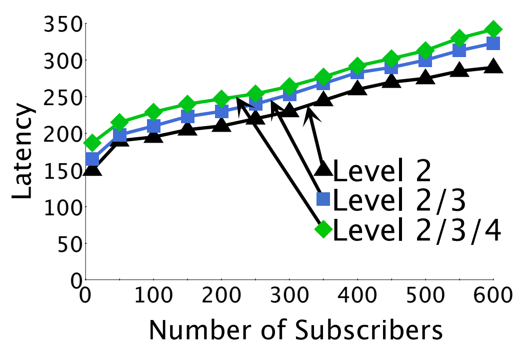

Figure 3.12.: Latency values (ms) for other total order approaches.

in Figure 3.10(f), the latency for 4 types improves slightly. This is because disjunctions provide more than one possibility for event delivery, and the system is no longer throttled by the rate of the slowest upstream process as with conjunctions.

Figure 3.13 shows the non-normalized values for disjunctions in Scenario $D$. As previously explained for Figure 3.11, the values at each level are averaged over all subscribers in a given level as well as the levels with fewer types. Figure 3.13(a) shows the averaged throughput for subscribers and Figure 3.13(b) shows the averaged latency for the propogation of events from publishers to subscribers.




(a) Scenario $D$ averaged throughput for disjunctions.




(b) Scenario $D$ averaged latency (ms) for disjunctions.

Figure 3.13.: Disjunction averaged values.

## 3.6 Conclusions

We have presented decentralized algorithms for event correlation, which are implemented in FAIDECS. Our algorithms provide clear properties, hinging on a novel notion of subscription subsumption tailored to correlation. The same properties can be achieved by less specialized solutions such as sequencer-based schemes, yet our solutions are inherently more scalable and reliable, leading to strong properties with practical performance; our solutions are also more scalable than peer-based approaches, e.g., relying on tokens, while still achieving practical fault-tolerance. We are currently exploring extensions of our algorithms and additional properties (e.g., causal order).

# 4 FAULT TOLERANT EVENT CORRELATION [1]

Event *correlation* enables higher-level reasoning about interactions in distributed applications by supporting the assembly of *composite* events from elementary ones [60,66]. Event correlation is widely used in algorithmic trading, intrusion detection [56], network monitoring [54], or many emerging application scenarios. As we detail in Section 4.6 while presenting related work, most approaches to event correlation exhibit important limitations in *decentralized asynchronous systems prone to crash failures*: (A) no guarantees on composite event deliveries, or (B) no support for multicast and thus no guarantees *across* individual processes; (C) specific architectural setups with centralized components assumed to be reliable or other strong assumptions.

Seminal work on event correlation in the context of *active databases* [17, 34, 35], for instance, just like *stream processing* [8, 24], considers events to be *uni*cast and focuses on individual processes (cf. B) and centralized correlation engines or components (cf. C). Especially in the presence of failures, processes with the same *subscriptions* may thus receive differing sets and combinations of events (if any at all) and thus reach differing outcomes. Event correlation has also been investigated in the context of *content-based publish/subscribe* systems [15] centered on multicast. Examples include Gryphon [85], PADRES [60] and Hermes [66]. However, most such extensions focus on efficiency and matching complexity or on the number of possible combinations and thus yield only best-effort guarantees on event delivery (cf. A) unless relying on centralized rendezvous nodes [66] (cf. C).

The absence of guarantees or the violation of expectations due to failures can have drastic effects [74]. Consider, for example, monitoring a network to decide which one of two gateways to route certain traffic through. Even if the two gateways receive the same

[1] PUBLISHED IN TOIT 2013
AUTHORS: G. A. WILKIN, K. R. JAYARAM AND P. EUGSTER

events but in different orders, each gateway might consider itself to be responsible for routing. Worse even, each can consider the other to be responsible. Of course, individual systems can be designed to deal with some of these issues (e.g., by using a proxy process to merge and multiplex streams to replicas), but corresponding solutions are hardly generic and can easily introduce bottlenecks to performance and dependability.

## 4.1 FAIDECS

As demonstrated by a wealth of literature, achieving strong guarantees in the presence of failures is a hard problem, even for single event/message delivery scenarios [23]. As we showed recently [81], achieving agreement on composite events delivered among processes with identical subscriptions in the presence of process crash failures is as hard as solving the problem of Total Order Broadcast [42] on individual events, which in turn is equivalently hard to the fundamental Consensus [29] problem. Intuitively, we considered total order as a suggestion to focus on achieving that in an efficient manner, and proposed FAIDECS (FAIr Decentralized Event Correlation System – "fedex") [82] which builds specific overlay graphs to consistently merge streams, providing correlation-specific strong guarantees with practical performance. As we demonstrate, this is more efficient than a straightforward solution based on a peer-based (global) total order [42] and also more scalable than a less fault tolerant setup with a centralized "sequencer" [82]. Others have recently proposed solutions to totally order events, albeit layered on top of an order-agnostic overlay [84].

However, the FAIDECS system and model consider very special restricted semantics to achieve strong guarantees. More precisely, only *first received matching* semantics together with *prefix+infix disposal* semantics have been considered thus far: the former means that events from a stream (*type* in FAIDECS) are invariably matched and consumed in their order of reception; the latter means that after matching and delivering an event from a stream, all previously received and still buffered events on the same stream are discarded together with the consumed one. As a consequence, FAIDECS only supports *tumbling* windows on streams of events, but does not support the popular *sliding* windows. In short, FAIDECS thus far provides strong guarantees but with an idiosyncratic model of correlation and subscriptions, making it hard to transpose any results to other systems and languages.

### 4.1.1 Contributions

The goal of this paper is to bridge the gap between strong guarantees proposed for FAIDECS and known correlation models and languages. Achieving this goes through several steps: First, we increase the expressiveness of FAIDECS in order to accommodate existing languages. To that end, we present alternative implementations for the *matching* and *disposal* modules of the FAIDECS correlation engine, yielding alternative semantics to the fixed matching and disposal in FAIDECS [81, 82]. Together with some variations of properties, this allows us to express popular features like sliding windows. Second, we investigate properties that are violated by individual combinations of matching and disposal semantics. Third, we map features of existing correlation languages to these semantic options. This allows us to state the properties that are retained by specific operators and features of these languages if the corresponding engine is substituted for that of FAIDECS in the nodes of the FAIDECS overlay network. If we construct complex events by combining operators, intuitively, the set of properties we achieve for the combination is the *intersection* of the properties retained by each of the operators. This paper thus makes the following contributions. After presenting a comprehensive overview of the FAIDECS model [81] and system [82] (Section 4.2), we

- increase its expressiveness by describing alternative matching and disposal semantics for its correlation engine (Section 4.3);
- pinpoint which properties of the FAIDECS model are violated by which combinations of matching and disposal semantics (Section 4.3.4);

- map four concrete correlation languages — TESLA [21], StreamSQL [49], CEL [25] and EQL[2] — to the semantic framework for FAIDECS, identifying the properties retained by their core operators (Section 4.4).

- demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs by comparing two different Java implementations of FAIDECS with three different implementations of a global total order of which two are fault tolerant (Section 4.5).

Section 4.6 presents related work. Section 4.7 concludes with final remarks. Section 4.4.1 presents an overview of a less popular language TESLA, while Sections 4.4.2–4.4.3 present overviews of the well-known StreamSQL, CEL, and EQL languages.

[2] http://esper.codehaus.org/

## 4.2 FAIDECS Model and System Overview

This section summarizes the FAIDECS model [81] and system [82].

### 4.2.1 System Model and Notation

FAIDECS assumes a system $\Pi$ of *processes*, $\Pi$=$\{p_1$, ..., $p_u\}$, interconnected pairwise by reliable channels [11] with primitives to SEND events and receive (RECV) them. The crash-stop failure model is considered [29], i.e., a faulty process may stop prematurely and does not recover. Further, the existence of a discrete global clock is assumed, which processes cannot access. An algorithm run $R$ consists in a sequence of "system" events (not to be confused with the "higher-level" events correlated) on processes. Similar to other models [5], one process thus performs an action per clock tick, which is either of (a) a protocol action (e.g., RECV), (b) an internal action, or (c) a "no-op."

A failure pattern $F$ is a function mapping clock times to processes, where $F(t)$ yields all the processes that crashed by time $t$. Let $crashed(F)$ be the set of all processes $\in$ $\Pi$ that have crashed during $R$. Thus, for a correct process $p_i$, $p_i \in correct(F)$ where $correct(F) = \Pi - crashed(F)$ [20].

### 4.2.2 Properties

A formal notation is adopted for properties. Consider the well-known problem of Total Order Broadcast (TOBcast) [42] defined over primitives TO-BCAST$(e)$ and TO-DLVR$(e)$ with event $e$. If TO-DLVR$^i(e)_t$ and TO-BCAST$^i(e)_t$ denote the TO-delivery of $e$ by process $p_i$ at time $t$, and the TO-broadcasting of $e$ by $p_i$ at time $t$, respectively, then the property SDM Agreement [42] ("*if some process delivers an event* $e$ *all correct processes eventually deliver* $e$") is defined as follows (note that we elide any of $i, t$, or $e$ when not germane to the context, and write $\exists s$ for a system event $s$ such as a SEND or TO-BCAST as shorthand for $\exists s \in R$): $\exists$TO-DLVR$^i(e) \;\Rightarrow \forall p_j \in correct(F)\backslash\{p_i\}, \exists$TO-DLVR$^j(e)$

### 4.2.3 Predicate Grammar

In FAIDECS, ordered sets of delivered events — *relations* — are events aggregated according to specific subscriptions. Such subscriptions are combinations of predicates on events expressed in disjunctive normal form according to the following grammar:

*Subscription* $\Psi$ ::= $\Phi_1 \vee ... \vee \Phi_n$ *Predicate* $\rho$ ::= $T[i].a\ op\ v \mid T[i].a\ op\ T[i].a \mid T[i] \mid \top$

*Conjunction* $\Phi$ ::= $\rho_1 \wedge ... \wedge \rho_m$ *Operation* *op* ::= $< \mid > \mid \leq \mid \geq \mid = \mid \neq$

A type $T$ can be viewed as a stream of events with identical structure. Such a structure encompasses an ordered set of attributes $[a_1, ..., a_n]$, each of which has a type of its own – typically a scalar type, e.g., `Integer` or `Float`. An event $e$ of type $T$ is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of $T$. $T[i].a$ denotes an attribute $a$ of the $i$-th *instance* of type $T$ ($T[i]$) – multiple instances of a same type allow *windows* over streams to be captured. $v$ is a value. As syntactic sugar, predicates can refer to just $T.a$, which is automatically translated to $T[1].a$.

A predicate that compares a single event attribute to a value or compares two event attributes on the *same* event, i.e., on the same instance of a same type (e.g., $T_k[i].a\ op\ T_k[i].a'$) is referred to as a *unary* predicate. A *binary* predicate involves two distinct events (two distinct types or different instances of the same type) in a predicate ($T_k[i].a\ op\ T_l[j].a',\ k \neq l \vee i \neq j$). To simplify properties, an *empty* predicate $\top$ is also introduced, which trivially yields *true*. Pointless predicates, such as those comparing an attribute of an event to itself ($T_k[i].a\ op\ T_k[i].a$) are prohibited. *Wildcard* predicates of the form $T$ (or $T_k$ for some $k$) simply specify a desired type $T$ of events of interest. $T[i]$ implicitly also declares $T[j]$ $\forall j \in [1..i-1]$ if these are not already explicitly declared in the same subscription.

A process $p_j$'s subscription is referred to as $\Psi(p_j)$. By abuse of notation but unambiguously, disjunctions or conjunctions are sometimes handled as sets (of conjunctions and predicates respectively). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$, or $\Phi_r \in \Psi \Leftrightarrow \Psi = \Phi_1 \vee ... \vee \Phi_n$ with $r \in [1..n]$. Due to space limitations, and as done in a first step in [81] as well, we focus on subscriptions consisting in *single* conjunctions in the following.

As an example, a subscription for an increase in three successive stock quotes following an earnings report is expressed in the above grammar as:
$\Phi_S = \texttt{SQuote}[0].\texttt{time} > \texttt{EReport}[0].\texttt{time} \wedge \texttt{SQuote}[1].\texttt{value} > \texttt{SQuote}[0].\texttt{value}$
$\wedge\ \texttt{SQuote}[2].\texttt{value} > \texttt{SQuote}[1].\texttt{value}$

### 4.2.4 Predicate Types and Evaluation

FAIDECS assumes a deterministic order $\prec$ within subscriptions based on the names of event types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on event types and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly.

The number of events involved in a subscription is given by the number of types and corresponding instances involved. i.e., the types involved in a subscription are represented as *sequences*. As alluded to by the index $i$ in $T[i]$, a same type can be admitted multiple times. Such sequences can be viewed as predicate *signatures*:

$$\begin{array}{lcl@{\qquad}lcl}
\mathbb{T}(\rho_1 \wedge ... \wedge \rho_m) & = & \mathbb{T}(\rho_1) \uplus ... \uplus \mathbb{T}(\rho_m) & \mathbb{T}(\top) & = & \emptyset \\
\mathbb{T}(T_1[i].a_1\ op\ T_2[j].a_2) & = & \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) & = & [\underbrace{T, ..., T}_{i\times}] \\
\mathbb{T}(T[i].a\ op\ v) & = & \mathbb{T}(T[i]) & & &
\end{array}$$

$\uplus$ stands for in-order union of sequences defined below ($\oplus$ represents simple concatenation):

$$\emptyset \uplus [T, ...] = [T, ...] \qquad [T, ...] \uplus \emptyset = [T, ...]$$

$$\begin{array}{l} [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \\ \\ \uplus[\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] \end{array} = \begin{cases}
[\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec T_2 \\
[\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec T_1 \\
[\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2
\end{cases}$$

Any subscription $\Phi$ thus involves a sequence of event types $\mathbb{T}(\Phi)$= $[T_1, ..., T_n]$, where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j]\ T_k = T_i = T_j$, that is, a subsequence of identical types. These imply each a window of $j - i + 1$ events of the respective type. A subscription is evaluated for an ordered set of events $[e_1, ..., e_n]$, where $e_i$ is of type $T_i$. We assume that types of values in predicates are checked statically with respect to the types of events. $T(e)$ returns the type of a given event $e$. Note that we do not introduce a set of uniquely identified types $\{T_1, T_2, ...\}$. This allows for the set of types to be unbounded, which does not violate the assumptions or properties and keeps notation more brief in that we can use $[T_1, ..., T_k]$ to refer to a sequence of $k$ *arbitrary* types, as opposed to, e.g., $[T_{i_1}, ..., T_{i_k}]$.

The evaluation of a conjunction $\Phi$ on a relation is written as $\Phi[e_1, ..., e_n]$. $e_i.a$ denotes the evaluation of an attribute $a$ on an event $e_i$. Evaluation semantics for predicates are thus defined as:

$$\begin{array}{lcllcl}
(\Phi \vee \Psi)[e_1, ..., e_n] & = & \Phi[e_1, ..., e_n] \vee \Psi[e_1, ..., e_n] & (T)[e_1, ..., e_n] & = & \textit{true} \\
(\rho \wedge \Phi)[e_1, ..., e_n] & = & \rho[e_1, ..., e_n] \wedge \Phi[e_1, ..., e_n] & (\top)[e_1, ..., e_n] & = & \textit{true}
\end{array}$$

$$\begin{array}{c} (T[i].a\ op\ v) \\ [e_1, ..., e_n] \end{array} = \begin{cases} e_{k+i-1}.a\ op\ v & T(e_k) = T \wedge (T(e_{k-1}) \neq T \\ & \quad \vee (k-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\begin{array}{c} (T_1[i].a_1\ op\, T_2[j].a_2) \\ [e_1, ..., e_n] \end{array} = \begin{cases} e_{k+i-1}.a_1\ op\, e_{l+j-1}.a_2 & T(e_k) = T_1 \wedge (T(e_{k-1}) \neq T_1 \\ & \quad \vee (k-1) = 0) \wedge T(e_l) = T_2 \\ & \quad \wedge (T(e_{l-1}) \neq T_2 \vee (l-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

Parentheses are used for clarity. For brevity, we write simply $\Phi[...]$ for $\Phi[...] = \textit{true}$.

We consider the DLVR primitive to be generically typed, i.e., for delivering a relation $[e_1, ..., e_n]$, we write $\text{DLVR}_\Phi([e_1, ..., e_n])$ where $e_i$ is of type $T_i$ such that $\mathbb{T}(\Phi)$=$[T_1, ..., T_n]$. Analogous to TOBcast, $\text{DLVR}^i_\Phi([..., e, ...])_t$ defines the delivery event of an event $e$ on process $p_i$ in response to $\Phi$ at time $t$ and $\text{MCAST}^i(e)_t$ defines the multicasting of an event $e$ by $p_i$ at time $t$.

### 4.2.5 Properties

FAIDECS provides primitives MCAST and DLVR, where DLVR is parameterized by a subscription $\Phi$ and delivers relations. From here on, *deliver* refers to DLVR and *multicast* refers to MCAST.

#### Basic Safety Properties

FAIDECS defines three basic safety properties:

MDM No Duplication $\exists\ \text{DLVR}^i_\Phi([..., e, ...])_t \ \Rightarrow\ \nexists \text{DLVR}^i_\Phi([..., e, ...])_{t'} \mid t' \neq t$

MDM No Creation $\exists \text{DLVR}_\Phi([..., e, ...])_t \ \Rightarrow \exists \text{MCAST}(e)_{t'} \mid t' < t$

MDM Admission $\exists \text{DLVR}^i_\Phi([e_1, ..., e_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow\ \Phi \in \Psi(p_i) \wedge \Phi[e_1, ..., e_n]\ \wedge\ \forall k \in [1..n] :\ T(e_k) = T_k$

MDM No Duplication implies that a same event is delivered at most once on any single process for a conjunction, which may be opposed to certain systems that allow a same event to be correlated multiple times. We present an alternative property for sliding windows later on.

#### Liveness

MDM Admission can trivially hold while not performing any deliveries. We have to be careful about providing strong delivery properties on *individually* multicast events though, as events may depend on others to match a given conjunction. FAIDECS proposes the two following complementary liveness properties:

MDM Conjunction Validity $\exists\text{MCAST}(e_l^k), k \in [1..n], l \in [1..\infty] \;\wedge\; p_i \in correct(F) \;\wedge\; \exists\Phi \in \Psi(p_i) \mid \Phi[e_l^1, ..., e_l^n] \;\Rightarrow \exists\text{DLVR}_\Phi^i([...])_{t_j} \mid j \in [1..\infty]$

MDM Event Validity $\exists\text{MCAST}^i(e^x), \text{MCAST}^{k,l}(e_l^k), k \in [1..n]\backslash x, l \in [1..\infty] \mid \{p_i, p_j, p_{k,l}\} \subseteq correct(F) \;\wedge\; \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \;\wedge\; \forall z \in [w..y], T_z = T(e^x) \wedge \nexists(T(e^x)[x-w+1].a_1 \; op \; T[r].a_2) \in \Phi \mid (T \neq T(e^x) \vee r \neq x-w+1) \;\wedge\; \Phi[e_l^1, ..., e_l^{x-1}, e^x, e_l^{x+1}, ..., e_l^n] \Rightarrow \exists\text{DLVR}_\Phi^j([..., e^x, ...])$

These two properties deal with the two possible cases that can arise. The first property deals with dependencies across events and can be paraphrased as follows: "If for a correct process $p_i$ there is an infinite number of relations of matching events that are successfully multicast, then $p_i$ will deliver infinitely many such relations." This property is reminiscent of the Finite Losses property of fair-lossy channels [11]. It allows matching algorithms to discard *some* events for practical purposes (e.g., agreement, ordering), yet ensures that when matching events are continuously multicast, a corresponding process will continuously deliver.

MDM Event Validity provides a property analogous to validity for single event/message deliveries (e.g., TOBcast): If an event is multicast by a correct process $p_i$, and its delivery in response to a conjunction on some correct process $p_j$ is not conditioned by binary predicates with other event types, then the event must be delivered by $p_j$ if events of all other types matching each other are continuously multicast. This latter condition is necessary because the delivery of the event even in the absence of binary predicates requires the *existence* of other events.

The condition also ensures that any unary predicates on the respective event type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force an event to be delivered more than once as the position of the event is not fixed in the implied delivery. The example in Section 4.2.3 does not contain a unary predicate, and thus is not affected by this property. If the subscription $\Phi_S$ were extended to trigger only if the value of the U.S. dollar is below some value `v` as in $\Phi'_S = \Phi_S \wedge$ `USDollar.value < v`, then any event matching this predicate will be delivered with the entire relation given by $\Phi_S$.

Agreement

We now consider a stronger property for relations delivered across processes:

MDM Conjunction Agreement $\exists \text{DLVR}^i_\Phi([e_1, ..., e_n]) \Rightarrow \forall p_j \in \mathit{correct}(F) \backslash \{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DLVR}^j_\Phi([e_1, ..., e_n])$

The uniform MDM Conjunction Agreement property ensures that two correct processes $p_i$ and $p_j$ with identical subscriptions expressed by the conjunction $\Phi$ must deliver the same relation, *without constraining the respective orders of such deliveries*.

FAIDECS also defines a stronger agreement property, which supports *subscription subsumption* on complex events [81], i.e., the recognition of inclusion or covering relationships among subscriptions, a fundamental concept in publish/subscribe systems [4, 15, 77].

MDM Covering Agreement $\exists \text{DLVR}^i_{\Phi \wedge \Phi'}([e_1, ..., e_{n,...}]) \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \Rightarrow \forall p_j \in \mathit{correct}(F) \backslash \{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DLVR}^j_\Phi([e_1, ..., e_n])$

Formalizing such a property is not trivial because one would also want to retain agreement on (sub-)relations, i.e., that events delivered *together* as part of the more specific subscription are delivered *together* as well for the more generic one. This leads to fundamental limitations. MDM Covering Agreement only holds for conjunctions which are respectively "extended to the right" with respect to the subscription order $\prec$, and the condition on disjointness of the sets of types, e.g., between $\Phi$ and $\Phi'$, makes the sub-conjunctions *independent*.

Ordering

FAIDECS defines a number of ordering properties [81], corresponding to the classic FIFO, total, and causal order properties [42]. We consider two total order properties:

**MDM Type Total Order** $\exists \text{DLVR}^i_\Phi([..., e, ...])_{t_i}, \text{DLVR}^i_\Phi([..., e', ...])_{t'_i},\ \text{DLVR}^j_{\Phi'}([..., e, ...])_{t_j}, \text{DLVR}^j_{\Phi'}([..., e', ...])_{t'_j} \mid T(e) = T(e') \Rightarrow (t_i < t'_i \Leftrightarrow \neg(t'_j < t_j))$

**MDM Conjunction Total Order** $\exists \text{DLVR}^i_{\Phi\wedge\Phi'}([e_1, ..., e_n, ...])_{t_i}, \text{DLVR}^i_{\Phi\wedge\Phi'}([e'_1, ..., e'_n, ...])_{t'_i}, \text{DLVR}^j_{\Phi\wedge\Phi''}([e_1, ..., e_n, ...])_{t_j},\ \text{DLVR}^j_{\Phi\wedge\Phi''}([e'_1, ..., e'_n, ...])_{t'_j} \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset\ \wedge (\mathbb{T}(\Phi) \cap \mathbb{T}(\Phi'')) = \emptyset\ \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

MDM Type Total Order ensures that there is a total (sub-)order on the messages of a same type. MDM Conjunction Total Order ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces MDM Conjunction Total Order, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure MDM Type Total Order. Inversely, MDM Type Total Order does not imply MDM Conjunction Total Order.

Similarly to MDM Type Total Order, the following property depends on the equivalence of event types among ordered events:

**MDM Type FIFO Order** $\exists \text{MCAST}^i(e)_{t_i}, \text{MCAST}^i(e')_{t'_i}, \text{DLVR}^j_\Phi([..., e, ...])_{t_j}, \text{DLVR}^j_\Phi([..., e', ...])_{t'_j} \mid T(e) = T(e')\ \wedge\ t_i < t'_i\ \Rightarrow\ t_j \leq t'_j$

The following property yields a type-specific form of causal order for relations when combined with MDM Type FIFO Order (like Local Order and FIFO Order for single-event deliveries [42]):

**MDM Type Local Order** $\exists \text{DLVR}^i_\Phi([..., e, ...])_{t_i}, \text{MCAST}^i(e')_{t'_i}, \text{DLVR}^j_{\Phi'}([..., e, ...])_{t_j}, \text{DLVR}^j_{\Phi'}([..., e', ...])_{t'_j} \mid T(e) = T(e')\ \wedge\ t_i < t'_i\ \Rightarrow\ t_j \leq t'_j$

Executed by every $p_i$.

**init**
  $\Psi \leftarrow \Phi$
  $\Phi \leftarrow \rho_1 \wedge \ldots \wedge \rho_m$
  $Q[T] \leftarrow \emptyset$
  $\text{SEND}(\text{SUB}, \Phi)$ to $\text{PROCESS}(\sqcup\mathbb{T}(\Phi))$

To $\text{MCAST}(e)$:
  $\text{SEND}(\text{EVENT}, e)$ to $\text{PROCESS}([T(e)])$

**function** MATCH $([e'_1, ..., e'_n], \Phi, Q)$
  $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
  $l \leftarrow max(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid \exists k \in [1..n] : e_j = e'_k$
  **for all** $k = (l+1)..h$ **do**
    **if** $|\mathbb{T}(\Phi)| = n + 1$ **then**
      **if** $\Phi[e'_1, ..., e'_n, e_k]$ **then**
        **return** $[e'_1, ..., e'_n, e_k]$
    **else**
      $E \leftarrow \text{MATCH}([e'_1, ..., e'_n, e_k], \Phi, Q)$
      **if** $E \neq \emptyset$ **then**
        **return** $E$
  **return** $\emptyset$

**upon** $\text{RECV}(\text{EVENT}, e)$ **do**
  **if** $\text{ENQUEUE}(e, \Phi, Q)$ **then**
    $[e_1, ..., e_l] \leftarrow \text{MATCH}(\emptyset, \Phi, Q)$
    **if** $l > 0$ **then**
      $\text{DEQUEUE}([e_1, ..., e_l], Q)$
      $\text{DLVR}_\Phi([e_1, ..., e_l])$

**function** ENQUEUE $(e, \Phi, Q)$
  $win \leftarrow max(j \mid \exists ... T(e)[j].a ... \in \Phi)$
  **if** $\forall j = 1..win\ ((\exists \rho = (T(e)[j].a\ op\ v) \in \Phi \mid \neg\rho[e]) \vee (\exists(\rho = T(e)[j].a\ op\ T(e)[j].a') \in \Phi \mid \neg\rho[e]))$ **then**
    **return** false
  **else**
    $Q[T(e)] \leftarrow Q[T(e)] \oplus e$
    **return** true

**procedure** $\text{DEQUEUE}([e_1, ..., e_m], Q)$
  **for all** $Q[T] = ... \oplus e_k \oplus e \oplus ..., k \in [1..m]$ **do**
    $Q[T] \leftarrow e \oplus ...$

Figure 4.1.: First received (FR) matching with prefix+infix (PI) disposal.

### 4.2.6 Decentralized System

FAIDECS implements the above properties with much better scalability than centralized sequencers or peer-based Consensus approaches [42], and inherently better fault-tolerance than a sequencer-based approach. The solution assumes a distributed hashtable (DHT) for uniquely identifying processes for given "roles." Lightweight replication mechanisms of such roles are used for reliabiliy.

#### Mergers

All processes with conjunctions on a sequence of event types $[T_1, ..., T_k]$ send their subscriptions to a same process, identified as $p_j$=$\text{PROCESS}(\sqcup[T_1, ..., T_k])$, responsible for handling all conjunctions on the involved sequence of types *without duplicates*[3]:

$$\sqcup[T_1, ..., T_1, T_2, ...] \quad = \quad [T_1] \oplus \sqcup[T_2, ...]$$

[3] Different processes could be used but deduplication simplifies the algorithm [82].

The function PROCESS relies on a DHT to deterministically identify such responsible processes, called *mergers*. Lodged at the root of the thereby created overlay network (see Figure 4.2) are mergers responsible for individual event types $T_1$, $T_2$, etc. To ensure the properties with respect to extensions of conjunctions to the right, events undergo an *ordered merge by type* where a merger $p_j$=PROCESS$(\sqcup[T_1, ..., T_k])$ gets events of types $T_1$, ..., $T_k$ from two processes: those identified as PROCESS$(\sqcup[T_1, ..., T_{k-1}])$ and PROCESS$([T_k])$. Mergers are replicated in FAIDECS to increase fault tolerance, which emphasizes the focus on total order as opposed to FIFO order (which would trivially solve the former in the absence of multiple destinations).

Clients

The core constitutents of the algorithm in Figure 4.1 which performs full correlation at subscribers based on merged streams are twofold: (1) *matching* (MATCH, Line 8) and (2) *disposal* (DEQUEUE, Line 33). The presented implementations of these modules provide *first received* (FR) matching and *prefix+infix* (PI) disposal respectively [81,82]. In short, the former means that events are matched on a process in the order received by that process. The latter implies the following: Upon a successful match $[e_1, \ldots, e_n]$, for each event $e_i$, all events of the same type received prior to $e_i$ are discarded via the garbage collection mechanism DEQUEUE. Each process $p_i$ maintains one queue $Q$ per event type in its conjunction $\Phi$=$\Psi(p_i)$. For example, for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 <$ 20, the subscriber maintains one queue for events of type $T_1$ and one for events of type $T_2$. When receiving an event, $p_i$ will check if the type of the event is in $p_i$'s subscription. If so, $p_i$ attempts to ENQUEUE the event. $Q[T(e)] \oplus e$ denotes the appending

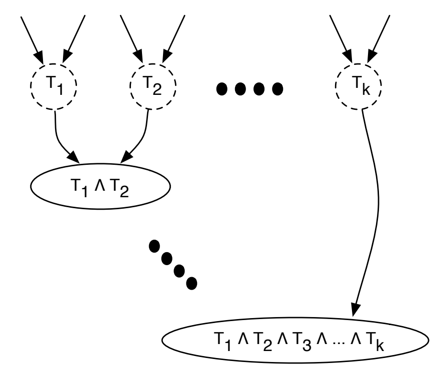



Figure 4.2.: Overlay for conjunctions. Streams merging follows $\prec$

of event $e$ to the queue of type $T(e)$. The ENQUEUE primitive returns $true$ if the event has been ENQUEUEd, meaning it satisfies all unary predicates on the respective types in the conjunction. Then $p_i$ proceeds to MATCHing. Any single received event may complete up to one relation. If a match $[e_1, \ldots, e_n]$ is identified, the corresponding events are discarded (DEQUEUE) and for each event $e_i$, *all* preceding events of the same type are discarded from the respective queue for that type. MATCH iterates through the queues deterministically. The semantics attempt to find the *first* instance of the first type in $\Phi$ for which there are events of the remaining types with which $\Phi$ is satisfied. Among all such possibilities, the algorithm recursively seeks for a match with the *first* instance of the second type in $\Phi$, etc., until a match is found or all possibilities are exhausted. For multiple instances of a same type, a first instance is recursively matched with the *first follow-up instance* in the same queue until the needed number of instances is found for that type or the queue is exhausted.

[81] shows how the algorithm of Figure 4.1 ensures all properties previously outlined. Obviously, there are more efficient ways to implement matching and disposal semantics and will be presented later in Section 4.4 when we present other correlation languages and also in Section 4.5 when evaluating the FAIDECS overlay network using these correlation languages.

## 4.3 Semantic Options

This section presents semantic alternatives to the default FAIDECS matching algorithm of Figure 4.1. For the purpose of this section, we will use an example to demonstrate the different semantics described below. For this example, suppose a process $p_1$ has a queue for type $T_1$ such that $Q[T_1] = \{e_1, e_2, e_3, e_4, e_5\}$ and a second queue for type $T_2$ such that $Q[T_2] = \{e_a, e_b, e_c, e_d\}$ at some instant in time.

### 4.3.1 Event Matching Semantics

The algorithm of Figure 4.1 makes use of first received *non-contiguous* (FR) matching. In this case, events in each respective queue are considered in the FIFO order for matching. (In the example queues above, $p_1$ would thus consider $e_1$ for a match before $e_2$, and so on within the queue $Q[T_1]$ with this type of matching and $e_a$ before $e_b$ for queue $Q[T_2]$. Note, with non-contiguous matching, $e_1$ and $e_3$ could appear in the same relation without $e_2$.) However, in real-time systems and algorithmic stock trading, which require the most up-to-date information, first received matching may not be the most efficient matching when more recent events tend to be the most pertinent. In this instance, *most-recently received* (MR) matching may be a preferred matching semantic: when an event is received, the *last* instance of an event of a first type is matched with the last found instance of the next, etc., moving backwards in the queues as necessary until either a match is found, or all queues are exhausted. (In the example queues above, $p_1$ would thus scan $Q[T_1]$ starting with $e_5$, then $e_4$ for type $T_1$ and correspondingly $e_d$ first for the queue $Q[T_2]$.) Figure 4.3 provides the MATCH function for most-recently received *non-contiguous* (MR) matching, which replaces MATCH of Figure 4.1. As mentioned, Figure 4.1 is an exhaustive search, thus the following extensions are presented for readability rather than efficiency. Later, in Section 4.4, we present three correlation languages with more efficient matching semantics.

The matching is thus still *non-contiguous*, meaning that if more than one event of a same type is matched, these events are not guaranteed to be consecutive events from the queue, but rather may be interleaved by other events in the queue. Some applications may also require that matched events of the same type (i.e., from the same stream) are matched in a *contiguous* manner (meaning, for instance, that if $e_1$ and $e_3$ were to appear in the same relation, either $e_2$ *must* also appear in that relation, or it is not considered a match). Figure 4.4 shows first received *contiguous* (FRC) matching while Figure 4.5 shows most-recently received *contiguous* (MRC) matching. Both MATCH functions assure that a first found instance of an event is only matched with the next consecutive event if possible.

### 4.3.2 Event Consumption Semantics

The needs of applications may dictate also how events are discarded/consumed when relations are delivered. There are four main possibilities (suppose, for the following, from the queues $Q[T_1]$ and $Q[T_2]$ given in Section 4.3, that a relation [$e_2$, $e_4$, $e_b$, $e_c$] is delivered by process $p_1$ for the following semantics).

Prefix+infix (PI) disposal is the default disposal semantics shown in Figure 4.1. It discards all events which have been consumed and all events which have been received prior to the last matched event in each respective type queue. Many events which have never been delivered may be discarded. With this type of disposal semantics, if the above relation is delivered, then $Q[T_1]$ will then contain $\{e_5\}$ and $Q[T_2]$ will contain $\{e_d\}$.

*Infix only* (I) disposal exclusively discards events which have been consumed, i.e., delivered as part of a relation. Undelivered events remain in the queue until they are delivered. This is shown by the DEQUEUE function of Figure 4.6 which replaces that of Figure 4.1. With this type of disposal semantics, when the above relation is delivered, then $Q[T_1]$ will contain $\{e_1$, $e_3$, $e_5\}$ and $Q[T_2]$ will contain $\{e_a$, $e_d\}$.

*Infix+postfix* (IP) disposal discards all events which have been consumed and all currently queued events received *after* these delivered events. Again, many events which have never been delivered may be discarded. This disposal semantic may allow for an alert of some occurrence of interest, but can eliminate repetitive alerts when only one is desired in a certain time frame. IP disposal is demonstrated by the DEQUEUE function of Figure 4.7, Replaces Lines 8-19 of Figure 4.1.

**function** MATCH $([e'_1, ..., e'_n], \Phi, Q)$
  $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
  $l \leftarrow min(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
    $\exists k \in [n..1] : e_j = e'_k$
  **for all** $k = (l-1)..1$ **do**
    **if** $|\mathbb{T}(\Phi)| = n+1$ **then**
      **if** $\Phi[e'_1, ..., e'_n, e_k]$ **then**
        **return** $[e'_1, ..., e'_n, e_k]$
    **else**
      $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
      **if** $E \neq \emptyset$ **then**
        **return** $E$
  **return** $\emptyset$

Figure 4.3.: MR matching.

Replaces Lines 8-19 of Figure 4.1.

**function** MATCH $([e'_1, ..., e'_n], \Phi, Q)$
  $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
  **if** $T_n \neq T_{n+1}$ **then** $\{$*if this type is a new type*$\}$
    $h \leftarrow |Q[T]|$
    $l \leftarrow 1$
  **else** $\{$*look only to the contiguously next event*$\}$
    $l \leftarrow max(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
      $\exists k \in [1..n] : e_j = e'_k$
    $h \leftarrow l + 1$
    $l \leftarrow l + 1$ $\{$*assure loop only looks at next event*$\}$
  **for all** $k = l..h$ **do**
    **if** $|\mathbb{T}(\Phi)| = n+1$ **then**
      **if** $\Phi[e'_1, ..., e'_n, e_k]$ **then**
        **return** $[e'_1, ..., e'_n, e_k]$
    **else**
      $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
      **if** $E \neq \emptyset$ **then**
        **return** $E$
  **return** $\emptyset$

Figure 4.4.: FRC matching.

Replaces Lines 8-19 of Figure 4.1.

**function** MATCH $([e'_1, ..., e'_n], \Phi, Q)$
  $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
  **if** $T_n \neq T_{n+1}$ **then** $\{$*if this type is a new type*$\}$
    $h \leftarrow |Q[T]|$
    $l \leftarrow 1$
  **else** $\{$*look only to the contiguously next event*$\}$
    $l \leftarrow min(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
      $\exists k \in [n..1] : e_j = e'_k$
    $h \leftarrow l - 1$
    $l \leftarrow l - 1$ $\{$*assure loop only looks at next event*$\}$
  **for all** $k = h..l$ **do** $\{$*loop backwards*$\}$
    **if** $|\mathbb{T}(\Phi)| = n+1$ **then**
      **if** $\Phi[e'_1, ..., e'_n, e_k]$ **then**
        **return** $[e'_1, ..., e'_n, e_k]$
    **else**
      $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
      **if** $E \neq \emptyset$ **then**
        **return** $E$
  **return** $\emptyset$

Figure 4.5.: MRC matching.

Replaces Lines 33-35 of Figure 4.1.

**procedure** DEQUEUE$([e_1, ..., e_m], Q)$
  **for all** $Q[T] = ... \oplus e_i \oplus e_k \oplus e \oplus ..., k \in [1..m]$ **do**
    $Q[T] \leftarrow ... \oplus e_i \oplus e \oplus ...$

Figure 4.6.: I disposal.

Replaces Lines 33-35 of Figure 4.1.

**procedure** DEQUEUE$([e_1, ..., e_m], Q)$
  **for all** $Q[T] = ... \oplus e \oplus e_k \oplus ..., k \in [1..m]$ **do**
    $Q[T] \leftarrow ... \oplus e$

Figure 4.7.: IP disposal.

Replaces Lines 33-35 of Figure 4.1.

**procedure** DEQUEUE$([e_1, ..., e_m], Q)$
  $Q[T] \leftarrow e_2 \oplus ...$

Figure 4.8.: FP disposal (sliding window).

replacing that of Figure 4.1. In this case, if the above relation is delivered, then $Q[T_1]$ will contain $\{e_1\}$ and $Q[T_2]$ will contain $\{e_a\}$.

Lastly, in what we call *first prefix* (FP) disposal, every event in each type queue which appears *before* the *first* matched event is discarded along with the very first matched event. As will be shown, this type of disposal is tailored to sliding windows. FP disposal is shown in Figure 4.8. Here, if the above relation is delivered, then $Q[T_1]$ will contain $\{e_3, e_4, e_5\}$ and $Q[T_2]$ will contain $\{e_c, e_d\}$.

### 4.3.3 Windows

Much like in stream processing, along with reasoning about the above in terms of matching and disposal semantics, events can be grouped together and discarded according to the current "window" in which events may be matched. If $T_k[i]$ is the largest $i$ for type $T_k$ occurring in a predicate, then the subscription involves a window of size $i$. A window may be viewed as moving forward as time progresses, as events are received or as events are delivered, allowing a certain number, or subset, of events to be considered for matching at any one instance. When the window has passed events, these events may be discarded, while only events within the window may be considered for matching.

*Tumbling windows* consider a number of events, and when the window is to move forward, it "tumbles" to the next set of events in the queue, which is a completely new set, i.e., no events are considered more than once. In FAIDECS, the disposal semantics (i.e., PI disposal) equate to that of a tumbling window: The window starts as a single event per type, and events are added to the window when a match is not found. After a match is found, the window tumbles over to the immediate next set of events in the respective queues, which have not yet been considered.

*Sliding windows* are common in stream processing. Most commonly, a sliding window considers a fixed number of events, and moves forward by one event at a time as it progresses. Within a window, events may be matched so that they are contiguous, i.e., if more than one event is used from the same window for a single operation, each event must have been immediately received after the previous in the set. In other variations, events may be matched that are non-contiguous, as long as they are each a part of the same window. Sliding windows allow for a same message to be matched more than once in multiple relations, which immediately violates the MDM No Duplication property given above. Another variation of the property could allow for a single event to be delivered more than once, but never in the same position within two different relations, for instance. A variation of the property which allows for sliding windows is as follows.

MDM No Duplication′ $\exists\, \text{DLVR}^i_\Phi([e_1, ..., e_n])_t \;\Rightarrow \nexists \text{DLVR}^i_\Phi([e'_1, ..., e'_n])_{t'} \mid e_j = e'_j \wedge t' \neq t$

In the case for correlation, sliding windows might be implemented slightly differently. Firstly, as in the FAIDECS algorithm, there could be one window per type. And instead of moving a window one event per round when an event is received, a window might start at the beginning of a match for each type, and then once the corresponding relation is delivered, move each window per type queue by one event. This would assure that no event is delivered twice in the same position of a window, thus ensuring MDM No Duplication′. The above described sliding window is equivalent to FP disposal found in Figure 4.8.

### 4.3.4 Properties of Semantic Options

In this and the following section, we discuss the unmet properties by comparing matching semantics and disposal semantics. Table 4.1 enumerates the properties violated for various combinations of matching and disposal semantics.

Table 4.1: Table of semantic options specifying which properties are not met with applicable theorems in parentheses. Shaded area indicates default semantics for FAIDECS.

| | FR matching | FRC matching | MR matching | MRC matching |
|---|---|---|---|---|
| I disposal | Type Total Order (6)<br>Type FIFO Order (6)<br>Type Causal Order (6) | Event Validity (5)<br>Conjunction Validity (5)<br>Type Total Order (6)<br>Type FIFO Order (6)<br>Type Causal Order (6) | Event Validity (1)<br>Covering Agreement (2)<br>Type Total Order (6)<br>Type FIFO Order (3, 6)<br>Type Causal Order (4, 6) | Event Validity (1, 5)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type Total Order (6)<br>Type FIFO Order (3, 6)<br>Type Causal Order (4, 6) |
| PI disposal (Tumbl. w.) | (All properties met as shown previously [82]) | Event Validity (5)<br>Conjunction Validity (5) | Event Validity (1)<br>Covering Agreement (2) | Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2) |
| IP disposal | Event Validity (7)<br>Covering Agreement (8) | Event Validity (5, 7)<br>Conjunction Validity (5)<br>Covering Agreement (8) | Event Validity (1)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) | Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) |
| FP disposal (Sliding w.) | No Duplication (9)<br>Type FIFO Order (10)<br>Type Causal Order (10) | No Duplication (9)<br>Event Validity (5)<br>Conjunction Validity (5) | No Duplication (9)<br>Event Validity (1)<br>Covering Agreement (2)<br>Type FIFO Order (3, 10)<br>Type Causal Order (4, 10) | No Duplication (9)<br>Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) |

#### First Received vs. Most-Recently Received

Since FAIDECS uses non-contiguous FR matching (with PI disposal) and all of the above properties are met (as shown in the shaded box of Table 4.1), it is clear that taken by itself, FR matching does not violate any properties. Only when non-contiguous FR matching is paired with different disposal semantics are any properties violated. On the contrary, with MR matching, there is *no* combination with disposal semantics that *does not* violate some properties. Particularly, MR matching always violates MDM Event Validity and MDM Covering Agreement. Further, aside from using PI disposal, MR matching violates MDM Type FIFO Order and MDM Type Causal Order.

**Most-Recently Received Matching** The following Theorems 1–4 prove that MR matching violates several properties.

**Theorem 1** *MR matching violates* MDM *Event Validity*

**Proof** This proof will be by counter-example. Suppose a process $p_i$ has a subscription over three event types $T_1$, $T_2$ and $T_3$ such that $\Phi = T_1.a_1 = v \wedge T_2.a_1 < T_3.a_1$. Now suppose that an event $e_1^1$ such that $e_1^1.a_1 = v$ is received, thus qualifying as an event to which MDM Event Validity applies. However, due to the lack of other matching events of type $T_2$ and $T_3$, this event $e_1^1$ is not delivered as part of a relation. As more events are received, it is possible that more events of type $T_1$ that match the respective unary predicate are received than may be delivered before matching events of types $T_2$ and $T_3$ are received. As matching events of types $T_2$ and $T_3$ are received, they are then matched with the newer events of type $T_1$. In this case, $e_1^1$ is essentially "buried" and is never again viewed for another possible match since the newer events only are considered, thus MDM Event Validity may be violated. ■

As an example, consider the following subscription for some arbitrary value of the US dollar of 1: $\Phi_S = \texttt{SQuote}[0].\texttt{time} > \texttt{EReport}[0].\texttt{time} \wedge \texttt{SQuote}[1].\texttt{value} > \texttt{SQuote}[0].\texttt{value} \wedge \texttt{SQuote}[2].\texttt{value} > \texttt{SQuote}[1].\texttt{value} \wedge \texttt{USDollar.value < 1}$ In this case, it is possible that an event of type `USDollar` could be received with value 0.74 and then placed in the buffer to await a match with three successive stock quotes of increasing value. However, there may be many events received of type `SQuote` (which are not successively increasing) along with many other events of type `USDollar` before the first three conditions are met. Once three successively increasing events of type `SQuote` are received, there may be a large number of events in the buffer of type `USDollar` that are less than 1 which qualify first to be matched with the stock quote events since the most recent events are desired here. If more events are being received than there are relations being delivered, since MR is used, the first received event of type `USDollar` with value 0.74 may never be used in a match, and thus MDM Event Validity is not met.

**Theorem 2** *MR matching violates* MDM *Covering Agreement*

**Proof** The following proof is by counter-example. Suppose a process $p_i$ has a conjunction $\Phi_i = T_1.a_1 < T_2.a_1$ and another process $p_j$ has a conjunction $\Phi_j = \Phi_i \wedge T_3.a_1 < z$. In this example, now suppose that both $p_i$ and $p_j$ receive two events $e_1^1$ and $e_1^2$ such that $e_1^1.a_1 = v$ and $e_1^2.a_1 = v'$ (s.t. $v < v'$). In this case, both $e_1^1$ and $e_1^2$ match $\Phi_i$, thus process $p_i$ may deliver the relation $[e_1^1, e_1^2]$. However, process $p_j$ must wait for a matching event of type $T_3$ before it may deliver any relations. Now suppose that both $p_i$ and $p_j$ receive a third message $e_2^2$ such that $e_2^2.a_1 = u$ (s.t. $u > v'$). Now, process $p_j$ could receive an event $e_1^3$ such that $e_1^3.a_1 = w$ (s.t. $w < z$). When process $p_j$ triggers a match, it will view the most recent events by the most-recently received matching function, and thus the relation $[e_1^1, e_2^2, e_1^3]$ is delivered which violates MDM Covering Agreement since process $p_i$ matched $e_1^1$ with $e_1^2$ but process $p_j$ matched $e_1^1$ with $e_2^2$. ■

When PI disposal is not used, MR matching may also violate a number of ordering properties, namely MDM Type FIFO Order and MDM Type Causal Order.

As an example, consider the subscription $\Phi_S$ above except that one process is only looking for three successive stock quotes, and the second process has the same constraints but also has the last constrain above where `USDollar.value < 1`. In this case, it is possible that three successive stock quotes are published and the first process would deliver them. However, if the second process has not received any events of type `USDollar` with value less than 1, no relations may be delivered by the second process. If a fourth successive stock quote is received, followed by an event of type `USDollar` with value 0.89, then the *last* three stock quote events received (by MR) will be matched with this new `USDollar` event. Thus, the two processes will not agree on the three stock quote events that are delivered since the first process delivered the *first* three stock quote events received, while the second delivered the last three of four received.

**Theorem 3** *MR matching with the absence of PI disposal violates* MDM *Type FIFO Order*

**Proof** Since events are matched backwards in the queue, it is clear that if some later message $e_j^k$ is matched and delivered in a relation before an earlier event $e_i^k$ such that $i < j$, and some event other than the type $T(e_j^k)$, say $e_l^m$, is then later received, the earlier event $e_i^k$ such that $i < j$ in the same queue as $e_j^k$ might be matched with $e_l^m$ thus violating MDM Type FIFO Order since $e_j^k$ is delivered *before* $e_i^k$ and $i < j$. ■

Consider a more simple subscription: $\Phi_T = \texttt{SQuote}[0].\texttt{time} > \texttt{EReport}[0].\texttt{time} \wedge \texttt{SQuote}[0] \wedge$ `USDollar.value < 1` which is looking for any stock quote after an earnings report when the US dollar drops below the value 1. In this case, if two events of type `USDollar` are received, both with values less than 1, before any stock quotes are received after an earnings report, it will be the case by MR that the *second* `USDollar` event will eventually be delivered first. Without PI disposal, the first `USDollar` event remains in the queue and can later be delivered, violating MDM Type FIFO Order.

**Theorem 4** *MR matching with the absence of PI disposal violates* MDM *Type Causal Order*

**Proof** Without FIFO order, there cannot be causal order in this instance. Thus, it follows by Theorem 3 that MDM Type Causal Order is violated. ■

PI disposal rectifies the issues in Theorems 3–4 since if $e_j^k$ were delivered, the event $e_i^k$ such that $i < j$ would be thus discarded and never delivered and FIFO order would still hold.

The reason why MDM Type Total Order is violated for MR matching with I disposal will be explained shortly when comparing disposal semantics in this setting.

### Contiguous vs. Non-contiguous Matching

In addition to FR and MR matching, the added constraint that matched events must be contiguous may cause the violation of validity.

**Theorem 5** *Contiguous matching violates* MDM *Event Validity and* MDM *Conjunction Validity*

**Proof** This proof will be by counter example. Suppose that a process $p_i$ has a subscription on a type $T_1$ such that $\Phi = T_1[1].a_1 = v \wedge T_1[2].a_1 = v$, which attempts to match two events from the same stream each with a first attribute with a value of $v$. In this scenario, it is possible that no two consecutive events have the same value for the first attribute. Suppose that a process sends events such that $a_1$ alternates between values $v$ and some $v'$ such that $v' \neq v$. Thus, with contiguous matching, no two consecutive events have the value $v$, whereas with non-contiguous matching, a match is possible by considering every other event. Thus both MDM Event Validity and MDM Conjunction Validity are violated. ■

Consider a simple subscription: $\Phi_U = \texttt{SQuote}[0].\texttt{value} = 3.44 \wedge \texttt{SQuote}[1].\texttt{value} = 3.44$ which looks for two stock quote events to have the same value. It is easy to see that if published events of type `SQuote` are consistently alternating between different values between any two events, then with the requirement of contiguous matching, there would never be a match for $\Phi_U$, thus violating MDM Event Validity and MDM Conjunction Validity since no events would ever be delivered. This could be solved by matching stock quote events in a non-contiguous manner.

Infix vs. Prefix+Infix vs. Infix+Postfix Event Consumption

Taken by itself, PI disposal does not violate any properties as shown by the left middle portion of Table 4.1. The properties violated with PI disposal together with MR matching are due to the matching semantics as shown in Section 4.3.4. In contrast though, I and IP disposal cause the violation of a number of properties.

**Properties of Infix Only Event Consumption** I disposal causes the violation of the properties MDM Type Total Order, MDM Type FIFO Order and MDM Type Causal Order. This is due to the fact of how different events may be correlated over time. The following theorem demonstrates why I disposal can violate all three properties simultaneously.

**Theorem 6** *I disposal violates* MDM *Type Total Order,* MDM *Type FIFO Order and* MDM *Type Causal Order*

**Proof** By counter example, consider two processes $p_i$ and $p_j$ such that their subscriptions are $\Phi_i = T_1.a_1 < T_2.a_1$ and $\Phi_j = T_1.a_1 > T_3.a_1$ respectively. Now suppose that both $p_i$ and $p_j$ (starting with empty queues) receive the event $e_1^1$ of type $T_1$ such that $e_1^1.a_1 = v$. Since this is the only event which either process has received, then both will queue $e_1^1$ for later matching. Now, suppose that $p_j$ receives the event $e_1^3$ of type $T_3$ such that $e_1^3.a_1 = w$ (s.t. $v > w$). Now, process $p_j$ may trigger a match and deliver the relation $[e_1^1, e_1^3]$. This match would be triggered by either MR or FR matching. Next, suppose that both $p_i$ and $p_j$ receive another event $e_2^1$ of type $T_1$ such that $e_2^1.a_1 = v'$ and then $p_j$ receives an event $e_2^3$ of type $T_3$ such that $e_2^3.a_1 = w'$ (s.t. $v' > w'$). The process $p_j$ may now trigger another match and deliver the relation $[e_2^1, e_2^3]$, which may be matched by either MR or FR matching. Since $p_i$ has not yet received any events of type $T_2$, it may not yet deliver any relations.

Note, at this point, no properties have yet been violated. Now suppose that process $p_i$ receives an event $e_1^2$ of type $T_2$ such that $e_1^2.a_1 = u$ (s.t. $v' < u < v$). By either MR or FR matching, when $p_i$ attempts to trigger a match, $e_1^2$ will only match with $e_2^1$ and thus $p_i$ delivers the relation $[e_2^1, e_1^2]$. By I disposal, $p_i$ discards only the events delivered, and thus the event $e_1^1$ remains in $p_i$'s queue. Again, at this moment, no properties have yet been violated. But if $p_i$ were to now receive an event $e_2^2$ of type $T_2$ such that $e_2^2.a_1 = u'$ (s.t. $v < u'$), this event may be matched by $p_i$ with $e_1^1$ and $p_i$ would thus deliver the relation $[e_1^1, e_2^2]$.

MDM Type Total Order has been violated since both $p_i$ and $p_j$ have different conjunctions, but receive events over a common type, i.e., $T_1$. Since $p_j$ delivers $e_1^1$ *before* $e_2^1$ within separate relations, but $p_i$ delivers $e_2^1$ before $e_1^1$ within separate relations, this total order over the events of the same common type $T_1$ is thus violated.

The above also shows the violation of MDM Type FIFO Order since the event $e_1^1$ was clearly sent before $e_2^1$ (it may be assumed that both were sent by the same process for the sake of argument), but $p_i$ delivered those events in a conflicting order. Lastly, since causal order requires FIFO order and FIFO order has here been violated, it follows that MDM Type Causal Order may also be violated. ■

As an example, consider the two subscriptions (by processes $p_1$ and $p_2$ respectively) that resemble the subscriptions in the counter example above:

$$\Phi_{V_1} = \texttt{SQuote}[0].\texttt{value} < \texttt{Euro}[0].\texttt{value}$$

$$\Phi_{V_2} = \texttt{SQuote}[0].\texttt{value} < \texttt{USDollar}[0].\texttt{value}$$

As in the counter example, consider the following events are received in the following order (where `Type`($v$) represents receiving an event of type `Type` with value $v$): {`SQuote`(3), `USDollar`(1), `SQuote`(2), `USDollar`(0.98)} In this case, $p_2$ may deliver the relations [`SQuote`(3), `USDollar`(1)] and [`SQuote`(2), `USDollar`(0.98)] by $\Phi_{V_2}$ but $p_1$ has not yet received any events of type `Euro` yet, so both `SQuote`(3) and `SQuote`(2) are queued in that order. Now supposed the event `Euro`(2.5) is now received by $p_1$. The only possible relation that may be delivered by $p_1$ (using any matching semantics for $\Phi_{V_1}$) is thus [`SQuote`(2), `Euro`(2.5)] while the event `SQuote`(3) remains in $p_1$'s queue due to I disposal. However, if the event `Euro`(3.1) were then received by $p_1$, the relation [`SQuote`(3), `Euro`(3.1)] may then be delivered by $p_1$. In this case, it is clear that MDM Type FIFO Order is violated since `Euro`(2.5) was sent and received before `Euro`(3.1) which also shows that MDM Type Total Order is violated over the type `SQuote` since $p_1$ and $p_2$ delivered the events of type `SQuote` in differing orders. Since MDM Type FIFO Order is violated, MDM Type Causal Order is thus also violated.

**Properties of Infix+Postfix Event Consumption** Section 4.3.4 discussed violation of MDM Event Validity and MDM Covering Agreement by IP disposal with MR matching. These properties remain to be investigated in the context of FR matching.

**Theorem 7** *FR matching with IP disposal violates* MDM *Event Validity*

**Proof** By counter example, consider a process $p_i$ that has a subscription such that all predicates over a type $T_x$ are unary predicates, i.e., only comparing attributes of the type to scalar values. If $p_i$ were to start with an empty set of queues, and immediately received two events of type $T_x$ that both meet all the unary predicates over $T_x$ and then received other events which completed a match, the first event of type $T_x$ would be matched in a relation with the other received events and then the second would be discarded by IP disposal, thus violating MDM Event Validity. ∎

As an example, consider the subscription:

$\Phi_W \;=\; \texttt{SQuote}[0].\texttt{value} < 3 \wedge \texttt{USDollar}[0].\texttt{value} < 1$

If a process with the subscription $\Phi_W$ receives the two events `SQuote`(2.5) and `SQuote`(2) respectively, no relations may yet be delivered. However, if an event `USDollar`(0.98) were received, then by FR matching, the relation [`SQuote`(2.5), `USDollar`(0.98)] may then be delivered. By using IP disposal, then the event `SQuote`(2) is discarded, which violates MDM Event Validity.

**Theorem 8** *FR matching with IP disposal violates* MDM *Covering Agreement*

**Proof** Consider, again by counter-example, two processes $p_i$ and $p_j$ with subscriptions $\Phi_i = T_1.a_1 < T_2.a_1$ and $\Phi_j = \Phi_i \wedge T_3.a_1 < v$ respectively. If both $p_i$ and $p_j$, starting with empty queues, receive two events $e_1^1$ and $e_1^2$ such that $e_1^1.a_1 = u$ and $e_1^2.a_1 = u'$ (s.t. $u < u'$), then $p_i$ may deliver the relation $[e_1^1, e_1^2]$ whereas $p_j$ must wait for a matching event of type $T_3$. Next, if both $p_i$ and $p_j$ were to receive two more events $e_2^1$ and $e_2^2$ such that $e_2^1.a_1 = u''$ and $e_2^2.a_1 = u'''$ (s.t. $u'' < u'''$), again $p_i$ may deliver another relation $[e_2^1, e_2^2]$, but $p_j$ must wait for a matching event of type $T_3$. Lastly, if $p_j$ were to then receive an event $e_1^3$ such that $e_1^3.a_1 = w$ (s.t. $w < v$), using FR matching, $p_j$ may now perform a match and deliver the relation $[e_1^1, e_1^2, e_1^3]$. However, due to IP disposal, $p_j$ will discard both $e_2^1$ and $e_2^2$ since these are in the same type queues as the delivered events of types $T_1$ and $T_2$. Thus, MDM Covering Agreement is violated since $p_i$ delivered the two events $e_2^1$ and $e_2^2$ whereas $p_j$ discarded them. ∎

Consider the subscriptions for processes $p_1$ and $p_2$ respectively:

$\Phi_{X_1} \;=\; \texttt{SQuote}[0].\texttt{value} < \texttt{USDollar}[0].\texttt{value}$

$\Phi_{X_2} \;=\; \texttt{SQuote}[0].\texttt{value} < \texttt{USDollar}[0].\texttt{value} \wedge \texttt{USDollar}[0].\texttt{value} < 1.9$

If both $p_1$ and $p_2$ receive the events `SQuote`(1) and `USDollar`(1.1) respectively, then only $p_1$ can deliver these events as a relation, whereas $p_2$ must place them in a queue to await an event of type `Euro`. If now, both processes receive the two events `SQuote`(1.2) and `USDollar`(1.3) respectively, then again, $p_1$ may deliver these events in a relation but $p_2$ cannot since there is still yet no event of type `Euro` with which to match these received events (i.e., the queue for the type `Euro` for $p_2$ is empty). If $p_2$ were to then receive the event `Euro`(1.8), then $p_2$ may now perform a match on $\Phi_{X_2}$. By FR matching, $p_2$ will deliver the relation [`SQuote`(1), `USDollar`(1.1), `USDollar`(1.8)]. However, by IP disposal, $p_2$ will then discard the events `SQuote`(1.2) and `USDollar`(1.3) from its queue. Since $p_2$ will thus never deliver these discarded events, $p_1$ and $p_2$ will not agree on all sub-relations delivered over the types `SQuote` and `USDollar`, and thus MDM Covering Agreement is violated.

Tumbling vs. Sliding Windows

Windows in this context are equivalent to replacing the disposal semantics. In particular, note that tumbling windows are equivalent to using PI disposal. Thus, what remains to be discussed is the topic of sliding windows. Sliding windows may be implemented as either a *contiguous* sliding window, i.e., all events matched within the same window must be contiguous, or a *non-contiguous* sliding window where as long as all events matched are currently in the window, they need not be contiguous. Note that for contiguous sliding windows, the only additional property that is unmet, aside from those by the matching semantics, is MDM No Duplication; however, MDM No Duplication$'$ may still be met. Non-contiguous sliding windows also violate MDM No Duplication while maintaining MDM No Duplication$'$.

**Properties of Sliding Windows** Sliding windows can violate MDM No Duplication as shown below in demonstrating that FP violates this property.

**Theorem 9** *FP disposal violates* MDM *No Duplication.*

**Proof** By counter-example, suppose that a process $p_i$ has a conjunction $\Phi = T_1[1].a_1 < T_1[2].a_1$. Now, suppose that $p_i$ receives three events $e_1^1$, $e_2^1$ and $e_3^1$ of type $T_1$ such that $e_1^1.a_1 = v$, $e_2^1.a_1 = v'$ and $e_3^1.a_1 = v''$ (s.t. $v < v' < v''$). Process $p_i$ will first deliver the relation (here by FR) $\{e_1^1, e_2^1\}$ and then discard $e_1^1$ by FP disposal. Process $p_i$ will then deliver the relation $\{e_2^1, e_3^1\}$ also by FR. Since $e_2^1$ is delivered within more than one relation, MDM No Duplication is violated. This same argument holds for MR matching, with the relations delivered thus being ■

As a quick example, consider the subscription $\Phi_Y = \texttt{SQuote}[0].\texttt{value} < \texttt{SQuote}[1].\texttt{value} \wedge \texttt{USDollar}[0].\texttt{value} < 1.1$ Suppose that a process with subscription $\Phi_Y$ received (in the following order) the events `SQuote`(1), `SQuote`(1.1) and `SQuote`(1.2) before receiving any events of type `USDollar`, thus no relations are yet able to be delivered. If an event `USDollar`(1) were then received, then the process will (by FR matching) deliver the relation [`SQuote`(1), `SQuote`(1.1), `USDollar`(1)] and by FP disposal, the only events to be discarded from each of the queues respectively are `SQuote`(1) and `USDollar`(1). This leaves the queue for the `SQuote` with the two events `SQuote`(1.1) (which has been delivered as part of a relation), and `SQuote`(1.2) respectfully with the queue for `USDollar` now empty. Lastly, if an event `USDollar`(0.99) were received, then another relation may be delivered. By FR matching, the relation delivered is thus [`SQuote`(1.1), `SQuote`(1.2), `USDollar`(0.99)] with the only events discarded are thus `SQuote`(1.1) and `USDollar`(0.99) (by FP, leaving `SQuote`(1.2) in the queue. Since between the two relations [`SQuote`(1), `SQuote`(1.1), `USDollar`(1)] and [`SQuote`(1.1), `SQuote`(1.2), `USDollar`(0.99)], one can see that `SQuote`(1.1) was delivered in two separate relations, which thus violates MDM No Duplication.

Note that in the proof above, since $e_2^1$ (in the counter-example) is delivered in different positions between the two relations, MDM No Duplication$'$ is retained in both the proof counter-example and the provided example following the proof.

Sliding windows with non-contiguous matching may cause the violation of a number of the ordering properties. The following theorem demonstrates how non-contiguous matching with a sliding window via FP disposal may cause the violation of the properties MDM Type Total order, MDM Type FIFO Order, and MDM Type Causal Order.

**Theorem 10** *Both (non-contiguous) FR and MR matching with FP disposal violate* MDM *Type FIFO Order, and* MDM *Type Causal Order*

**Proof** The following is by counter-example. Consider a subscription by process $p_i$, $\Phi = T_1[1].a_1 = T_1[2].a_1$ where this subscription denotes that an attribute of some event of type $T_1$ must be equal to the same attribute of a later received event. Again, in this semantic, these are not guaranteed to be contiguous events. Consider if a current queue for process $p_i$ were $[e_1^1, e_2^1, e_3^1, e_4^1]$ where $a_1$ for each of these events are respectively $[v, v', v, v']$. Consider FR matching. The first relation to be delivered would thus be $\{e_1^1, e_3^1\}$. By FP disposal, the event $e_1^1$ would be discarded. The next delivered relation would then be $\{e_2^1, e_4^1\}$. In this case, since $e_2^1$ is delivered *after* $e_3^1$, the aforementioned ordering properties are thus violated. This same argument can be used for MR matching. ■

Consider a subscription $\Phi_Z = \texttt{SQuote}[0].\texttt{value} = \texttt{SQuote}[1].\texttt{value}$. Suppose that the incoming quotes alternated values, such that a process with subscription $\Phi_Z$ receives the following events in order: `SQuote`(1.1), `SQuote`(1.2) and `SQuote`(1.1). By the subscription, the process would deliver the relation [`SQuote`(1.1), `SQuote`(1.1)] (the first and third received events). If using FP disposal, then only the first event would be discarded, with the resulting queue being {`SQuote`(1.2), `SQuote`(1.1)}. Now suppose a fourth event `SQuote`(1.2) were received. The relation [`SQuote`(1.2), `SQuote`(1.2)] may then be delivered. However, in this case, since the first instance of `SQuote`(1.2) was received *before* the second instance of `SQuote`(1.1), but they were delivered such that the second instance of `SQuote`(1.1) was delivered first, then MDM Type FIFO Order is violated.

## 4.4 Case Studies

In this section we investigate the properties obtained when substituting previously proposed correlation engines/languages in the FAIDECS overlay network. We investigate how their constructs relate to the properties previously introduced by mapping them to the semantic options discussed. Tables 4.2 and 4.3 summarize our findings.

Table 4.2: Basic safety as well as liveness properties violated by various language operators.

| | Basic Safety | | | | Liveness | |
|---|---|---|---|---|---|---|
| | No Duplication | No Duplication′ | No Creation | Admission | Conjunction Validity | Event Validity |
| TESLA | `each-within` | - | - | - | - | `first-within` `last-within` `not` |
| StreamSQL | `select` | - | - | - | `select` `create window` | `select` |
| EQL | `select` | - | - | - | `select` `create window` | `select` `limit` |
| CEL | `select` | - | - | - | `select` | `select` |

Table 4.3: Agreement and ordering (safety) properties violated by various language operators.

| | Agreement | | Order | | | |
|---|---|---|---|---|---|---|
| | Conjunction Agreement | Covering Agreement | TYPE TOTAL ORDER | CONJUNCTION TOTAL ORDER | FIFO ORDER | CAUSAL ORDER |
| TESLA | - | `first-within` `last-within` | `each-within` `consuming` | - | `each-within` `consuming` | `each-within` `consuming` |
| StreamSQL | - | `create window` | `select`, `union`, `merge` | - | `select`, `union` `create window` `merge` | `select`, `union` `create window` `merge` |
| EQL | - | `create window` | `select`, `union`, `merge` | - | `select`, `union` `create window` `merge` | `select`, `union` `create window` `merge` |
| CEL | - | - | `select` | - | `select` | `select` |

### 4.4.1 The TESLA Language

TESLA [21], a complex event specification language, provides a high degree of expressiveness and flexibility for event subscriptions with an intuitive and simple syntax. In particular, the operators that TESLA provides are operators for event occurrence, event composition, parameterization, timers, negation, event consumption, aggregates, event hierarchies and iterations. The following represents a general TESLA query.

```
define subscription([att1:type1, ..., attn:typen])
  from event_source ([pattern]) [and interval_operation]
  [where predicate] [consuming event_identifiers]
```

Replacing the matching logic (i.e., the matching and disposal semantics) of FAIDECS with that of TESLA would thus allow for a much more expressive event correlation system. The following describes each of the operators of TESLA, and how the addition of each to FAIDECS affects the respective properties.

#### Event Occurrence/Selection

TESLA allows for a simple subscription, specifying constraints over singleton events in both content and time. Because the properties of FAIDECS may be simplified for single event delivery, all aforementioned properties still hold for these operators. The following is in the syntax of TESLA using the semantics of FAIDECS to denote events and their types and attributes:

```
define Subscription1() from SQuote (SQuote.val > 10)
```

The equivalent subscription in FAIDECS is $\Phi = \texttt{SQuote}[0].\texttt{val} > 10$.

#### Event Composition

Event correlation is possible in TESLA through event composition operators. TESLA provides three variants with specific matching and disposal rules associate with each. They are, respectively, **each-within**, **first-within** and **last-within**. The idea regarding these operators is that in specifying an event composition, it is possible that a single event could be matched with one or more events to make composite events or relations. In particular, when events are to be matched within a certain time interval of the occurrence of some singular event, these operators specify *precisely* how the single event is to be correlated with the others.

The **each-within** operator provides the most composite events. This operator is equivalent to FR matching with I disposal of Table 4.1. An example subscription in TESLA follows:

```
define Subscription2() from EReport() and
  each SQuote(SQuote.val > 10) within 5min from EReport
```

In this subscription, any occurrence of an event of type `EReport` would be saved for five minutes to be matched with any of type `SQuote` where `SQuote.val` $> 10$. For events of type `EReport`, the property MDM No Duplication is not met, but as discussed for windows, MDM No Duplication′ may be met instead. For events of type `SQuote`, all events are matched in a FR order and I disposal applies. Thus, by Table 4.1, the properties MDM Type Total Order, MDM Type FIFO Order and MDM Type Causal Order are violated. Thus, if an event $e^{ER}$ of type `EReport` were received, any and all events of type `SQuote` for which `SQuote.val` $> 10$, received within five minutes after having received $e^{ER}$, will be delivered in separate relations with $e^{ER}$.

The **first-within** operator only allows for a single composite event or relation to be delivered for a given subscription within the specified time interval. In the above subscription of Section 4.4.1, if replacing the keyword **each** with **first**, then of all events received of type $T_2$ within five minutes of receiving an event $e^1$ of type $T_1$, only the *first* event for which $e^2.a > 10$ will be delivered.

Depending on *when* the matching is triggered, in the worst case, the **first-within** operator is equivalent to FR matching with PI disposal for all events of type $T_2$. Thus, by Table 4.1, the properties that are violated are MDM Event Validity and MDM Covering Agreement.

The `last-within` operator, similar to `first-within`, allows only for a single composite event or relation to be delivered within a specific time interval. By replacing `each` with `last` in the example subscription in Section 4.4.1, then of all events received of type $T_2$ within five minutes of receiving an event $e^1$ of type $T_1$, only the *last* event for which $e^2.a > 10$ will be delivered.

The properties which are violated again depend on when the matching is triggered, but in the worst case, the `last-within` operator is equivalent to most-recently received matching with PI disposal. By Table 4.1, this operator thus violates MDM Event Validity and MDM Covering Agreement.

Parameterization

Parameterization in the context of TESLA is the composition of events when related by some higher order function such as `area`. An example of a parameterized subscription, in English, could thus be: *Warn of an avalanche when 3 or more sensors detect movement when these sensors are within the same area $x.* Where *area $x* can be specified as a parameter in the subscription. In this case, location, or whatever other parameters, may be included within events as further attributes, which equates to nothing more than further constraints on attributes of events. Parameterization does not cause the violation of any of the above properties.

Timers

The TESLA language allows for events to be matched using timers. An example would be to attempt to trigger a match every morning at 10 a.m. over all received events since the last time the matching was triggered. Because this type of matching can use any matching and disposal semantics, this operator will not suffer further violations of properties aside from any that may be violated by the matching and disposal semantics themselves.

Note that the use of timers assumes at least a partially synchronous system, however, which is opposed to the assumptions of FAIDECS. However, specialized solutions do exist, which deal with such cases and are out of the scope of this paper.

### Negations

The negation operator allows the control of when certain composite events should **`not`** be matched. Since this operator only specifies that certain events should not be delivered, only MDM Event Validity may be violated in this case.

### Aggregates

Operators such as **`min`**, **`max`**, **`average`**, **`sum`**, etc., are examples of aggregate operators. Aggregate operators take more than one event from a particular queue and yield a single result. This is equivalent to consuming events in streams using a tumbling window, thus aggregates do not violate any properties.

### Event Consumption

TESLA provides the expressiveness to specify which events should be consumed or discarded. Consider the following example:

```
define Subscription2() from EReport() and
  each SQuote(SQuote.val > 10) within 5min from EReport
  consuming SQuote
```

This subscription provides a specific disposal policy for all events of type `SQuote`. To avoid the scenario where the same events of type `SQuote` may be matched with multiple events of type `EReport`, the **`consuming`** keyword specifies that any events of that type may only be matched once, and then discarded such that any new events of type `EReport` must be matched with new events of type `SQuote`. This is equivalent to I disposal. Thus, the properties which may be violated will be the intersection between those violated by the composition operators (as specified in Section 4.4.1), and then the resulting matching semantics of those operators together with I disposal. The unmet properties are thus shown in Table 4.1.

Event Hierarchies

Certain single events may be matched together to form a composite event or relation, which may be matched together to form further, more complex composite events comprised of simple events and complex events. These subscriptions require levels (i.e., hierarchies) of correlation. Hierarchies allow for more expressiveness while meeting all properties.

Iterations

Iterations specify constraints over a set of events of the same type over time. An example would be to "capture" every iteration of events of type $T_1$ such that the attribute $a_1$ never decreases. Due to TESLA's ability to define hierarchies of events and the ability to specify different selection and consumption policies for different rules, no further operators are needed to allow for iterations. In this case, again, when an iteration is specified, the violated properties are thus the intersection of the violated properties of the selection and consumption policies.

### 4.4.2 The StreamSQL Language

StreamSQL [49] is a stream processing and querying language that extends SQL with the ability to define and manipulate real time data streams. While SQL is primarily intended for manipulating traditional database tables, which are finite bags of tuples (rows), StreamSQL adds the ability to manipulate streams, which are infinite sequences of tuples that are not all available at the same time. Because streams are infinite, operations over streams must be monotonic. Queries over streams are generally "continuous," executing for long periods of time and returning incremental results. The StreamSQL language is typically used in the context of a Data Stream Management System (DSMS) for applications including algorithmic trading, market data analytics, network monitoring, surveillance, e-fraud detection and prevention, clickstream analytics and real-time compliance (anti-money laundering).

#### Overview

Like SQL, StreamSQL consists of a DDL (Data Description Language) and a DML (Data Manipulation Language). The DDL is straightforward – the schema of a stream is the same as that of a table – a stream consists of a tuple of typed fields. Several new operations are introduced in the DML to manipulate streams – (1) Selecting from a stream – A standard `select` statement can be issued against a stream to calculate functions (using the target list) or filter out unwanted tuples (using a `where` clause). The result will be a new stream. (2) Stream-Relation Join – A stream can be joined with a relation to produce a new stream. Each tuple on the stream is joined with the current value of the relation based on a predicate to produce zero or more tuples. (3) Union and Merge – Two or more streams can be combined by unioning or merging them. Unioning combines tuples in strict FIFO order. Merging is more deterministic, combining streams according to a sort key. (4) Windowing and Aggregation – A stream can be windowed to create finite sets of tuples. For example, a window of size 5 minutes would contain all the tuples in a given 5 minute period. Window definitions can allow complex selections of messages, based on tuple field values. Once a finite batch of tuples is created, analytics such as count, average, max, etc., can be applied. (5) Windowing and Joining – A pair of streams can also be windowed and then joined together. Tuples within the join windows will combine to create resulting tuples if they fulfill the predicate.

### Selection

**select** is used to retrieve events from an unwindowed stream, one or two windowed streams, a materialized window, or a table. A **select** statement includes required subclauses. There are two forms of the **from** clause: the first identifies the streams, materialized window or table from which the events are extracted. The optional **where** subclause adds additional restrictions to the **select** result, such as a range of values or a limit to the number of events received. A **select** statement can also include nested **select** statements (also called subqueries).

```
select target_entry1, ..., target_entryn
  from event_source1, ..., event_sourcen
  within (interval_time | value_on_field)
  [where predicate]             [having predicate]
  [group by field_identifier] [order by field_identifier]
  [into stream_identifier]
```

An example query is below where the query looks for stocks where the price are greater than some analyst report's price for that same stock. The two events should have been received within 2 minutes of each other.

```
select stockquote.id, analystreport.id
  from stockquote, analystreport
  within 120
  where stockquote.firm = analystreport.firm
    and stockquote.price > analystreport.price
  group by analyst report.id
```

A target_entry is a rule that expands to a value that will be included in each row of the result set, and this can be used to select different sets of fields from each event source. An entry can be extracted from an event present on a stream, from an event in a materialized window, from a row in a table, or from the return from a StreamBase function or expression. The `group by` and `order by` clauses are similar to SQL. The `select` clause uses I disposal of events, i.e., unmatched events remain in the queue to be matched to subsequent `select` or other join statements. Hence, the `select` statement causes a violation of the properties MDM Type Total Order, MDM Type Total Order and MDM Type Causal Order.

Windowing

A window specification describes how a stream of tuples will be subdivided prior to analysis through an aggregate stream query or used within a tuple join statement. In a StreamSQL application, windows are used within an aggregate stream or tuple join statement. The window specification may be entered as a separate statement or included within the aggregate stream or tuple join statement. The advantage of writing the specification as a separate statement is that the definition may be reused in multiple aggregate stream or tuple join statements.

```
create window window_identifier (
  size natural advance increment
  {time | tuples |on field_identifier_w});
```

A window specification embedded within an aggregate stream or tuple join statement is only available to that statement, as shown below which creates a window advancing every second.

```
create window squotewindow ( size 100 advance 1s )
```

The `advance` keyword of StreamSQL's windowing construct enables developers to support both sliding windows and tumbling windows by modifying the time interval over which a window is created as well as the number of tuples by which the window advances.

As discussed earlier in Table 4.1, tumbling windows have PI disposal semantics and meet all properties, while sliding windows violate numerous properties as shown in Table 4.1.

Event Composition

Event composition is of two types:

- *Stream and Stream-Relation Join:* Joining two streams or joining a stream with a traditional relation is accomplished by the `select` statement. The `select` statement can be followed by a `delete` statement similar to vanilla SQL. This can be used to support all four disposal schemes by using `select` and `delete` inside a transaction. In the absence of transactions, Stream-Stream and Stream-Relation joins only exhibit I disposal semantics.
- *Windowing and Joining:* Joining multiple windows, streams with windows and windows with tables is also accomplished by the `select` statement. The `select` statement can also be used to match contiguous events and non-contiguous events in a window with events in a table or other streams. Along with the flexibility to place arbitrary sets of statements inside a transaction, StreamSQL supports all event disposal and matching semantics outlined in Table 4.1.

Union and Merge

`union` and `merge` statements take two or more input streams with compatible schemas (structurally equivalent types following the traditional definition of structural equivalence) and produces one output stream with all the tuples from the original streams. The difference between the operators is that `merge` can be used to order the output based on some field of the input. `union` and `merge` may thus violate MDM Type Total Order, MDM Type FIF Order and MDM Type Causal Order.

### 4.4.3 The EQL Language

EQL (Event Query Language) is an object-oriented event stream query language for the Esper engine[4]. EQL has a similar syntax to SQL, but adds further functionality for event stream processing such as sliding and tumbling windows for continuous queries over event streams. The full syntax for EQL is as follows:

```
[insert into insert_into_def]
select select_list from stream_def1
       [as name1], ..., stream_defn [as namen]
[where search_conditions]     [group by grouping_expression_list]
[having grouping_search_conditions] [output output_specification]
[order by order_by_expression_list] [limit num_rows]
```

Further operations include grouping, aggregation, sorting, filtering, merging, splitting or duplicating of event streams, combining windows with intersection and union semantics, inner and outer joins. As with CEL and StreamSQL, the `select` statement causes a violation of the properties MDM Type Total Order, MDM Type FIFO Order and MDM Type Causal Order since `select` uses I disposal. The use of the `limit` operator causes the violation of MDM Event Validity since rows in this context are events, and limiting the number of events in a relation may cause some matching events to be discarded. The use of sliding windows causes the violation of MDM No Duplication. The use of the union and merge semantics can violate MDM Type Total Order, MDM Type FIFO Order and MDM Type Causal Order as described in Section 4.4.2 for StreamSQL.

[4] http://esper.codehaus.org/

### 4.4.4 The CEL Language

The Cayuga Event Language (CEL) [25] is the language used to specify event correlation queries over event streams in the Cayuga complex event processing system. CEL is similar to SQL, and adds three main operators for event correlation – **`filter`**, **`fold`** and **`next`**. All three operators are part of the **`select`** statement, whose general syntax is: **`select`** *`attributes`* **`from`** *`stream_expression`* **`publish`** *`output_stream`*. The **`select`** clause in CEL is similar to the SQL **`select`** clause. Each event stream has a fixed schema similar to that of an SQL table. The **`filter`** operator specifies predicates that compare a single event schema to a constant, e.g., **`filter`** `Company = 'Google'` on a stock quote stream when one of its fields is 'Company.' The **`next`** operator, on the other hand is used for event correlation. The **`fold`** operator is used to specify tumbling windows. The CEL **`select`** statement uses I disposal of events, i.e., unmatched events remain in the queue to be matched to subsequent **`select`** or other join statements. Hence, the **`select`** statement causes a violation of the properties MDM Type Total Order, MDM Type FIFO Order and MDM Type Causal Order.

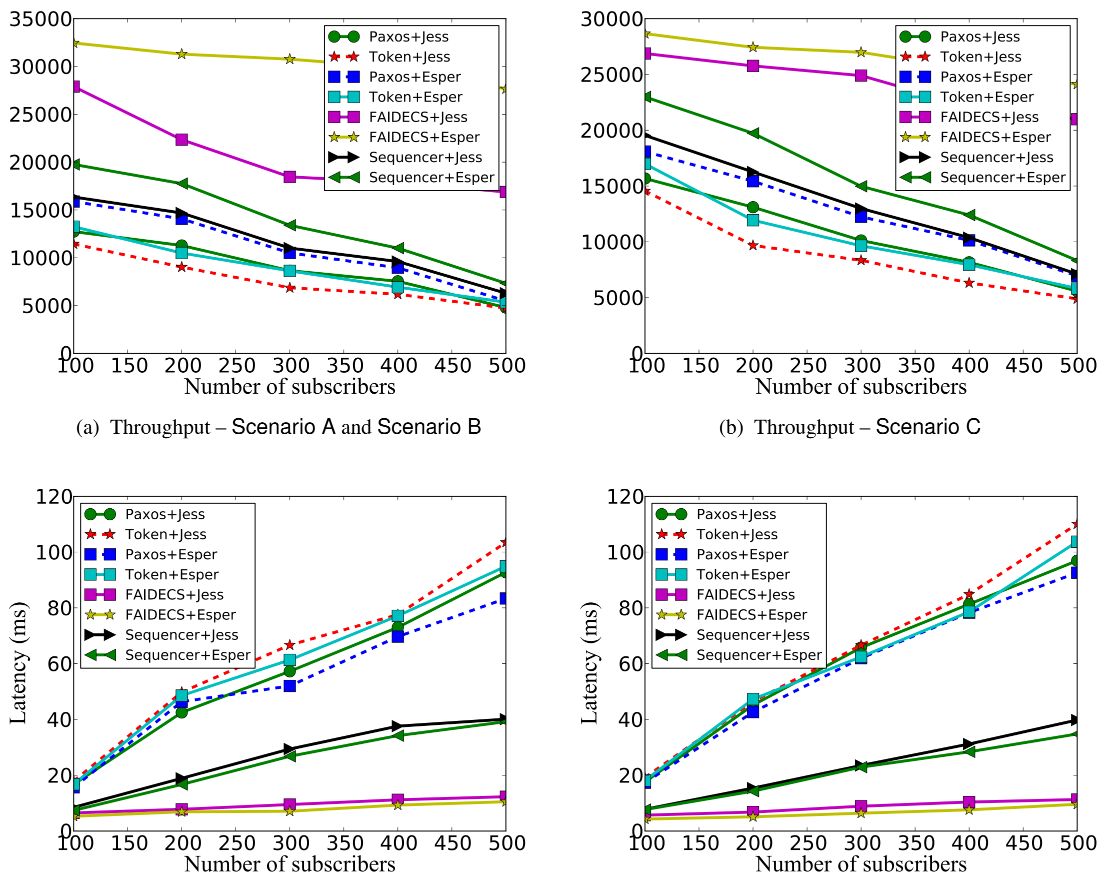
## 4.5 Evaluation

To demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs, we compare the performance of the FAICECS system using two different matching engines implemented in Java — Esper (`http://esper.codehaus.org`) and Jess (`http://www.jessrules.com`) — with three different implementations of a global total order: two fault tolerant ones and a non-replicated sequencer (with Esper and Jess again) for event correlation at the subscribers. We have included the non-replicated non-fault tolerant sequencer because that is the most efficient sequencer.

### 4.5.1 Metrics and Setup

We used two metrics: (1) *throughput* measures the average number of events delivered per second by a subscriber; (2) *latency* measures the average delay between the production time of an event and its delivery to a subscriber. We chose subscriptions based on the default workload in the Marketcetera algorithmic trading system (`http://marketcetera.org/`). In this workload, the publisher is the Marketcetera stock exchange simulator, and the subscribers are algorithmic traders. The default workload has 23 event types, and several conjunctions. The maximum number of event types in any conjunction is 6. The number of subscribers (traders) was increased from 10 to 500. We used three nodes for Paxos and the token passing total order implementation, i.e., the state of the replicated fault tolerant sequencer was replicated on three nodes. For both Paxos and Token-passing total order, the publisher sent its events randomly to one of the three nodes.

We have three deployment scenarios. With FAIDECS, conjunctions are performed by merger processes and predicates are evaluated at the subscribers by two popular event correlation systems – Jess (originally used in [82]) and Esper. In Scenario A and Scenario B, we used a setup for conjunctions similar to Figure 4.2. All filtering occurred at end nodes rather than in mergers through the selectivity of binary predicates, which differed across conjunctions to achieve the same expected delivery rates at all subscribers in a respective level. This scenario demonstrated the limits of the overlay. In Scenario B, events were filtered at the mergers through unary predicates propagated upwards from subscriptions, allowing higher aggregate multicast rates than in Scenario A. In Scenario C, we statistically generated subscriptions uniformly over all event types in the system with all possible conjunction combinations. This allowed us to explore the potential of traffic separation. Subscribers were uniformly distributed across all merger processes and throughput/latency values were averaged for each group of subscribers for a given level. We expect that the bottleneck in our decentralized algorithms would occur at the merger process(es), which would merge all involved types, limiting throughput consistently for all overlay depths from either the publisher or subscriber.




(a) Throughput – Scenario A and Scenario B

(b) Throughput – Scenario C

(c) Latency – Scenario A and Scenario B

(d) Latency – Scenario C

Figure 4.9.: Empirical evaluation of FAIDECS

### 4.5.2 Results

Figure 4.9 illustrates our results, both for throughput and latency. We observe from all four figures that the results for the sequencer are better than possibly expected. This is because the sequencer we used was a non-fault tolerant counter. Regardless, both versions of FAIDECS easily outperform the corresponding sequencer implementations. This demonstrates how correlation-specific ordering enables strong guarantees even with support for fault tolerance. Both Jess and Esper are current state of the art correlation languages, but in some scenarios, as seen above, Esper is more efficient than Jess since Esper uses a more optimized event correlation algorithm. In either case however, the benefits of the FAIDECS overlay are preserved – in fact, a more efficient matching engine further amplifies its benefits. For Scenario A and Scenario B, the throughput of FAIDECS is at least 84%-4.82× higher than that of the sequencer. The corresponding numbers for Scenario C are 59% to 4.12× higher than that of the sequencer. The difference in latency is yet more pronounced, because lower throughput typically has a cascading effect on latency when the number of subscribers is high. The latency of FAIDECS is up to 3.7× lower than the sequencer, and scales much better than the sequencer. The throughput of the implementations with Paxos and Token-passing total order are lower than Sequencer, though sometimes the differences are less pronounced due to the processing in Esper and Jess at the subscribers.

## 4.6 Related Work

Event correlation has been vigorously investigated in the context of *content-based publish/subscribe systems*. Most such systems rely on a *broker network* for routing events to the subscribers (e.g., SIENA [15] and Gryphon [4]). *Advertisements* are typically used to form routing trees in order to avoid propagating subscriptions by flooding the broker network. Upon receiving an event $e$, a broker determines the subset of parties (subscribers and brokers) with matching interests and forwards $e$ to them. Subscription *subsumption* [15] is used to summarize subscriptions and avoid redundant matching on brokers and redundant traffic among them. If any event $e$ that matches a first subscription also matches a second one, then the latter subscription subsumes the former one.

A broker network can be used to gather all publications for the elementary subscriptions and perform correlation matching. A successful match yields a composite event which is delivered to interested subscribers, where no guarantees are typically provided on correlation. If the events matching a composite subscription shared by two subscribers are produced by several publishers, then unless the subscribers are connected to a same edge broker, they may receive the events through different routes. This leads to different orders among the events and consequently to different composite events for the two subscribers. PADRES [60] performs composite event detection for each subscription at the first broker that accumulates all the individual subscriptions, providing no global properties. Hermes [66] proposes *complex event detectors* using an interval timestamp model as a generic extension for existing middleware architectures. Hermes uses a DHT to determine rendezvous nodes for publishers and subscribers; however, these are not replicated for fault-tolerance.

Recent work [84], motivated by solving agreed correlation, further demonstrates the need for stronger guarantees on correlated deliveries. However the approach proposes a more generic primitive for publish/subscribe systems which is opportunistically layered on top of an existing overlay network, leading to high overhead.

Hummer et al. [45] propose a unified fault taxonomy for general event-based systems. This work generically categorizes faults into separate classes as well as the sources for these faults to better detect and predict faults in future systems.

The work of Lumezanu et al. [61] proposes a decentralized network of sequencers and uses a DHT for load balancing. However, this work only provides total order among messages of the same type/topic, and not for conjunctions, and thus differs from FAIDECS, which performs decentralized merging for conjunctions of types. One could implement FAIDECS-style mergers on top of Lumezanu et al.'s work [61] by mapping conjunctions as types in the DHT and routing messages from the node responsible for a type to a node responsible for a conjunction. (The merging additionally would take predicates into account.) A similar approach could be used to deal with disjunctions (omitted from this paper for simplicity). The work of Baldoni et al. [9] establishes an ordering among topics, and totally orders events within topics and to some degree across, however without distinguishing (guarantees) across conjunctions and disjunctions. Their system is devised to work on top of an arbitrary basic publish/subscribe system (which improves its portability but adversely affects latency), but then still allows messages to be explicitly delivered out of order to the application with a corresponding specific notification.

TimeStream [68] is a recent fault-tolerant stream processing architecture, which is similar to Apache Storm, except for additional reconfiguration and re-starting guarantees provided to stream processing elements. However, TimeStream does not provide ordering guarantees because it is targeted at generic stream processing, where each processing element contains arbitrary code, and is not targeted at events or tuples of data. Aurora [3] and its successor Borealis [8] are seminal stream processing systems, where Borealis uses replication for fault tolerance. Each replica processes events in the same order, and Borealis provides TYPE TOTAL ORDER, but does not provide CONJUNCTION TOTAL ORDER, i.e., in Borealis, it is possible to obtain total order among subscribers for all messages delivered on a given type, e.g., "StockQuote", but not a total order among messages delivered to subscribers on a join or a conjunction, e.g., "StockQuote and AnalystReport". The guarantees provided by System S [48] are similar to Borealis, but the mechanisms (E.g.

checkpointing techniques and orchestration) differ. Cayuga [24] is a generic correlation engine supporting correlation across streams and is based on a very expressive language but is centralized.

## 4.7 Conclusions

FAIDECS presents a powerful event correlation model for trading between (a) strong guarantees in the face of failures and (b) performance; its implementation hinges on an overlay network for deterministic type-wise merging of event flows with replication of merger nodes. We have presented semantic options for several modules of the FAIDECS matching engine. We have shown for each of these alternatives which of the proposed properties are maintained and which are violated. We have investigated four correlation languages – StreamSQL, EQL, CEL and TESLA – and have mapped their features to the semantic options introduced. This then determines which properties are withheld when replacing the matching engine of FAIDECS with that of the respective correlation languages. To demonstrate that the benefits of the FAIDECS overlay in terms of performance (besides fault tolerance) are not dependent on any specific matching engine (while its specific properties do depend on the corresponding correlation language) we substituted the default engine of FAIDECS (Jess) by the more efficient Esper engine. As we have illustrated, Esper in fact amplifies the benefits of the FAIDECS overlay.

Besides investigating further semantic options and properties especially in the context of *disjunctions*, we are currently in the process of investigating security features for FAIDECS.

LIST OF REFERENCES

# LIST OF REFERENCES


[1] OpenFlow. `http://www.openflow.org`.

[2] SPECjms 2007. `http://www.spec.org/jms2007/`.

[3] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The International Journal on Very Large Data Bases*, 2003.

[4] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching Events in a Content-based Subscription System. In *ACM Symposium on Principles of Distributed Computing*, 1999.

[5] M.K. Aguilera and S. Toueg. Randomization and Failure Detection: A Hybrid Approach to Solve Consensus. *Distributed Algorithms*, Volume 1151, 1996.

[6] H.S. Alavi, S. Gilbert, and R. Guerraoui. Extensible Encoding of Type Hierarchies. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[7] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[8] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD International Conference on Management of Data*, 2005.

[9] R. Baldoni, S. Bonomi, M. Platania and L. Querzoni. Dynamic Message Ordering for Topic-based Publish/Subscribe Systems. In *IEEE International Parallel and Distributed Processing Symposium*, 2012.

[10] R. Baldoni, S. Cimmino, and C. Marchetti. Total Order Communications: A Practical Analysis. In *European Dependable Computing Conference*, 2005.

[11] A. Basu, B. Charron-Bost, and S. Toueg. Simulating Reliable Links with Unreliable Links in the Presence of Failures. In *Workshop on Distributed Algorithms on Graphs*, 1996.

[12] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004.

[13] K.P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 1999.

[14] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient Filtering in Publish-Subscribe Systems using Binary Decision Diagrams. In *International Conference on Software Engineering*, 2001.

[15] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Compututer Systems*, 19(3):332–383, 2001.

[16] A. Carzaniga and A.L. Wolf. Forwarding in a Content-Based Network. In *Special Interest Group on Data Communication*, 2003.

[17] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Very Large Data Bases*, 1994.

[18] C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *The International Journal on Very Large Data Bases*, 11:354–379, 2002.

[19] T.D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 1996.

[20] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 1996.

[21] G. Cugola and A. Margara. Tesla: A Formally Defined Event Specification Language. In *ACM International Conference on Distributed Event-Based Systems*, 2010.

[22] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming in AmbientTalk. In *European Conference on Object-Oriented Programming*, 2005.

[23] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 2004.

[24] A.J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W.M. White. Towards Expressive Publish/Subscribe Systems. In *International Conference on Extending Database Technology*, 2006.

[25] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma and W. White Cayuga: A General Purpose Event Monitoring System. In *Conference on Innovative Data Systems Research*, 2007.

[26] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publishing Company, 2010.

[27] P. Eugster and K.R. Jayaram. EventJava: An Extension of Java for Event Correlation. In *European Conference on Object-Oriented Programming*, 2009.

[28] F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *ACM SIGMOD International Conference on Management of Data*, 2001.

[29] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 1985.

[30] C.L. Forgy. On the efficient implementation of production systems. PhD thesis, Carnegie Mellon University, 1979.

[31] N. Fotiou, P. Nikander, D. Trossen and G.C. Polyzos. Developing Information Networking Further: From PSIRP to PURSUIT. In *International Conference on Broadband Communications, Networks, and Systems*, 2010.

[32] N. Fotiou, D. Trossen and G. Polyzos. Illustrating a Publish-Subscribe Internet Architecture. *Springer Journal on Telecommunication Systems*, 2011.

[33] H. Garcia-Molina and A. Spauster. Message Ordering in a Multicast Environment. In *International Conference on Distributed Computing Systems*, 1989.

[34] S. Gatziu and K.R. Dittrich. Detecting Composite Events in Active Database Systems using Petri Nets. In *International Workshop on Research Issues in Data Engineering*, 1994.

[35] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Very Large Data Bases*, 1992.

[36] J. Gil and Y. Zibin. Efficient Subtyping Tests with PQ-Encoding. *ACM Transactions on Programming Languages and Systems*, 27(5):819–856, 2005.

[37] L. Greenemeier. Content Is King: Can Researchers Design an Information-Centric Internet? Scientific American, December 2012. `http://www.scientificamerican.com/article.cfm?id=internet-infrastructure-information-redesign`

[38] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T.E. Anderson, B.N. Bershad, G. Borriello, S.D. Gribble, and D. Wetherall. System Support for Pervasive Applications. *ACM Transations on Computer Systems*, 2004.

[39] R. Guerraoui and A. Schiper. Genuine Atomic Multicast in Asynchronous Distributed Systems. *Journal of Theoretical Computer Science*, 2001

[40] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. Streamcloud: A Large Scale Data Streaming System. In *International Conference on Distributed Computing Systems*, 2010.

[41] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. In *Symposium on Software Engineering for Parallel and Distributed Systems*, 2000.

[42] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. *Distributed Systems*, 2nd edition, 1993.

[43] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D.G. Anderson, J.W. Byers, S. Seshan and P. Steenkiste XIA: Efficient Support for Evolvable Internetworking. In *Symposium on Networked Systems Design and Implementation*, 2012.

[44] M. Hennessy and J. Rathke. Bisimulations for a Calculus of Broadcasting Systems. *Journal of Theoretical Computer Science*, 200(1-2):225–260, 1998.

[45] W. Hummer, C. Inzinger, P. Leitner, B. Satzger and S. Dustdar Dreiving a Unified Fault Taxonomy for Distribute Event-Based Systems. In *ACM International Conference on Distributed Event-Based Systems*, 2012.

[46] B. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, and R. Braynard. Networking Named Content. In *International Conference on Emerging Networking Experiments and Technologies*, 2009.

[47] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N. Briggs and R. Braynard. Networking Named Content. In Communications of the ACM 2012.

[48] G. Jacques-Silva, J. Challenger, L. Degenero, J. Giles and R. Wagle. Towards Autonomic Fault Recovery in System-S. In *International Conference on Autonomic Computing*, 2007.

[49] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H.Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbetts and S. Zdonik. Towards a Streaming SQL Standard. In *Very Large Data Bases*, 2008.

[50] K. Jayaram and P. Eugster. Scalable Efficient Composite Event Detection. In *International Conference on Coordination Models and Languages*, 2010.

[51] K.R. Jayaram, C. Jayalath, and P. Eugster. Parametric Subscriptions for Content-Based Publish/Subscribe Networks. In *Middleware*, 2010.

[52] Z. Jerzak and C. Fetzer. Bloom Filter based Routing for Content-based Publish/Subscribe. In *ACM International Conference on Distributed Event-Based Systems*, 2008.

[53] G.G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive Event Correlation in Distributed Systems. In *ACM International Conference on Distributed Event-Based Systems*, 2010.

[54] R.R. Kompella, J. Yates, A.G. Greenberg, and A.C. Snoeren. IP Fault Localization Via Risk Modeling. In *Symposium on Networked Systems Design and Implementation*, 2005.

[55] T. Koponen, M. Chawla, B.G. Chun, A. Ermolinkiy, K.H. Kim, S. Shenker and I. Stoica. A Data-Orented (and Beyond) Network Architecture. In *Special Interest Group on Data Communication*, 2007.

[56] C. Krugel, T. Toth, and C. Kerer. Decentralized Event Correlation for Intrusion Detection. In *International Conference on Information Security and Cryptology*, 2002.

[57] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 1978.

[58] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-Oriented Pattern Matching for Evolvable Distributed Systems. In *International Conference on Object Oriented Programming, Systems, Languages and Applications*, 2003.

[59] G. Li, S. Hou, and H.A. Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. In *International Conference on Distributed Computing Systems*, 2005.

[60] G. Li and H. Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. In *Middleware*, 2005.

[61] C. Lumezanu, N. Spring and B. Bhattacharjee. Decentralized Message Ordering for Publish/Subscribe Systems. In *Middleware*, 2006.

[62] T.D. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and Modular Predicate Dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 2009.

[63] E. Nordstrom, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S.Y. Ko, J. Rexford and M.J. Freedman. An End-Host Stack for Service-Centric Networking. In *Symposium on Networked Systems Design and Implementation*, 2012.

[64] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Symposium on Operating Systems Principles*, 1993.

[65] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving Agreement Problems with Weak Ordering Oracles. In *European Dependable Computing Conference*, 2002.

[66] P.R. Pietzuch, B. Shand, and J. Bacon. A Framework for Event Composition in Distributed Systems. In *Middleware*, 2003.

[67] K.V.S. Prasad. A Calculus of Broadcasting Systems. *Science of Computer Programming*, 25(2-3):285–327, 1995.

[68] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu and Z. Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *European Conference on Computer Systems*, 2013.

[69] E. Rabinovich, O. Etzion, S. Ruah, and S. Archushin. Analyzing the Behavior of Event Processing Applications. In *ACM International Conference on Distributed Event-Based Systems*, 2010.

[70] S.P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, 1990.

[71] M. Sadoghi and H.A. Jacobsen. BE-Tree: an Index Structure to Efficiently Match Boolean Expressions over High-dimensional Discrete Space. In *ACM SIGMOD International Conference on Management of Data*, 2011.

[72] C. Sanchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dill, and Z. Manna. Event Correlation: Language and Semantics. In *International Conference on Embedded Software*, 2003.

[73] P. Sewell, J.J. Leifer, K. Wansbrough, F. Zappa Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. *Journal of Functional Programming*, 17(4-5):547–612, 2007.

[74] H. Sturzrehm, P. Felber, and C. Fetzer. TM-Stream: An STM Framework for Distributed Event Stream Processing. In *IEEE International Parallel Distributed Processing Symposium*, 2009.

[75] K.J. Sullivan and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM Transactions on Software Engineering Methodology*, 1(3):229–268, 1992.

[76] N. Tatbul, U. Çetintemel, and S.B. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Very Large Data Bases*, 2007.

[77] P. Triantafillou and A.A. Economides. Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems. In *International Conference on Distributed Computing Systems*, 2004.

[78] A.J. Turon and C.V. Russo. Scalable Join Patterns. In *International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.

[79] A. Ulbrich, G. Mühl, T. Weis, and K. Geihs. Programming Abstractions for Content-Based Publish/Subscribe in Object-Oriented Languages. In *On the Move -OTM- to Meaningful Internet Systems and Ubiquitous Computing*, 2004.

[80] G.A. Wilkin and P. Eugster. Multicast with Aggregated Deliveries. *Journal on Parallel and Distributed Computing*, 2012.

[81] G.A. Wilkin and P. Eugster. Multicasting in the Presence of Aggregated Deliveries. *Journal on Parallel and Distributed Computing*, 2013 (pre-version in *AlMoDEP*, 2011).

[82] G.A. Wilkin, K.R. Jayaram, P. Eugster, and A. Khetrapal. FAIDECS: Fair Decentralized Event Correlation. In *Middleware*, 2011.

[83] T.W. Yan and H. García-Molina. Index Structures for Selective Dissemination of Information under the Boolean Model. *ACM Transactions on Database Systems*, 19:332–364, 1994.

[84] K. Zhang, V. Muthusamy and H.A. Jacobsen. Total Order in Content-Based Publish/-Subscribe Systems. In *International Conference on Distributed Computing Systems*, 2012

[85] Y. Zhao and R.E. Strom. Exploiting Event Stream Interpretation in Publish-Subscribe Systems. In *ACM Symposium on Principles of Distributed Computing*, 2001.

# VITA

# VITA

Gregory Aaron Wilkin was born in Rogers Arkansas. After completing his schoolwork in 1998 at the age of fourteen, Aaron was able to begin his Bachelor of Science in Computer Science degree in 2000 at Arkansas State University on scholarship. Upon completion, Aaron further pursued a Master of Science in Computer Science at Arkansas State University. After being accepted to the Purdue University Graduate School program in 2008, Aaron pursued his PhD in Computer Science there. In 2012, Aaron began doing research in absentia to accept a tenure track Assistant Professor position at Rose-Hulman Institute of Technology where he is currently employed.