Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2017

Francisco Marco Morais Alves

Plataforma de Localização Suportada por Utilizadores de Redes Móveis

Framework for Location Based System Sustained by Mobile Phone Users

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
Ano 2017

Francisco Marco Morais Alves

Plataforma de Localização Suportada por Utilizadores de Redes Móveis

Framework for Location Based System Sustained by Mobile Phone Users

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática realizada sob a orientação científica do Doutor Óscar Narciso Mortágua Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais, à Rita, aos meus irmãos, cunhada e sobrinhos. Pelo seu incansável e sempre presente apoio.

o júri / the jury

presidente / president	Prof. Doutor José Manuel Matos Moreira Professor auxiliar da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor João Pedro Carvalho Leal Mendes Moreira Professor auxiliar da Universidade do Porto – Faculdade de Engenharia
	Prof. Doutor Óscar Narciso Mortágua Pereira Professor auxiliar da Universidade de Aveiro

agradecimentos

Quero expressar o meu agradecimento ao professor Óscar Pereira pela disponibilidade e orientação científica ao longo desta dissertação.
Ao Engenheiro Mário Moreira agradeço toda a disponibilidade, compreensão e ajuda técnica que tornou possível a concretização deste trabalho.
À Engenheira Telma Mota e a todos os colegas da Altice Labs que me acompanharam neste percurso.

palavras-chave Flink, LBSN, LBS, Hadoop, HDFS, CDR, EDR, Radius, BTS, Kafka, tolerância a falhas, stream processing, plataforma, fiável, escalável.

resumo Vivemos na era da informação e da Internet das coisas e por isso nunca antes a informação teve tanto valor, ao mesmo tempo nunca existiu tão elevada troca de informação. Com toda esta quantidade de dados e com o aumento substancial do poder computacional, tem-se assistido a uma explosão de ferramentas para o processamento destes dados em tempo real.

Um novo paradigma também emergiu, pelo facto de que muita dessa informação tem meta informação da qual é possível extrair conhecimento adicional quando enriquecida.

No caso dos operadores de telecomunicações existem vários fluxos de informação trocados entre dispositivos dos clientes, utilizadores de redes móveis e as antenas. Como exemplos são os casos dos pacotes Radius, Call Detail Records CDR's e os Event Detail Records EDR's que servem para o controlo de tráfego e para outros tipos de controlo e configurações. Em muitos destes pacotes vem incluída informação geográfica e temporal.

Depressa se torna claro que a partir desta informação geográfica é possível extrair conhecimento e por isso valor adicional para os detentores da informação.

Esta dissertação recorre a fluxos devidamente anonimizados que possuem informação de antenas (id e por isso posição e distância ao dispositivo). Neste trabalho é apresentada uma solução escalável e fiavél que num ambiente de streaming determina a posição dos utilizadores de redes móveis, através de triangulação. A solução também determina métricas relativas a áreas geográficas. Devido a dificuldades externas, estes fluxos (dados) tiveram de ser simulados. As áreas são definidas e introduzidas por utilizadores da aplicação de forma a saberem as entradas e saídas, bem como o tempo de permanência em uma determinada área. Sendo o processamento realizado em ambiente de streaming, a solução desenvolvida tem de ser capaz de recuperar de falhas quando elas existirem de uma forma coerente e consistente.

keywords Flink, LBSN, LBS, Hadoop, HDFS, CDR, EDR, Radius, BTS, Kafka, fault tolerance, stream processing, framework, reliable, scalable.

abstract The time we live in is the time of information and the time of the Internet of Things. So, never before information had so much value. On the other hand, the volume of information exchange grows exponentially day by day. With all this amount of data as well with the computational power available nowadays, real time data processing tools emerge every day.
A new paradigm emerges because there is a lot of meta information in this data exchange. With the enrichment of this meta information, it is possible to extract additional knowledge.
From a telecommunication company point of view, there is a lot of exchanged data flows between clients' devices and the Base Transceiver Station (BTS) such as, Radius packets, Call Detail Records (CDR) and Event Detail Records (EDR). Frequently, these flows are for control and configurations purposes. But in many cases, it also contains geographical and time information.
Soon was clear that it is possible to perform data enrichment on this geographical information, in order to extract additional knowledge. In other words, additional value for the telecommunication company.
This dissertation through data flows previously anonymized, that contain BTS's information (e.g. position and distance from the client mobile), grants one scalable and reliable solution on a streaming environment that determines multiple metrics related to geographical areas. Due to external difficulties, it was necessary to simulate all the data flows. These areas are inputted by application user clients in order to know the number of people that get in or out of these areas as well the time spent inside. Since the work is done on streaming environment, the solution presented is able to recover from failures and fault tolerant in a consistent and coherent manner.

Index

List of figures

List of Listings

List of Tables

List of Charts

List of Acronyms

API	Application Programming Interface
BTS	Base Transceiver Station
CDR	Call Detail Record
DB	Database
DSL	Domain Specific Language
EDR	Event Detail Record
GNU	Gnu Not Unix
GPL	General Public License
GPS	Global Positioning System
HDFS	Hadoop Distributed File System
HTML	Hyper Text Markup Language
IoT	Internet of Things
JDBC	Java Database Connectivity
JM	Job Manager
JVM	Java Virtual Machine
LBS	Location Based Service
LBSN	Location Based Social Network
M2M	Machine to Machine
ML	Machine Learning
NAS	Network Access Server
RADIUS	Remote Authentication Dial In User Service
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
TM	Task Manager
TS	Task Slot

UDF	User Define Function
UDP	User Datagram Protocol
YARN	Yet Another Resource Negotiator

1. Introduction

This chapter aims to present a brief introduction, the motivations as well as the proposed objectives of this dissertation and ends with the structure of the document.

1.1 Preamble

Recently, with the explosion of Location Based Social Networks (LBSN) there was a very high number of exchanging data between users. A good example is Twitter where every day around half billion twits are sent [1] and a considerable number of these twits have geographical information [2]. Hence, it is possible to extract geographic information and infer behavior patterns about the users mobility [3], [4] or even predict users interests [5]. A good example is to infer knowledge about users locations and patterns of the users trajectories.

Regarding the mobile networks, there has been a continuously growth of mobile devices (figure 1), and consequently data exchange between mobile clients and Telecommunication company servers or services has also increased. In particular, for the MEO mobile network, the current number of daily data flows (sessions) are about 1Tb.

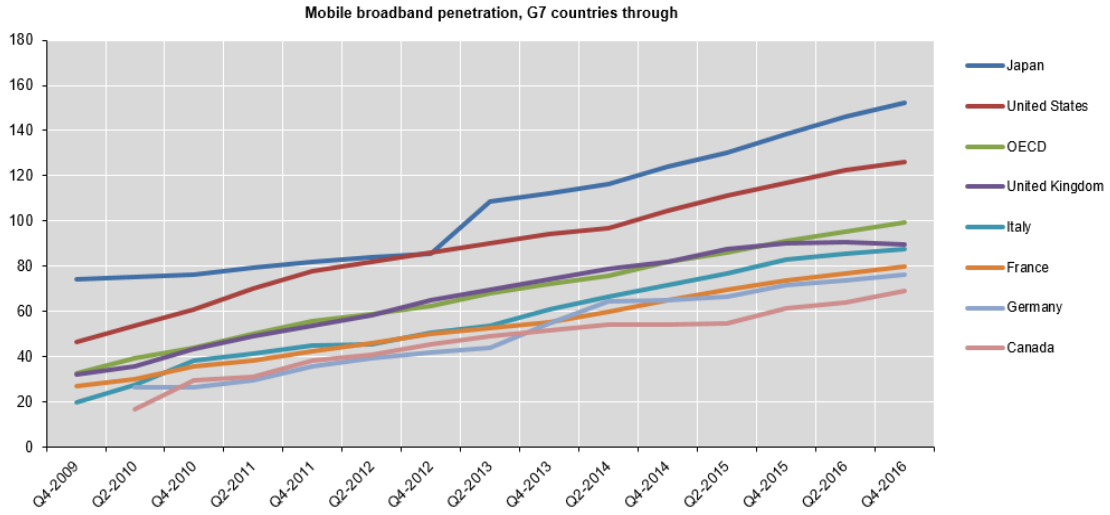

Figure 1: *Mobile broadband penetration in G7 countries* [6].

Several studies that analyze meta data from monitoring and control mobile network e.g. Call Detail Records (CDR's) and Event Detail Records (EDR's), already showed that it is possible through data mining and clustering methods to perform users classifications and behaviors patterns [7], [8]. One can also infer location based knowledge from geographical information present in meta data. Such as Base Transceiver Station (BTS) id's and consequent position.

Another approach is aimed to use BTS activity to infer land use patterns[9]. Note that authentication, authorization and accounting exchanged data have geographic information like the id of BTS. This association between mobile devices and BTS makes it possible not just to know the device position, but also the trajectory over the BTS's. Making this a window of opportunity to use all this information to determine users location behaviors and interests patterns. Due to this, it is possible to develop a location based solution sustained by mobile networks clients activities.

1.2 Motivations

For a Telecommunication company that already deals with this great amount of mobile data, it is a huge opportunity to infer users location behaviors patterns. The challenge is how to capture and process this enormous amount of real time data. Recently with the continuously development of tools like Hadoop, Flink and Kafka, it is possible to deal and to process huge amounts of data at lower costs.

The benefits of knowledge extracted from this big data processing are enormous. This kind of location knowledge could be applied at diversified fields. Take for example the transport companies or marketing business for advertising spots. As mentioned before, a lot of studies have been done using LBSN [2], [3]. One of the limitations in these studies is the temporal gap between collected data time and the processing time. Once this gap is eliminated the knowledge inferred can have several of other advantages like traffic or security information.

From a technical point of view, it is extremely challenging to work with new tools. In this particular case big data tools. Therefore, the know-how achieved at the end of this study will be certainly a major value in an area with increasing expansion like, big data and stream processing.

1.3 Objectives

The main objective of this dissertation aims at extracting through data from the mobile Telecommunication company network, geographical (locations) and temporal information about generic users daily life. There are multiple types of data that could be

used. In this dissertation we will use Radius packets. This type of packets is used for user accounting, authentication, authorization and security, this way making them a rich resource. Due to the fact that they have attributes such as cell towers id's (consequent position) and distances to the mobile devices among others.

Since the work is based on data from mobile clients, aspects related with privacy and anonymity became of primary importance. Due to this, all information related to users identification is previously anonymized. So, in this dissertation the work is done over anonymous data.

The continuously large amount of receiving packets and the probability of dealing with late incomings and consequently out of order flows, makes the enrichment of the information important. On the other hand, there are inherent problems to the solutions that are continuously reading a streaming. Namely, in case of failure, the solution must be able to restart from the point where the failure occurred, hence with the restored state to prevent reprocess and duplicate readings. Also, all enrichment must be consistent, coherent and accurate, meaning that it has to continue like if no failure happened and with no impact on the results.

Consequently, another aim is to build a reliable, coherent, scalable and fault tolerant solution. This will be used to process and perform the enrichment in a continuous flow stream. The most important aspect in enrichment is the triangulation. This will extract one location (longitude and latitude of the mobile device) from three others that are known (the three cell towers) and their respective distances. The three distances are from the unknown location (mobile device) to the three known locations (cell towers). Additionally, the throughput must be as higher as possible and the latency as small as possible. At the beginning of this dissertation, it was thought to perform the implementation with multiple java modules. However, the plan has changed and it was decided to use the streaming processing framework "Flink", since this is a more appropriate tool to the objective.

Additionally, the application should be able to receive another stream with boxing areas (possibly from application users containing two boxing points and permanency). It has also to compute various metrics related to these areas such as people that get in and out, distinct ins, distinct outs and means of permanency time.

The work developed brings some contributions to all location based systems that will be sustained by a continuous data flow, in particular the ability of having a reliable, scalable and fault tolerant processing. For this work, due to external difficulties, it was not possible to obtain real data flows. So, all the work performed was with simulated data.

1.4 Structure

This document is organized into five chapters. The current chapter covers aspects such motivation and goals for the dissertation. The aspects covered by the remaining ones:

- Chapter 2: background that will support this dissertation.
- Chapter 3: framework requirements, architecture and implementation details.
- Chapter 4: evaluation and discussion of the implemented solution.
- Chapter 5: overview of the implementation and some aspects that should be done in order to improve the solution.

2. Background

This chapter aims to present an overview of the streaming process and the descriptions of the technologies needed to this dissertation.

2.1 Stream Processing

In a world of Internet of Things (IoT) and Machine to Machine communication (M2M), real time data exchange displays exponential growth, that is observed day by day (figure 2) [10].

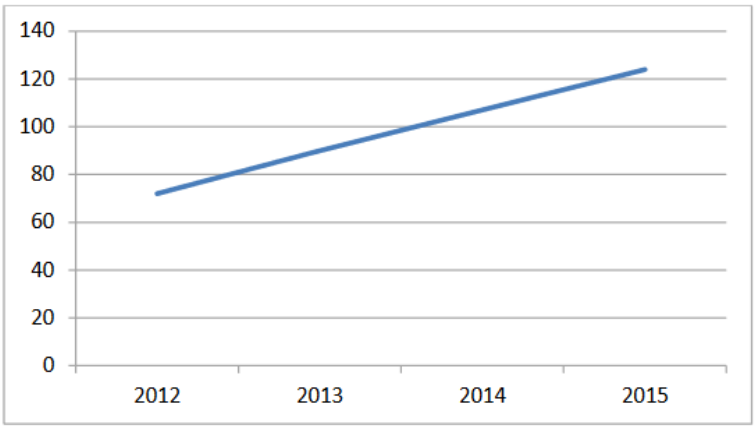

Figure 2: *M2M SIM cards subscriptions in OECD area, millions* [10].

Nowadays, most of all generated data are a continuous stream of information [11], e.g. sensors events, trades, user activities, logs among others. In general, these are events over time and processing must be performed in "real time".

Previously to stream processing, generated data was stored in databases, file systems, etc. All kind of processing performed by applications such analytics was made by query to this storage data (figure 3) [11]. Consequently, the processing was made over bounded data sets. Questions like out-of-order and late-arrive in general were nonexistent.

The streaming process introduces a new methodology: processing, analytics and queries coexist continuously over the stream (figure 4) [11]. So, the processing is made over unbounded data sets or data stream. When a source e.g. one sensor generates an event, the application reacts to perform some kind of action, analytics, among others. These applications are called *Stream Processors*. There are three fundamental aspects

inherent to all *Streams Processors* [11]: *Ensure efficient data flows*, high throughput and low latency; *Computation scales*, the readjust of the scale should not be a megalomaniac operation; *Fault tolerance*, in case of failure the Stream Processor must be able to accurate and reliable restore processing.

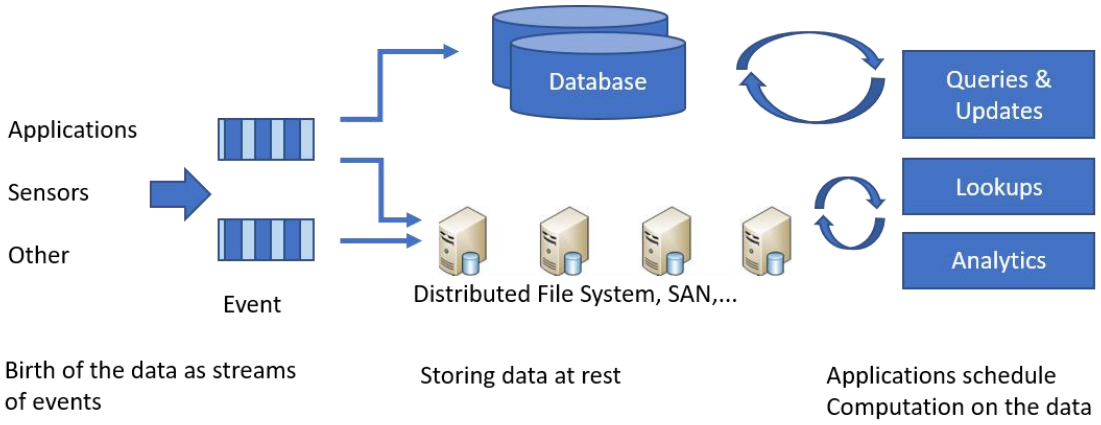

Figure 3: *Before stream processing: data-at-rest infrastructure* [11].

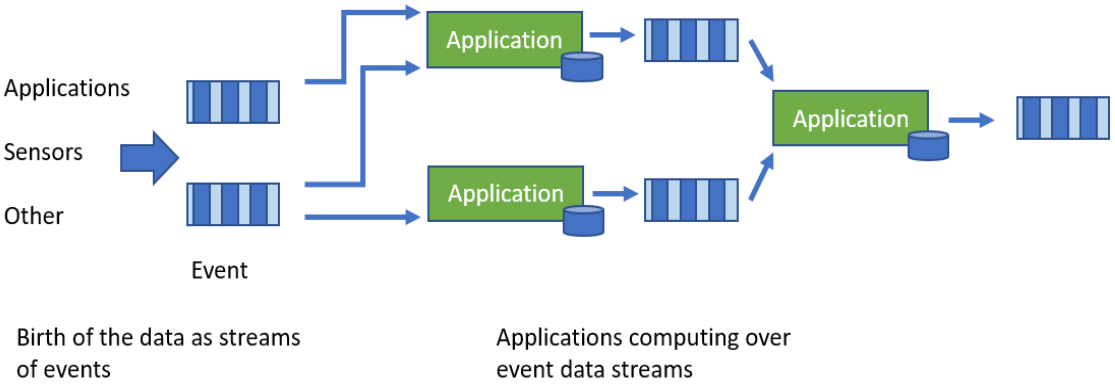

Figure 4: *Stream processing infrastructure* [11].

The necessity to deal with questions like consistency and fault-tolerance brings a subset of streaming processing where an application maintains a context state. This state is used to maintain meaningful information from the previous events that could be used to future computations or even to restore from failures.

Critical applications such as fraud prevention that use state to store last transactions, online recommender applications that keep users preferences in state, or e-commerce applications that use state to maintain the list of items, therefore requiring stateful stream processing. As illustrated in (figure 5) [11] the application maintains its

state on persistent storage. This way the application is always capable to recover from its state. Hence, the state is a very important aspect to all location based systems and, in particular to this work. For example, it is important to keep historic information in order to restore from failures in a reliable way.

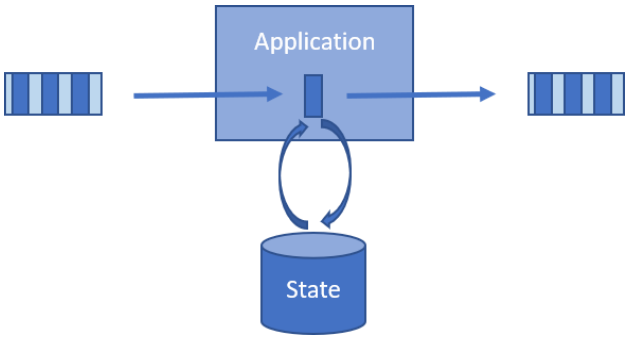

Figure 5: *Stateful stream processing* [11].

2.1.1. Lambda and Kappa Architectures

With the necessity to process "real time" data streams, two architectures emerged. The first was the Lambda architecture brought to us by *Nathan Marz*[12]. It aims to perform reads and updates in linearly, scalable and fault-tolerant way with minimal latencies. Lambda architecture consists by having two layers fed by data stream, the batch layer (batch and serving layer) and the speed layer. The batch layer has the responsibility to store the raw data, and to compute batch views to consumption. Speed layer provides the real-time views to serve as compliment to batch views (figure 6) [12].

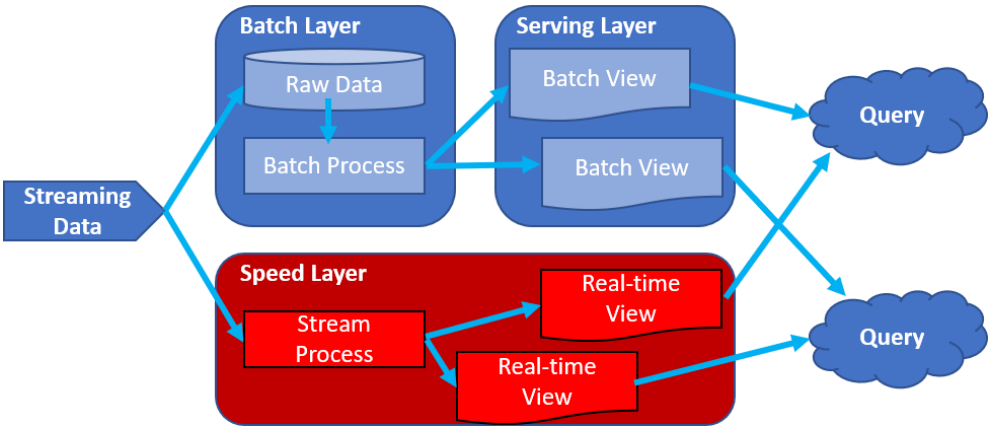

Figure 6: *Lambda Architecture* [12].

Jay Kreps was the first to describe the Kappa architecture[12], not as a replacement to the Lambda architecture but a solution oriented to stream processing without batch.

Kappa aims to keep just one code base, so that real-time data processing and continuous reprocessing is performed by the same engine. Further, there is just one location for views (figure 7) [12]. The data streaming feeds the stream process and therefore it is available for real time views.

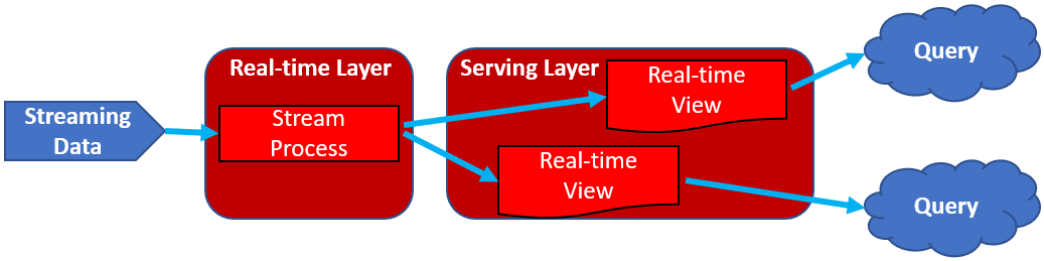

Figure 7: *Kappa Architecture* [12].

2.2 Flink

Flink [13], an Apache Project, is a distributed open source framework that brings important aspects to stream processing such as: possibility of accurately deal with out-of-order and late-arriving data; Stateful and fault-tolerant, possibility to recover from failures and keep the application state; Scalable, as a distributed framework it must be capable of running in thousands of nodes this way leading to a high throughput and low latency.

As mentioned in the above section 2.1 *Stream Processors* have the responsibility to deal with these aspects, making Flink the best tool to distribute stream processing.

Flink also provides the batch processing build on top of streaming engine, so there are two types of execution models: *Streaming*, continuous processing data as long it is being produced; *Batch*, job that runs in a finite interval of time releasing computational resources when finished.

In section 2.1 two types of datasets were mentioned, *Unbounded*, Infinite dataset (stream continuously appended information) and *Bounded*, finite and unchanged dataset.

The approach used by Flink is the *streaming-first* also called the Kappa Architecture. Flink deals with batch as a subset of stream processing with bounded data

streams. So, it is possible to process each type of dataset, *Bounded* or *Unbounded,* with each execution model. Hence, the capability of handling both batch and streaming workloads defined Flink as *Hybrid Processing System* [13], [14].

2.2.1 Stream Processing Model

Stream processing model aims to treat data item-by-item as a real stream, to solve problems such out-of-order and fault-tolerance. In this sense, Flink provides the following features and functionalities: *Snapshots*, Flink streaming tasks periodically take snapshots during computation for failure recovering; *State backends*, used to store state and to make the computation stateful and guarantees the exactly-once semantics; *Event time*, Flink understands this concept, so out-of-order problems can be solved.

Flink provides more than one type of *times*: *Processing Time, Event Time and Ingestion Time*. Developers are free to use the most pleasant to their needs. The *Processing Time* refers to the system time. This is the time when the record is processed in this particular case Flink Window Operator. *Event Time* is the time embedded in record by the producer device and *Ingestion Time* the time that records get in Flink Data Source. All these types of *times* are illustrated in (figure 8) [15].

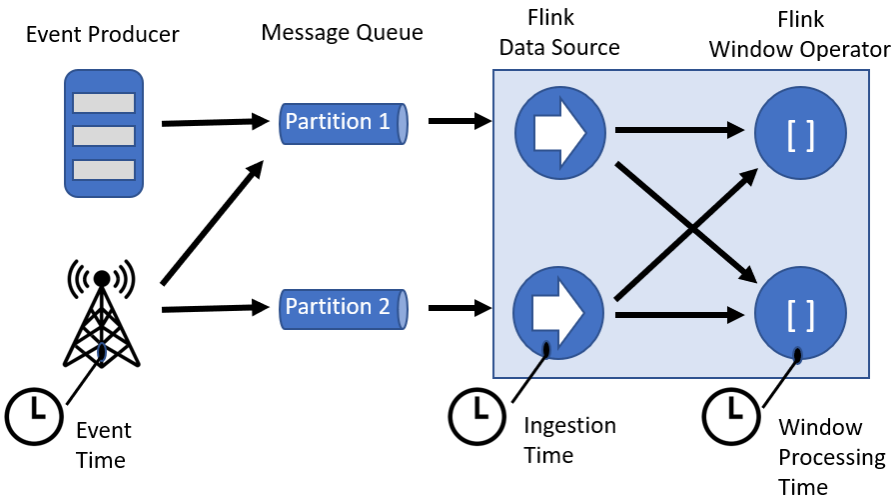

Figure 8: *Available Flink Time Characteristics* [15].

When working with *Event Time,* it is necessary to extract the timestamp from the record and to generate the watermarks. Watermark is the mechanism used by Flink to keep track of the stream progression, e.g. when a *window* is defined, the watermark measures the event progression and notifies the *window* when events pass the

boundary and should be closed. Due to this, it is possible to customize watermarks to handle late events. These watermarks are introduced into the stream as a timestamp when the extraction occurs.

2.2.2 Batch Processing Model

The execution of batch programs is treated as a special case of streaming programs, where the datasets are bounded or unbounded with periodically arriving data. In this case, the batch job is executed over a small part of the unbounded dataset. Those datasets are from a persistent storage as a stream and the runtime used by Flink is the same for both processing models as illustrated in (figure 9) [16] and both API's run over the same engine despite having different workflows .

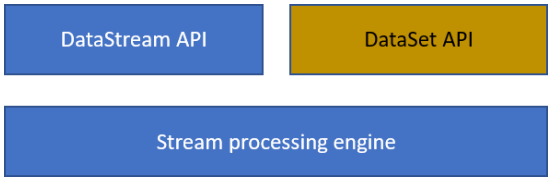

Figure 9: *Streaming and batch over the same engine* [16].

The main problem of having batch programs handling an unbounded dataset is the late-arriving data. This happens because the state is confined to the batch boundaries, and there is no way to correlate events across batches. So, if the data needed to perform a computation is not in the same batch the result will be incorrect. The solution is to introduce a lot of additional overhead to handle late events and state between batches. As expected this solution leads to delaying the processing and reprocessing a batch if needed [17].

2.2.3 Operators and API'S

On its lower level, Flink programs workflow are composed by data source, transformations and data sink (figure 10) [13].

To perform all necessary operations Flink provides operators (table 1). The source operators (listing 1) at starting point are responsible to make the stream available to the transformation operators (listing 2), where processing is performed. Sink operators (listing 3) is where the outcome is treated (figure 11) [18]. In particular case of (figure

11) the transformations performed are the *map, keyBy, window and apply.* But there are other transformations that could be done as mention in (listing 2).

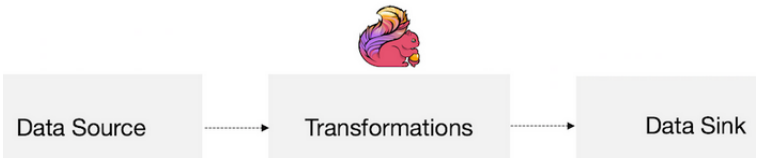

Figure 10: *Flink workflow* [13].

Flink Operators Types	Description
Source	Input reading
Transformation	Transform one or more data streams in new ones
Sink	Consumes data stream and forward them to the output

Table 1: *Flink Operators Types.*

```
readTextFile(path)

readFile(fileInputFormat, path)

readFile(fileInputFormat, path, watchType, interval, pathFilter)

socketTextStream

fromCollection(Collection)

fromCollection(Iterator, Class)

fromElements(T …)

fromParallelCollection(SplittableIterator, Class)

generateSequence(from, to)

addSource
```

Listing 1: *Source Operators.*

```
Map: DataStream → DataStream

FlatMap: DataStream → DataStream

Filter: DataStream → DataStream

KeyBy: DataStream → KeyedStream

Reduce: KeyedStream → DataStream

Fold: KeyedStream → DataStream

Window: KeyedStream → WindowedStream

Connect: DataStream, DataStream → ConnectedStreams

CoMap, CoFlatMap: ConnectedStreams → DataStream

Split: DataStream → SplitStream
```

Listing 2: *Some important Transformations Operators.*

```
writeAsText()

writeAsCsv()

print()

writeUsingOutputFormat()

writeToSocket

addSink
```

Listing 3: *Sink Operators.*

Depending on the use cases and where the transformations are going to actuate on data streams or data sets, Flink disposes two cores APIs: DataStream and DataSet API. Flink also provides the Table API for relational stream and batch processing.

Due to this, there are different levels of abstraction to develop streaming or batch programs (figure 12) [18].

Stateful Stream Processing is the lowest level of abstraction and offers the possibility to process stream events in a consistent and fault tolerant way using state. DataStream API provides the *Process Function* that allows developers to access the lowest level stateful streaming.

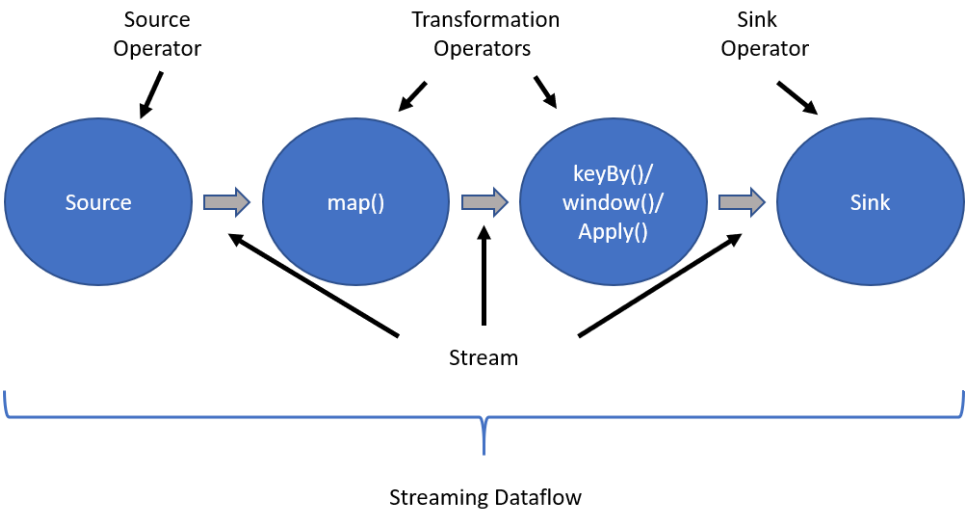

Figure 11: *Streaming data flow* [18].

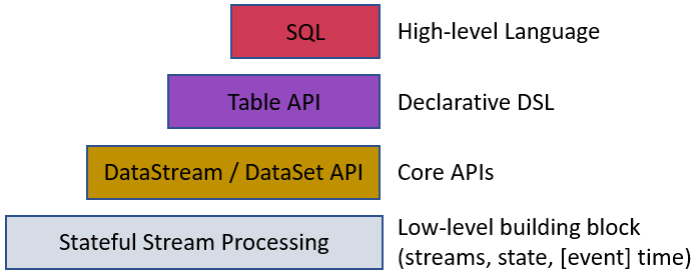

Figure 12: *Flink levels of abstraction* [18].

Generally, the core APIs have all the building blocks necessary to almost all applications. Depending on the streams, developers are allowed to choose between DataStream API for bounded or unbounded streams, or DataSet API for bounded data sets. These APIs offer various forms of user-specified-transformations such as, *joins*, *aggregations*, *map*, *flatmap*, among others. Due to all these features, in most cases, developers are free from digging in the lowest level of abstraction referred above [19], [20].

The declarative DSL (Domain-Specific-Language) Table API provides operations such, *selects*, *joins*, *aggregate*, among others. With all these User-Defined Functions (UDF), makes Table API less expressive and more concise than Core APIs to handle table

like data. The conversion between table and DataStream or DataSet is trivial. Allowing developers to use more than one APIs in the same program [18], [21].

Similar to Table API, Flink provides the highest level of abstraction SQL (Structured Query Language). The programs are seen as SQL query expressions, these queries are executed over tables defined via Table API [18], [21].

2.2.4 Parallel Data Flow

The data flow of Flink programs at their lower level is composed by the basic building blocks: streams and transformations (table 1, listing 2).

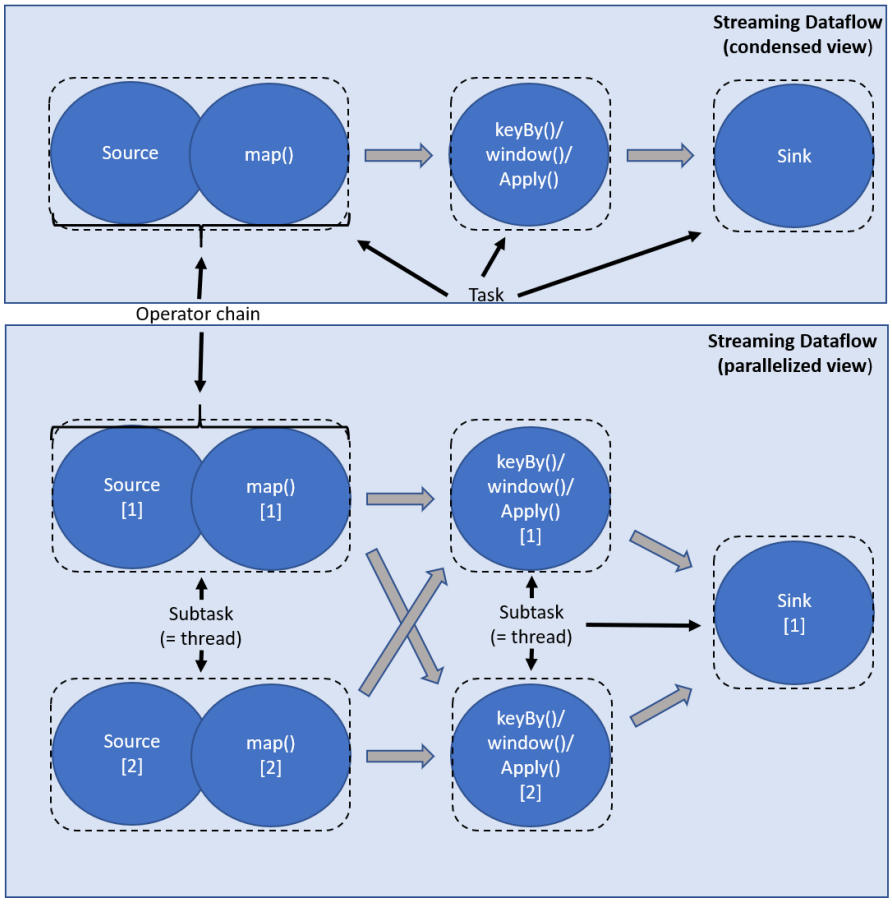

Figure 13: *Parallel data flow* [22].

As a distributed stream and batch data processing, Flink aims to parallelize and distribute processing. Hence, streams could be split in order to have one or more stream partitions according to the needs. The split of streams into partitions lead to one or more Flink operators subtasks for each Flink operator.

The Flink operators capability of dividing tasks and the fact that these tasks are independent of one another, makes possible the job distribution by different threads, machines or containers. To increase throughput and reduce latency, Flink has the capability of chaining the Flink operators subtasks into one task, reducing the handover and buffering of thread-to-thread (figure 13) [18], [22].

Since the dataflow can be sliced, there are two patterns in data transportation between Flink operators: one-to-one (forwarding) and redistributing. In one-to-one pattern, the receiver Flink operator sees the elements in the same order as they are outcoming by the sender Flink operator. This makes possible the optimization of chaining Flink operators subtasks as illustrated in the above figure (figure 13) with the *source* and *map* Flink operators. The redistributing pattern is showed between the *map* and *keyBy/window* Flink operators also, between *keyBy/window* and *sink*. This means that the sender Flink operator sends data to different targets or, in other words, changes the partitioning of stream. In particular case of (figure 13) the transformations performed are the *map, keyBy, window and apply.* But there are other transformations that could be done as mentioned in (listing 2).

2.2.5 Stateful Operations

In stream processing, frequently, Flink operators only consider the current event, without having to keep any kind of historical data. A good example is to emit an alert if the event value is above or below a given threshold. To emit the alert the Flink operator only has to look to the current value. Considering a use case of emitting an alert, if the last *n* values are above or below a given threshold, the Flink operator needs to remember the last *n* values to consistently emit or not the alert. This kind of operations are called stateful. Flink APIs already have some Flink operators, e.g. *window* that keeps information across events. This form of state is called *System state*. The other form of state is the *User-define state*, this kind of state allows developers to create stateful transformation functions such as *map* or *flatmap*, among others.

Concerning state, there are two kinds of states: The *Keyed state* and the *Operator state*. With non-keyed streams a stream that did not pass through a *keyBy* (the *keyBy* performs the stream partition by an attribute present in the stream or key) transformation, the Flink operator state is bounded to a parallel Flink operator instance, and it is possible to change the parallelism and to perform the redistributed state

through the parallel Flink operator instances. With keyed streams a stream that did pass through a *keyBy* transformation. Flink operator state is partitioned with one state-partition by key. Also there is a Key Group, the atomic unit to redistribute the Keyed State by parallel instance (figure 14) [18], [23].

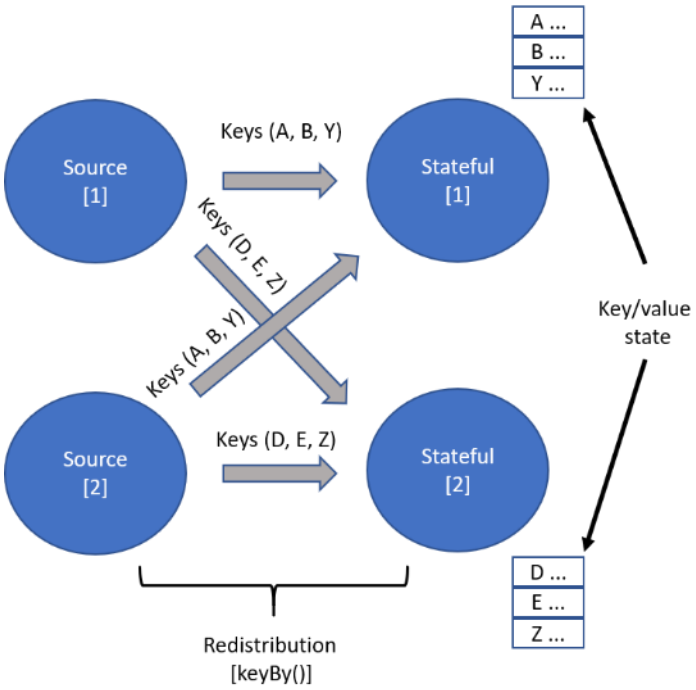

Figure 14: *Example of Keyed State* [18].

The usage of Keyed State goes through the access to *RuntimeContext*, that is available with *rich functions* (functions that provide additional four methods, *open, close, getRuntimeContext, setRuntimeContext*). The methods available to access state through *RuntimeContext* are as followed (listing 4).

```
ReducingState<T> getReducingState(ReducingStateDescriptor<T>)

ListState<T> getListState(ListStateDescriptor<T>)

FoldingState<T, ACC> getFoldingState(FoldingStateDescriptor<T, ACC>)

MapState<UK, UV> getMapState(MapStateDescriptor<UV, UK>)
```

Listing 4: *RuntimeContext available methods.*

As shown above (listing 4), all these methods receive as argument a *StateDescriptor* that holds the name for the state. Since it is possible the creation of multiple states, the name must be unique for futures references. Also, it holds the type of values for the state and possibly a user-specified function. The type of the *StateDescriptor* depends on the type of State.

Concerning the Operator State, to build a stateful function there are two possibilities. Both pass to implement interfaces: the general *CheckpointedFunction* interface or *ListCheckpointed<T extends Serializable>* interface.

Two methods need to be implemented in order to access the non-keyed state through the generic *ChecpointedFunction* (listing 5).

```
void snapshotState(FunctionSnapshotContext context) throws Exception

void initializeState(FunctionInitializationContext context) throws Exception
```

Listing 5: *CheckpointedFunction methods.*

The *snapshotState*() method is called to perform the snapshot. The logic of this method is to save all desired information in the state. The *initializeState*() method is called in the initialization of User-Define Function (UDF). However, there are two types of initializations that have to be considered: the actual initialization when the program runs for the very first time and the restored initialization, when recover from a checkpoint. Hence, the logic has to take care of both situations.

There is an additional method provided by the implementation of *CheckpointedRestoring<T>* function (listing 6).

```
void restoreState(T state) throws Exception
```

Listing 6: *CheckpointedRestoring method.*

This method is called every time that the recovery is necessary from a previous checkpoint.

The *ListCheckpointed<T extends Serializable>* interface is similar to *CheckpointedFunction* interface. The only difference, as the name infers, is that this function only supports list-style state (state store in lists) [23].

2.2.6 Checkpoints and State Backend

The concept of stream processing brings some issues, being the most problematic, the fault tolerance. When a failure happens in a stream application, some kind of guarantees have to prevail. One of the most important aspects it is to ensure that every record from the data stream is reflected exactly once. The fault tolerance mechanism used by Flink to ensure the recovery of state and consequently the application consistency is based on continuously draw snapshots of the data flow. For example, the Flink operators state and the distributed data stream, in other words checkpointing. Therefore, when a failure actually occurs, the system is able to fully recover from the early checkpoint, and to restore the state of Flink operators and the distributed data stream.

By default, Flink has checkpoints disable but the act of enable is trivial and it is reduced to call the method *enableCheckpointing(n)* on the *StreamExecutionEnvironment*. The *n* stands for the periodicity of checkpointing in milliseconds (listing 7) [24].

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// start a checkpoint every 1000 ms
env.enableCheckpointing(1000);
```

Listing 7: *Java example to enable checkpointing and configure time interval* [24].

There are other important advanced options that could be set to customize checkpoints according to the use case (listing 8) [24].

Exactly-once vs. at-least-once, these are the guarantees levels. Exactly-once is the most preferable, because ensures that records are reflected exactly once. For some applications, super low latency is more important than the exactly once guarantee and in these cases the at-least-once is the most pleasant mode. This mode is like a downgrade of the exactly-once where the alignment of checkpointing introduces less latency.

Checkpoint timeout, aborting time for the checkpoint in progress.

Minimum time between checkpoints, to ensure job progression between checkpoints, several times the time spent preforming the checkpoint can be more than the expected. Hence, it is better to define the *time between checkpoints* than the *checkpoint interval* to ensure job progression.

Number of concurrent checkpoints, to prevent the system from spending most of the time on checkpointing, by default this option is disable. There are no concurrent checkpoints, but in some use cases this could be an interesting configuration (pipelines with processing delay, still need frequent checkpoints to re-process less in case of failures). Note that this option cannot be used if minimum interval is defined.

Externalized checkpoints, this option allows the writing of checkpoints meta data out in persistent storage such Hadoop Distributed File System (HDFS). The cleanup process can also be configured (*retain on cancelation* and *delete on cancelation*). This means that, if chosen mode is *retain on cancelation* when a job fails, due to various causes, the checkpoint will still be in memory to be recovered. To resume a job from an externalize checkpoint via command line is trivial,

```
$ bin/flink run -s :checkpointMetaDataPath [:runArgs][25].
```

```
// set mode to exactly-once (default)
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);

// make sure 500 ms of progress happem between checkpoints
env. getCheckpointConfig().setMinPauseBetweenCheckpoints(500);

// checkpoints have to complete within one minute, or are discarded
env. getCheckpointConfig().setCheckpointTimeout(60000);

// allow only one checkpoint to be in progress at the same time
env. getCheckpointConfig().setMaxConcurrentCheckpoints(1);

// enable externalized checkpoints which are retained after cancellation
env. getCheckpointConfig().enableExternalizedCheckpoints(
       ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

Listing 8: *Checkpoints advanced options* [24].

Other configuration options are available. These can be performed via Flink configuration file (flinkDir/conf/flink-conf.yaml) and are all related to the state backend. In sum, where and how the state will be stored. The *state.backend,* can be configurated as *jobmanager*, *filesystem* or *rocksdb*. If the states size to store are minimal or in testing scenarios, *jobmanager* should be used. In all other cases *filesystem* or *rocksdb* should be

used, depending on the use case. The *state.backend.fs.checkpointdir* is a directory where checkpoints will be stored. The *state.backend.rocksdb.checkpointdir* is a directory to store *RocksDB* files. The *state.checkpoints.dir* is where the data from externalize checkpoints will be stored. This is the only option that can only be set via a configuration file. All others can be set in code. The *state.checkpoints.num-retained* is a maximum number of retained checkpoints (useful to fallback more than the last completed checkpoint, default one).

Depending on the chosen state backend the data structure of the storing data can be in an in-memory hashmap (*filesystem*) or in a *RocksDB* as key/value store (*RocksDB*) [22].

2.2.7 Job Managers, Task Managers and Clients

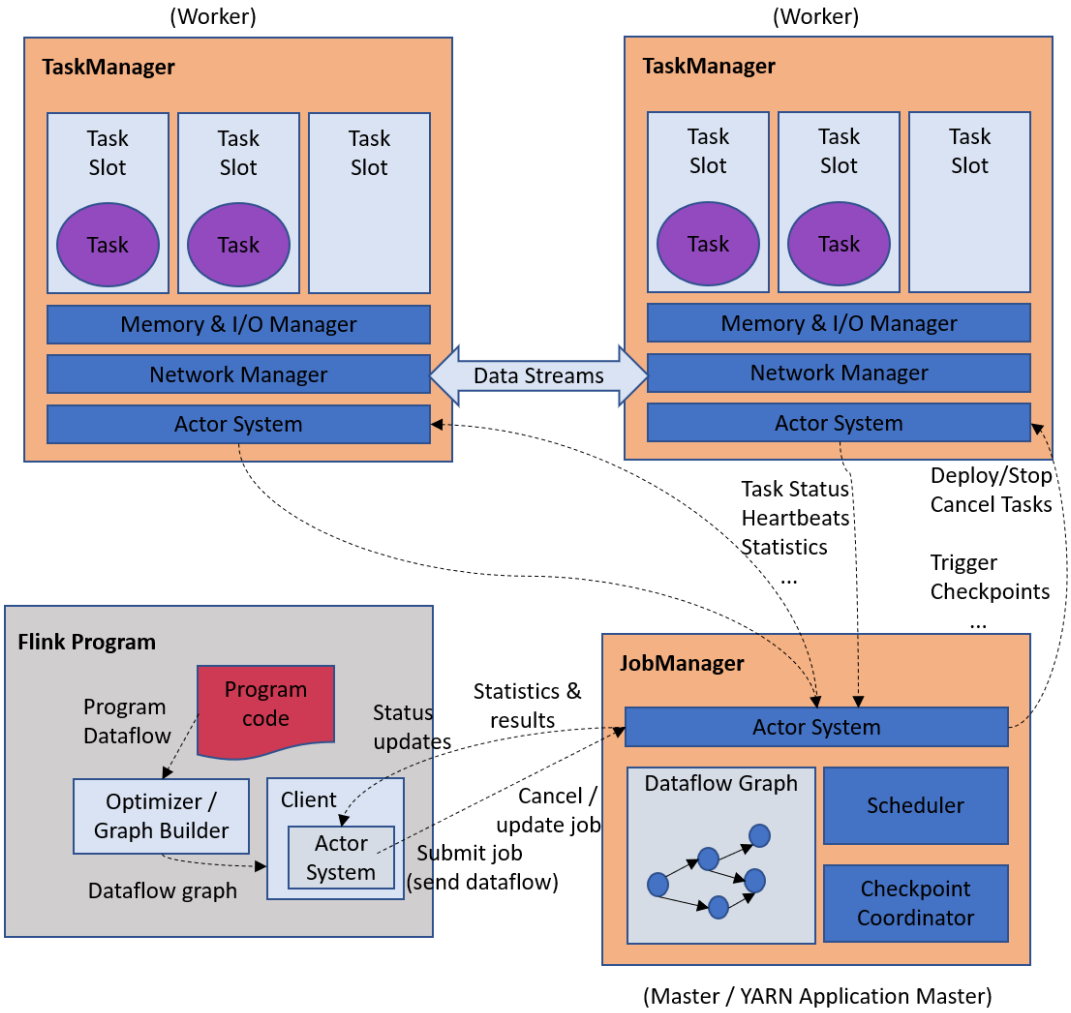

Figure 15: *JobManager, TaskManager and Client workflow* [22].

Job Managers and *Task Managers* are the different types of processes that compose the Flink runtime. As expected, the responsibilities of each one are totally different.

The *JobManager* (master) is responsible for the distributed execution coordination (checkpoints through checkpoint coordinator, schedule tasks through scheduler, recovery on failures among others). The minimum number of *JobManagers* is one. But in high-availability environment there are more than one. One is active as leader and the others are in standby mode. Hence, the *JobManager* can deploy, stop or cancel the tasks in *TaskManagers*, as well trigger the checkpoints. The coordination is done through Actor System using heartbeats (figure 15) [22].

TaskManagers (workers), a minimum of one worker must exist. This is because they have the responsibility of performing the execution of tasks.

There are multiple forms to start the above processes: locally (standalone cluster), in containers, or via *YARN* or *Mesos* resource frameworks.

The act of *Client* is to prepare and send the data flow to *JobManager*, *Client* is not part of the runtime and program execution. To trigger execution the *Client* could run as part of Java/Scala program or via command line process (flinkDir/bin/flink run …). So, the client is able to submit, cancel or update the job to the *JobManager* (figure 15) [22].

2.2.8 Task Slots and Resources

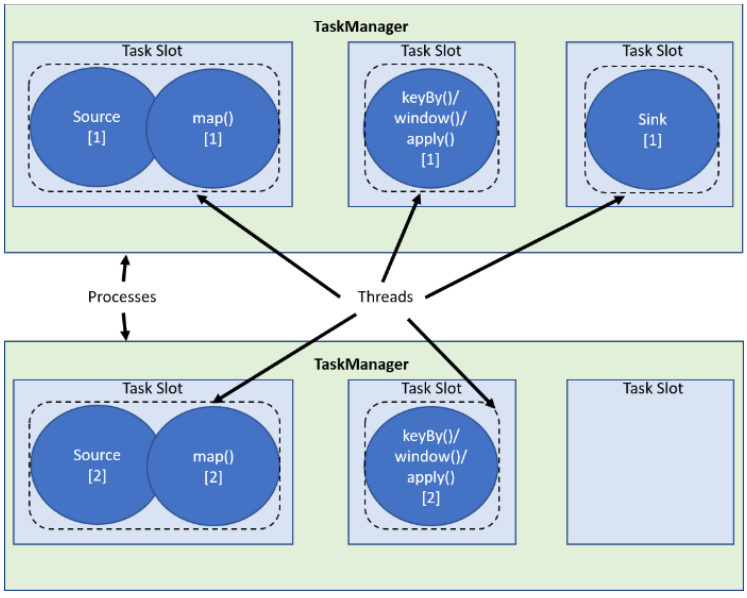

Figure 16: *Two TaskManagers with three task slots* [22].

Flink has a well-defined structure for slotting resources. All starts with the *TaskManager* (or worker) that is an independent *JVM* (Java Virtual Machine) process.

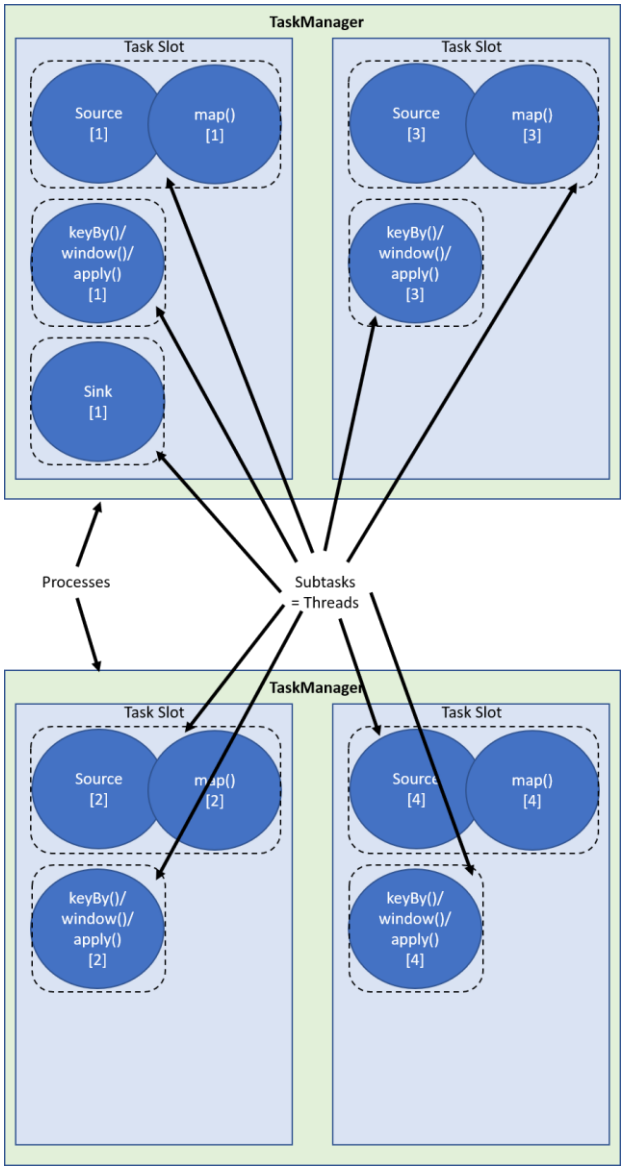

Figure 17: *Subtasks sharing task slot* [22].

Each *TaskManager* is composed by task slots, at least one (Flink allows users to specify both *TaskManager* and task slots numbers). Task slots have the responsibility to control the number of tasks that each *TaskManager* accepts, since users could define the number of task slots. This means that it is possible to perform subtasks isolation and each subtask will run in a separated thread of the *JVM* process. Also, the available

resources of *TaskManagers* are even split by all task slots that compose the *TaskManager*.

This kind of slotting prevents subtasks from different jobs to compete from resources as they have the task slot reserved resources (figure 16) [22].

Considering the tasks from the same job (program submitted to the *JobManager*), it is possible for subtasks (including of the different tasks) to share task slots (figure 17) [22]. In the particular case of (figures 16, 17) the transformations performed are the *map, keyBy, window and apply.* But there are other transformations that could be done as mentioned in (listing 2).

As illustrated above (figure 17), allowing the task sharing can lead to a task slot to handle an entire pipeline. Task sharing also brings two main benefits: first the number of task slots that a Flink cluster needs are the same of the maximum parallelism; Second brings better resource utilization (subtasks have different weight and without slot sharing some leads to task slots with different resource utilization), as shown (figure 17) the heavy subtasks (*keyBy/window/apply*) are fairly distribute.

2.2.9 Save Points

As mentioned in Checkpointing section, Flink programs written in DataStream API with externalized checkpoints are able to recover from failures. Save points are identical, with the difference that they have a manual trigger instead periodic triggers. Hence, save points are manually triggered checkpoints. This means that users can manually trigger a save point and perform some kind of update in a program or in a Flink cluster without losing state.

Users can create a save point from the command line with a simple command as shown in (listing 9) to trigger save point or, the command showed in (listing 10) to cancel a job and trigger the save point. Resuming from a save point is trivial and it is illustrated on (listing 11) [22], [26].

```
$ bin/flink savepoint :jobId [:targetDirectory]
```

Listing 9: *Command to trigger save point.*

```
$ bin/flink cancel -s  [:targetDirectory] :jobId
```

Listing 10: *Command to cancel job and trigger save point.*

```
$ bin/flink run -s :savepointPath -n [:runArgs]
```

Listing 11: C*ommand to resume from save point.*

2.3 Kafka

LinkedIn engineers with the exponential growth of their membership community, have to deal with more than 10 billion messages every day [27]. Initially, LinkedIn started, as expected, "*nice and simple*" with a monolithic architecture (figure 18) [28].

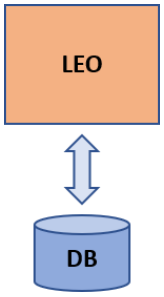

Figure 18: *Representative scheme of* monolithic architecture *of Leo and DB exchange* [28].

To solve the problem of the increasing traffic, LinkedIn engineers had to break Leo "*kill Leo*" into stateless and functional services. This leads to a service oriented architecture. The number of services increased from 150 in 2010 to 750 in 2015. However, for example in the case of LinkedIn, there was a hypergrowth, consequently there was a need to increase the scale factor. To do that, LinkedIn engineers started to introduce more layers of cache. Nevertheless, the introduction of caches brings other problems, like the increase of complexity around invalidation. So, to allow the horizontally scale, and decrease cognitive load, they had to position their caches closer as possible to the data store and reduce latencies.

From the necessity of having multiple pipelines for streaming and queuing data, like data flow to data warehouse or store batches of data into Hadoop among others, Kafka emerged (figure 19).

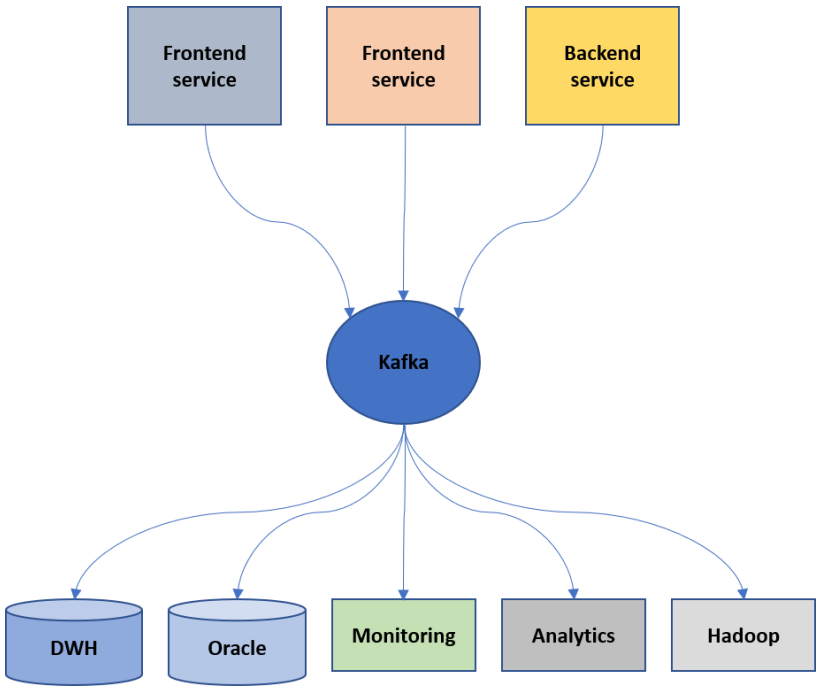

Figure 19: *Scheme of Kafka as the universal data stream broker* [28].

Kafka is a distributed streaming platform, a publishing/subscription system that allows the publishing and subscription of stream of records, like a message queue system. The stream of records are stored in categories also known as topics [29].

2.3.1 The Topic

Topics or categories are where the records (the information) are published by the producers, allowing Kafka to keep a partitioned log (figure 20).

Each topic can have multiple subscribers, through this zero or more consumers (information readers) are allowed to consume records in a topic. All consumers have their own reader offset that can be controlled by them. This allows consumers to handle records in the order they want to (figure 21).

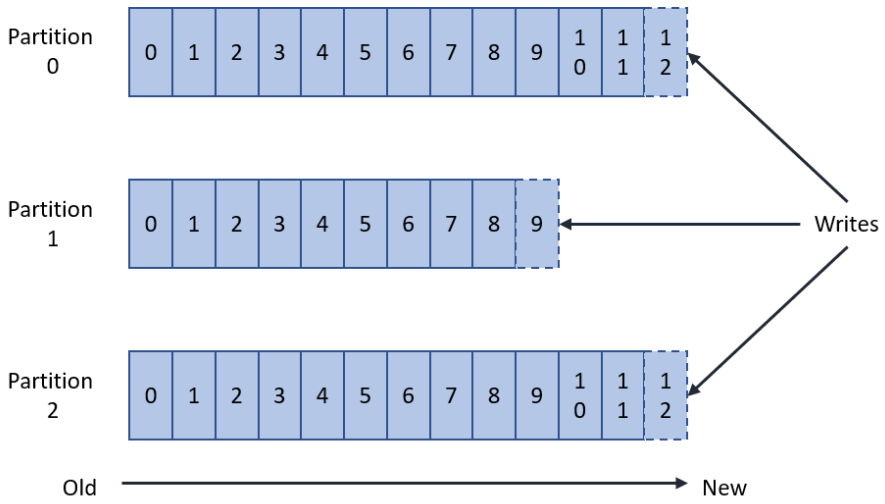

Figure 20: *Anatomy of a topic* [29].

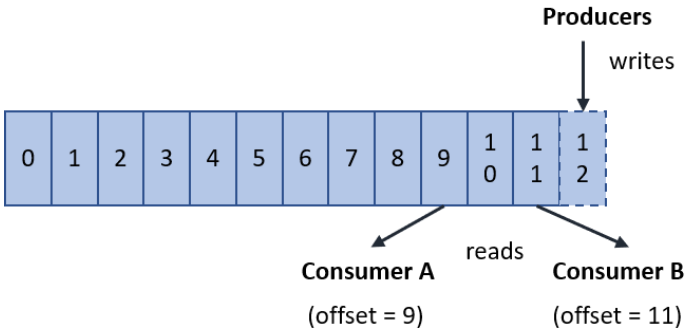

Figure 21: *Representative scheme of producers and consumers records over a topic* [29].

2.4 Remote Authentication Dial In User Service (Radius RFC-2865)

One of the most important key aspect to a Telecommunication company is to deal with the continuous increasing number of mobile users, and the respective administrative support such security, authorization and accounting. For this it is extremely important to keep a centralized users database. Not only to ensure a better service to clients but also to prevent loss to the Telecommunication companies.

Radius RFC-2865 is the protocol used to exchange messages between client users and Telecommunication company servers. These messages allow the accounting, authorization and other services configurations. In this exchange, a client/server model

is used and a network access server (NAS) operates as a Radius client. In message exchanging the NAS has to pass to Radius server all the necessary users information, and to act according with the response. Radius server has to receive the users connections, authenticate them and return the necessary information, according to user request.

2.4.1 Format

Radius packets are encapsulated in the UDP (User Datagram Protocol) data field, one Radius packet per UDP data field.

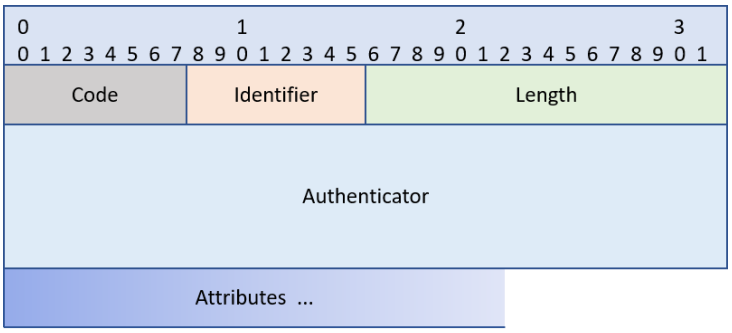

Figure 22: *Representative scheme of Radius packet* [30].

```
1     Access-Request
2     Access-Accept
3     Access-Reject
4     Accounting-Request
5     Accounting-Response
11    Access-Challenge
12    Status-Server (experimental)
13    Status-Client (experimental)
255   Reserved
```

Listing 12: *Radius messages types.*

As showed in the figure above (figure 22), the first field in Radius packet is the code with one octet, that defines the type of message. There are nine possible types (listing 12).

The next field is the Identifier with one octet. This field is used by Radius server to detect duplicate requests.

After comes the length, this field has two octets and it is used to indicate the packet total length (including, code, identifier, length, authenticator and attribute fields). Following up is the Authenticator field with sixteen octets. This is used to authenticate Radius server reply and password hidden algorithm. The previous mentioned fields are the mandatories fields that makes the twenty octets the minimum length of Radius packets.

The attribute field contains a list of attributes in the format showed below (figure 23). As mentioned before, the number of attributes could be zero or at maximum the max length of the Radius packet (4096 octets). The length of the Radius packet indicates the end of the attribute list. There are 63 standard attributes types, and a lot more types that are vendor-specific.

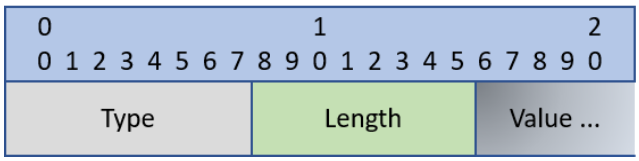

Figure 23: *Representative scheme of Radius attribute* [30].

In this dissertation we will not go through more Radius packets details. It is to note that there is a lot of geographical and temporal information available in the attribute list [30]. For this work the main information are three BTSs id's as well the distance from the BTSs to the mobile device.

2.5 PostgreSQL

Postgres was created by Michael Stonebraker, professor at UCB (University of California at Berkeley). The development starts in 1986 as a follow-up of Ingres, being the reason for the name Postgres (like after Ingres). Between 1986 and 1994 the fundamental idea to developers was to explore new concepts and technologies relative to "object relational". Illustra was the commercialized version of Postgres, brought by Informix. Later at 2001, IBM also brought Informix for one billion dollars. Initially,

Postgres has its own query language POSTQUEL. In 1995 two Ph.D. students from Stonebraker labs replaced POSTQUEL language per an extended set of SQL query language. This change conduces to a system rename, Postgres95. The new phase of Postgres95 started in 1996 when it started the open source adventure. This was achieved thanks to a group of developers that saw all the system potential, and for another eight years the development of Postgres continued. This group of developers gave great contribution at multiple levels such consistency and uniformity to code base, detailed regression tests to improve quality, mailing lists for bug reports and also added new fundamental features. After all this, the system was considered rock solid stable, with all these changes and with the start of open source reality, the system took its current name PostgreSQL.

Nowadays PostgreSQL goes on version 9.6 and it is still growing. Thanks to major companies such Afilias and Fujitsu, who use it and made serious contributes to the continuous development of PostgreSQL. This makes PostgreSQL one of the first choices. In fact it is extremely difficult to find one corporation or government agency which have not used PostgreSQL in their information system solution [31].

2.5.1 PostGis

The necessity of dealing with geometry, geography and other types, led to development of PostGIS. PostGIS extension brings functionalities to PostgreSQL that make it possible to handle with all special types like other normal type. This makes PostgreSQL management system more powerful, fast and robust to deal with this kind of special types. Since PostGIS was released under GNU General Public License (GPL), there was an open source software that respect the "Simple Features for SQL Specifications" by "Open Geospatial Consortium's" [32].

2.6 Zeppelin, Jupyter Notebooks and HTML

2.6.1 Zeppelin

In 2012, NFLabs [33] with the ambition to create an analytic tool data, the Peloton was born. This product was made for commercial use. After this, in 2013, NFLabs released Zeppelin as an open source feature from Peloton. With the increasing receptivity of Zeppelin, in 2014 come the opportunity to go at the global level, with Apache [34], [35]. Nowadays, Apache Zeppelin is a multiple-purpose notebook, not only a data analytic tool, also data ingestion, data discovery and data visualization and collaboration tool [36]. Almost any language could be plugged into Zeppelin. This is

achieved thanks to Apache Zeppelin interpreter concept. Zeppelin already supports some interpreters such Apache Spark [37], python [38], jdbc [39], markdown and shell among others. Beside this, users could create a new interpreter if desired.

For visualization propose, Apache Zeppelin already has some charts to visualize queries results. For this dissertation, it has not enough, so the addition of new charts is necessary to permit maps and geographical visualization.

2.6.2 Jupyter

From the IPython Project, in 2014 emerges the open-source project Jupyter [40]. This is a web application to create and share documents that can contain live code, equations and visualizations. There are multiple use cases such as, data visualization, data cleaning and machine learning, among others.

More than 40 programming languages are supported by Jupyter such as, python, R and Scala. The IPython kernel allows the use of multiple maps related to libraries such as Folium [41], Ipyleaflet [42] and the GoogleMaps [43]. Due to this, Jupyter is a strong tool to handle geographical visualization.

2.6.3 HTML

HTML (Hyper Text Markup Language) is a markup language to develop web pages. Consider HyTime (Hypermedia/Time-based Document Structuring Language) a structured language that is a base line to patterned hypertext. And SGML (Standard Generalized Markup Language) that aims the standardization language of documents in a way that the machines could interpret them. HTML is like the join of these two patterns.

When *Tim Berners-Lee* a British physicist that first introduced HTML, the goal was to solve their own problems. Problem related with the share and communication between researches on their workgroup. At this point, HTML has like a collection of tools. Now HTML stands in version 5 with well-defined rules.

2.7 Related Work

Location based services (LBS) bring several contributions to a lot of fields. Also, it is possible to extract geographical information from multiple sources. Such, LBSN, mobile networks and many other applications. On this study, the geographical information will

be extracted from a mobile network. Assuming that almost everyone has a mobile phone, makes this a valuable and reliable resource for location-based services.

C. Lin and M. Hung [44] showed that it is possible to extract location-based information from mobile devices. In their study the geographical information was extracted from the Global Positioning System (GPS) and "WI-FI" in case of GPS blind spots. Once the location is known, this information is used to alert users when they are near of one of their previous defined tasks. This task reminder is triggered by location instead of time.

T. Buda and I. Ireland [45] showed how to determine urban patterns from LBSN geographical information, in order to comprehend mobility behaviors. These solutions are powerful once they allow to understand crowds-mobility. In a catastrophe scenario this is a value resource for all kind of protections services. Other usages to these solutions are the traffic management, urban plaining, among others.

Mobile users locations are a valuable resource. As *M. Al-Rajab, S. Alkheder and S. Hoshang* study [46] showed that it is possible to use this geographical information from the mobile GPS in order to increase efficiency. The work done in this study is aimed to suggest users the less congested petrol station. Through mobile users locations, one can know which petrol station has more congestion. Due to this, it is possible to forward users to another station. The outcome is an increase in efficiency and a decrease on the delays.

In sum LBS solutions bring major contributions. As a result, the work done in this dissertation aims to process mobile network records in a reliable, scalable and fault tolerant way. From these records, geographical information will be extracted through enrichment. With this information, it will be possible to determine mobile clients presences on defined areas.

3. Framework

In this chapter the detailed implementation of the framework will be presented.

3.1 Requirements

Extracting geographical meaningful information about generic users daily life in a stream processing environment brings some challenges.

First of all, the sources of the streaming are files that continuously arrive to HDFS (Hadoop Distributed File System) from the mobile network. Also, it is extremely important to ensure privacy and anonymity. So, the data is previously anonymized using *one-way-script-obscure-function*. When the data arrives to HDFS, already comes anonymized. From the solution point of view, it is exactly the same. Since, the geographic information to extract from the data is about generic mobile clients. Hence, there are multiple sources that produce files. Only in a utopic world this process happens without network delays, or other kind of delay introduced in creation or delivery of these files. Due to this, the incoming of out of order files into HDFS is probably high. So, it is imperative that the application takes this in consideration. This is the reason why when dealing with users geographical information, in most cases to keep consistency in the results, earlier events must be treated first. On the other hand, the application has to maintain a throughput as higher as possible. Hence, it is extremely important to keep a fair contract between the solution for out of order incomings and the throughput higher as possible. Also, the result of the stream processing should be available for consumption as soon as possible to maximize the approach to "Real Time".

The users flows present in files do not hold all necessary information, so enrichment is mandatory. There are multiple attributes present for this solution but the used ones are: *userId (previously anonymized), timestamp, cellTowerOneId, distanceToCellOne, cellTowerTwoId, distanceToCellTwo, cellTowerThreeId, distanceToCellThree*. With additional information, in particular the position of the cell towers, it is possible to triangulate. This means to find the device position from the three positions of cell towers and the respective distance (figure 24). With the intuit to perform the enrichment and have the mobile users position.

When there are files continuously arriving to source stream, the application must display a high availability environment and working 24 hours/7 days. But there are multiple types of failures that can happen. These can be external to the application such as, cluster framework or machines failures. Other aspect is that from time to time, some kind of upgrade or update to the system must happen. When this happens, the application will be interrupted, at the same time it must have the capability of fail

recovery and resume processing from the previous point. Also having the guarantee that there is no reprocess of early processed events and, with the consistency of the previous computed results. Flink already provides some stateful operators such as, window and sources, among others. However, to perform a more complex computation it is necessary to implement multiple transformations. In most cases these transformations have to look to events that already happened. Here all transformations must be implemented with state to ensure a stateful application.

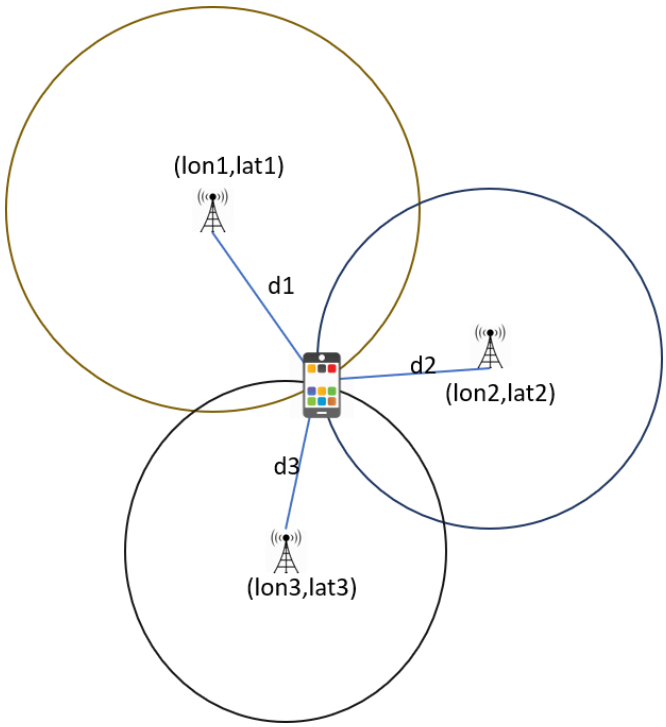

Figure 24: *Triangulation example.*

Considering that client users (solution users) have interaction with the application by submitting areas into the areas stream, the system must keep track into this situation and make reliable computation. So, when an area is inserted, all metrics related to these areas must be calculated and, stop them when the area is removed.

Relatively to the output results, it is necessary to maintain a good relation between out writer interval time and the size of the data to write. Reasons being that for a client user that query results, users expect that these results will be most updated. So, large intervals of time between write out are inviable. On the other hand, HDFS is

not efficient with small files, therefore it is important to establish a good relation between both aspects.

3.2 Architecture

3.2.1 Overview

Relatively to the rule "Divide and Conquer", it is possible to spit the problem and scale out the solution in multiple fundamental transformations. The implementation of the independent transformations leads to a final solution less tricky and complex.

After the implementation of all transformations, it is possible to link all together as unique dataflow.

In order to achieve the final solution, it is necessary the implementation of the following transformations (parts of the problem), not necessary in this same order:

- **Ensure that the solution is able to handle out of order incoming files,** implementation of a time buffer in data stream source.
- **Perform data enrichment,** using additional information (cell tower position) calculate the anonymous mobile user position (triangulation).
- **Correlate users and areas with country zones,** this correlation will be helpful to deal with parallelized and distributed processing. In this way it is possible to ensure that users and areas in the same zone are treated by the same parallel instances.
- **Correlate users with areas,** the effective metrics calculation.
- **Outcomes the results,** write results into HDFS in an efficient way.

3.2.2 Initial Version

At the beginning of the study, it was thought to build the solution from scratch. The plan was to have some Java modules which are the receptors of UDP Radius packets from the mobile network. These receptors after parsing, the packets were forward to a Kafka topic. Another Java module was consuming the Kafka topic to perform the data enrichment followed by the insertion into HDFS and PostgreSQL with the GIS extension. The aim was to store and visualize the data in "Real Time" and then apply Machine Learning algorithms to infer knowledge (figure 25).

Due to some external difficulties, it was impossible to obtain the data (Radius packets from the network). For this reason, it was decided to implement a Java simulator to produce Radius packets. On other words, replace the packet creator by a simulator.

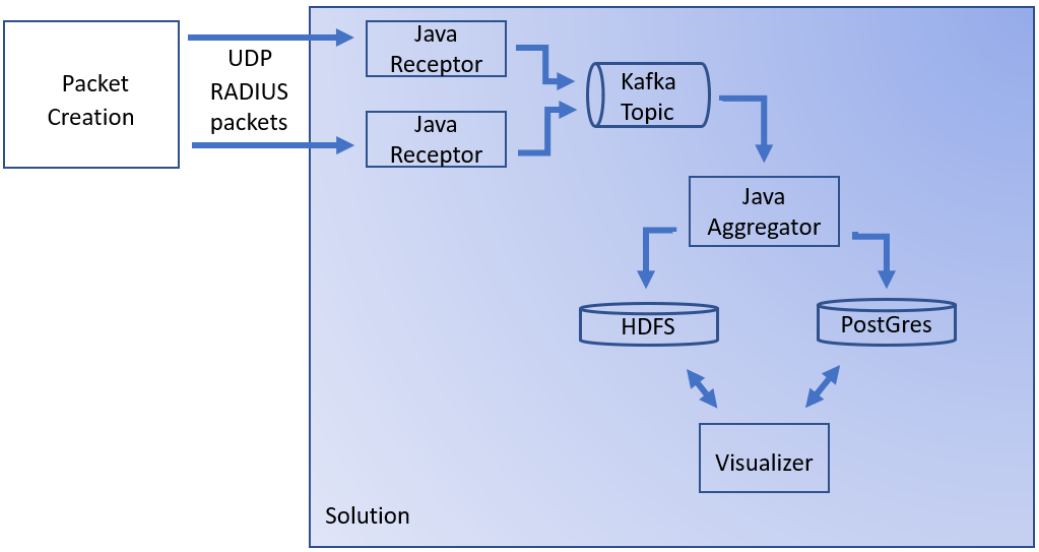

Figure 25: *Overall architecture of initial solution.*

Initially it was thought to use *Zeppelin* for visualization. However, the display of geographic information as points on a map was not a feasible solution. Due to this, *Jupyter* was tested has an alternative notebook. This one has multiple libraries to work with maps, but the problem was the latency to display a large number of points. It could be a good solution when working with a few hundred points, however this was not our case. In (figure 26) is showed the result of the Jupyter visualizer.

Due the latency to display points in Jupyter notebook, it was decided to change to HTML5 to develop the web visualization tool. This visualization shows on map a moving heatmap of the users positions. Hence, client users are able to walkthrough time and see people movements as a moving color heatmap (figure 27).

Since the work was done with simulated packets (simulated positions), it was no longer possible to perform machine learning over the data. Also, the throughput achieved was not the desired. So, it was thought to change the paradigm and the use cases of the solution. This was to use a big data tool to perform the processing oriented to the calculation of metrics, related with users and areas as describe in the next section Final Version.

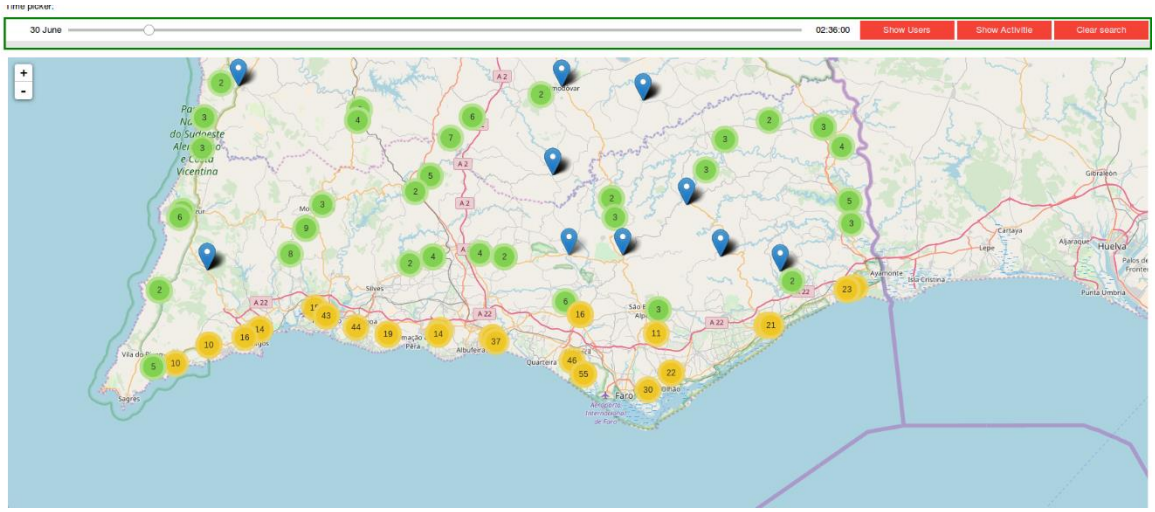

Figure 26: *Jupyter web visualizer.*

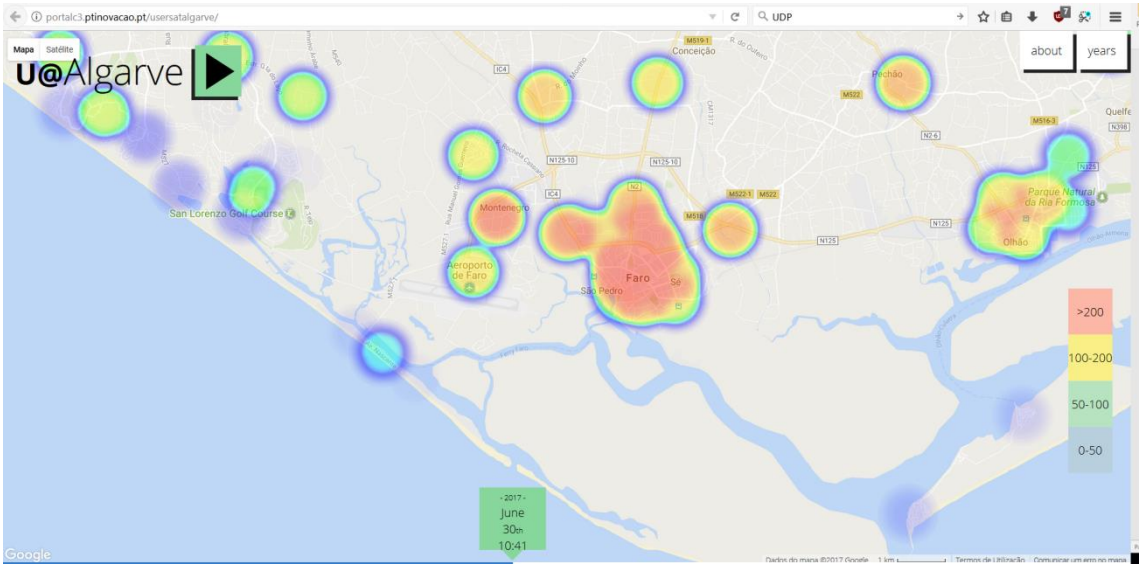

Figure 27: *HTML web visualizer.*

3.2.3 Final Version

As an initial approach and as a proof of concept, a batch solution was implemented. In this solution using Flink DataSet API, the goal was to work over all files that already come until the moment that the user query for results. The query involved to lunch a Flink batch job and as mentioned before in section 2.2.2. The processing was

made by a bounded data set. For each submitted job, among other parameters, the user provides the desired areas and the result has the metrics to all provided areas (figure 28).

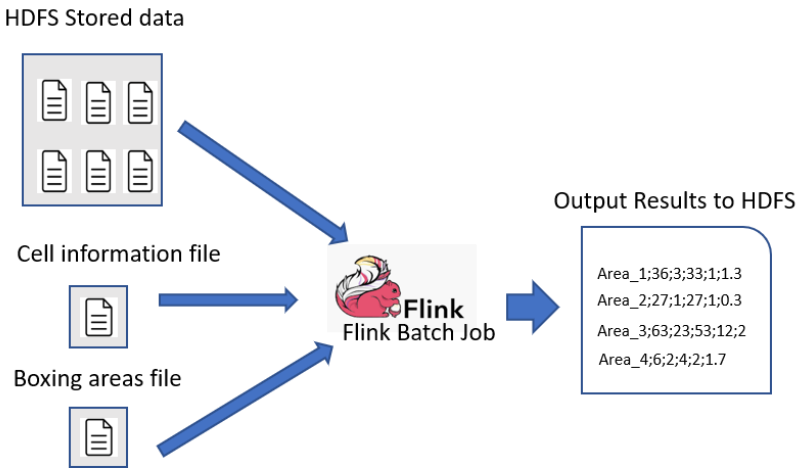

Figure 28: *Overall Batch Architecture.*

With a batch solution, several concerns that existed in a streaming environment, no longer exist such as, state for fault tolerance, out of order incoming files and outcome results. Relatively to fault tolerance, the approach to deal with a failure, since the job has a starting point and an ending point, it is to reprocess the bounded records. Since the data set is bounded, the out of order problem is solved using a *sort* Flink operator to order records by time and, to perform a consistent computation. Regarding the writer also it is resolved due to the same reason (bounded data set) because the processing is made to all files. In the end, just one result needs to be written (figure 29).

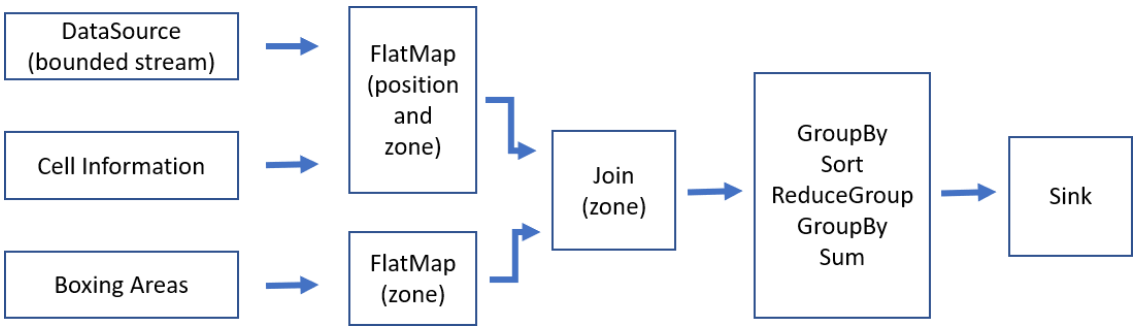

Figure 29: *Detail Flink Batch Job.*

As illustrated in the above (figure 29), the main components to perform the batch job are:

- *Flatmap*, receives the bounded data stream (mobile network flows) and the Tower Cells Information. This has a *HashMap* (cellId and position point) and computes by triangulation the user position and the users country zone. To note that a *map* transformation (for parsing) in Cell information occurs before the *flatmap* followed by the conversion from *DataSet<Tuple2<Integer, Point.2D>>* to *HashMap<Integer, Point.2D>,* the conversion is only possible in DataSet API. The Boxing Areas *flatmap* performs the parsing and computes the country zone.
- *Join,* performs the *join* between the data flow and areas flow by the same country zone.
- *GroupBy,* the first *groupBy* aggregates then joins flows by userId and areaId.
- *Sort,* order the aggregate result by timestamp to perform a consistent computation.
- *ReduceGroup,* iterate over the aggregate result and increment or not counters.
- *GroupBy,* the second *groupBy* aggregates dataset by areaId.
- *Sum,* the last transformation is to perform the final count by each area.
- *Sink,* write results into HDFS.

The major concern was to have a throughput as higher as possible, since batch is a finite job, as soon as done the clients could have their results.

After the conclusion of this previous version the decision was to keep them as complement of a future application. Making the same processing to give the same results over the historic data set. Application users are also able to see what happen over a desire area in the past.

The improvement of the batch solution, as expected, goes to switch the API from DataSet to DataStream API. The effective solution is not so trivial, as mentioned above and in Overview section there are some concerns when treating unbounded data streams.

So, the final solution must be able to handle late incomings, recover from failures (stateful and fault tolerant) and have an efficient writing among all the other requirements (figures 30, 31).

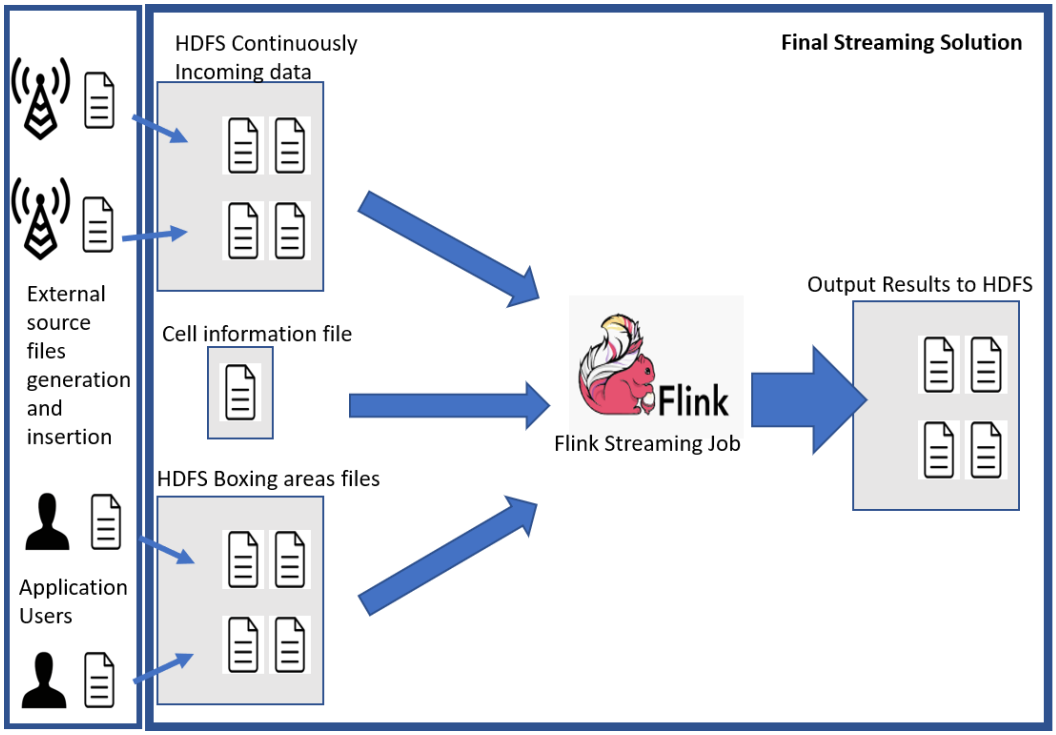

Figure 30: *Overall Stateful Streaming Architecture.*

In the streaming solution, it is necessary to continuously monitor the HDFS directory to look for new incomings and forward them to the data stream. At this point, it is necessary to handle with late incomings. The boxing areas have the same logic when a solution user desires to add or remove areas. Solution users send the information to HDFS directory (as proof of concept). The idea is that the information about areas, from solution users arrives at Flink via *Kafka* topic. Flink already provides *Kafka* connectors that handles the data injection and sink. Due to technical constrains, and as a proof of concept, it was decided for now to continuously read a HDFS directory. Since DataStream API is being used, cell Information must be treated like a streaming.

As shown (figure 31), the first transformation (Parsing and Time Buffer) applied to data source performs the data parsing. The timestamp extraction, watermarks generation and the buffer window are responsible to handle late arrivals as well the ordering by timestamp. After this, the stream is connected with the broadcast cell information stream to make a *CoFlatmap* transformation that computes the mobile users position and country zone (Enrichment). The areas stream goes through a *Flatmap* transformation to perform the parsing and calculate the country zone (Parsing and Enrichment). Then the *KeyBy* (by country zone) transformation is applied in both streams. Later, they are connected to perform the *CoFlatmap* transformation that will

compute the desired metrics about each area (Correlate). Finally, the result is sinking via HDFS connector *BucketingSink* to HDFS directory.

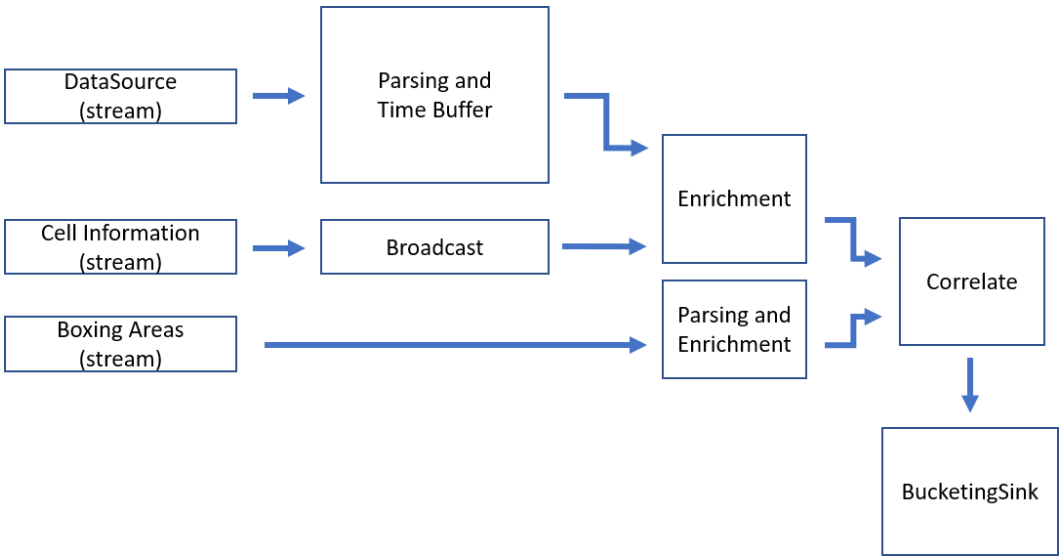

Figure 31: *Detailed Flink Streaming Job.*

3.3 Implementation

In this section a more detailed view of the streaming solution implementation, plus the solution components will be presented. In the sections above, Overview and Flink Version, the solution components were briefly described. However, in this section, since the batch implementation was a proof of concept, the focus here will be about effective final solution, the streaming implementation.

3.3.1 Out-of-Order Incoming Files

The effective data gets into Flink streaming job via a Flink function *readFile* as a continuous streaming (listing 13).

```
readFile(fileInputFormat, path, watchType, interval, pathFilter)
```

Listing 13: *ReadFile function.*

The *readFile* function receives multiple arguments: The *watchType* set with "FileProcessingMode.PROCESS_CONTINUOUSLY", allowing users to continuously monitor one directory. This way, Flink divides the reading process in two sub-tasks: one for directory monitoring and the other for data reading; The *interval* (time in milliseconds) is the interval of time that Flink takes to checking for new incoming files; Also, it is possible to give a *pathFilter* to exclude some files from being read if necessary.

The *readFile* is already a stateful function (System state) although it is not able to handle out-of-order files. Once it is required, to ensure the order in incoming records, a time buffer must be implemented after the records get into Flink, as illustrated above (figure 31). This was the first dataSource transformation. This transformation will be described in detail in this section (figure 32).

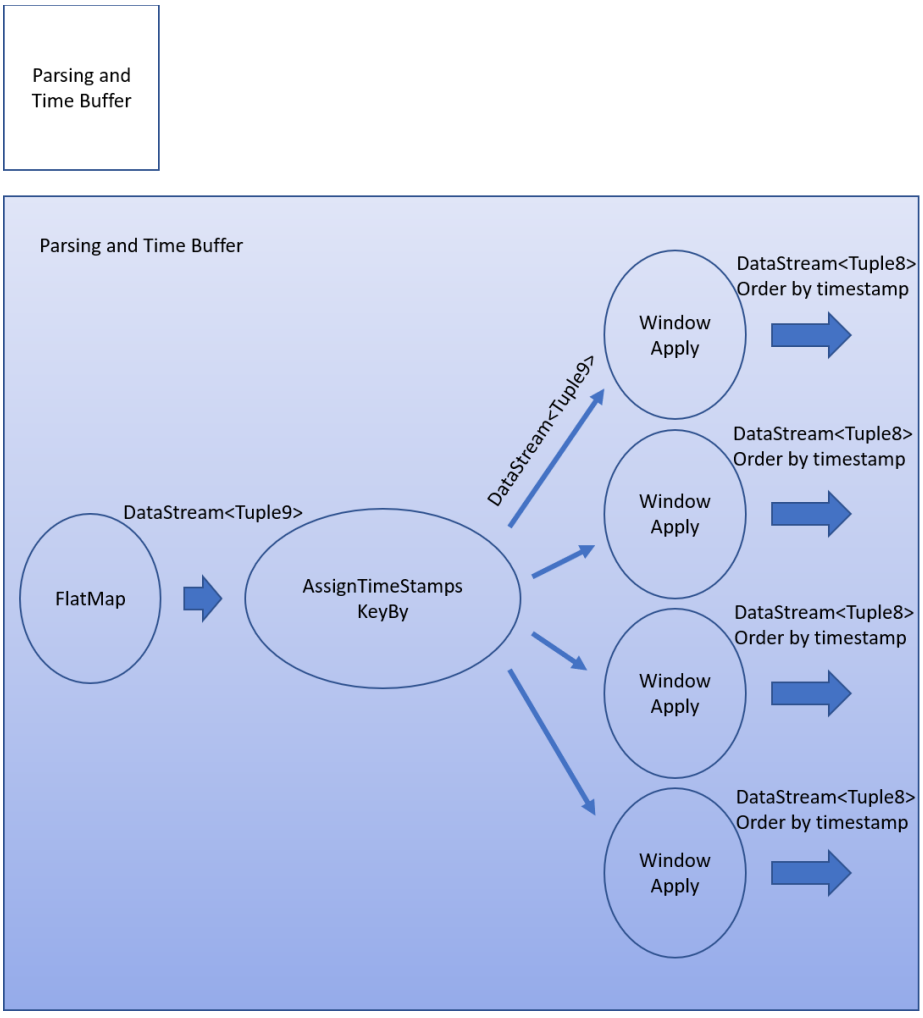

Figure 32: *Detailed Parsing and Time Buffer.*

Flatmap, receives the stream (*DataStream<String>*) from the *readFile* and performs the parsing. This transformation only looks to the current record, at this point a string. Then extracts the necessary values and forwards them as a Tuple with the respective attributes to the data stream. Among the necessary attributes, another one is appended to Tuple that will be used in *KeyBy* transformation to perform the partition on the stream. This attribute contains an Integer between zero and three representing the quarter of hour to which the record belongs. This is a parameterized value and it is related with the window size. Since the transformation is only based in current record, this function does not need to maintain state.

AssignTimeStamps, is a user defined function responsible to extract the timestamp from the records and generate watermarks. This is a necessary transformation due to the time characteristic used in *event time.* The watermark is generated taking care of the late events. To do that, the user defined function receives as argument the time about the delay of the event. So, the closure of the windows will have this lateness in consideration. This way, if a file containing late events and the late time is inside the defined lateness they will still be processed as expected. Obviously, if the lateness is greater than the defined the records are dropped, a good relation between late arriving time and latency must prevail.

KeyBy, as mentioned above, this transformation splits the stream into four partitions (first, second, third, fourth quarter of an hour, for this particular case). This way the Flink window operators will only actuate over fifteen minutes of records. If the window is defined to five minutes it will lead to twelve partitions. As mentioned above this is a parameterized value.

Window/Apply, this window function, as already mentioned, is closed after the time defined plus the lateness time. Hence, in the user defined *window apply* function after the window is closed, the records from this window are ordered by timestamp. With this, it is possible to guarantee that the earliest records are treated first and solved the out-order problem.

The times for the window, lateness and division of stream (*KeyBy* transformation and effective window size) are all parameterized in this case. It was decided to use fifteen minutes for stream division as well to window size and forty-five minutes to lateness. This way it is possible to ensure that a Flink window operator closes his window before it starts to receive records from the next hour (no process of records having the same quarter of hour and different hours).

3.3.2 Data Enrichment

The enrichment of the data goes through the calculation due triangulation (figure 24) of the user position fall back on the cell tower information. The cell information on a streaming environment needs to be treated as a stream like all the others. It is not possible, like in the batch environment, to convert them into an *HashMap*. So, the only way to correlate the two streams (data source and cells information) was through the use of the *connect* function followed by *CoFlatmap* function (figure 33).

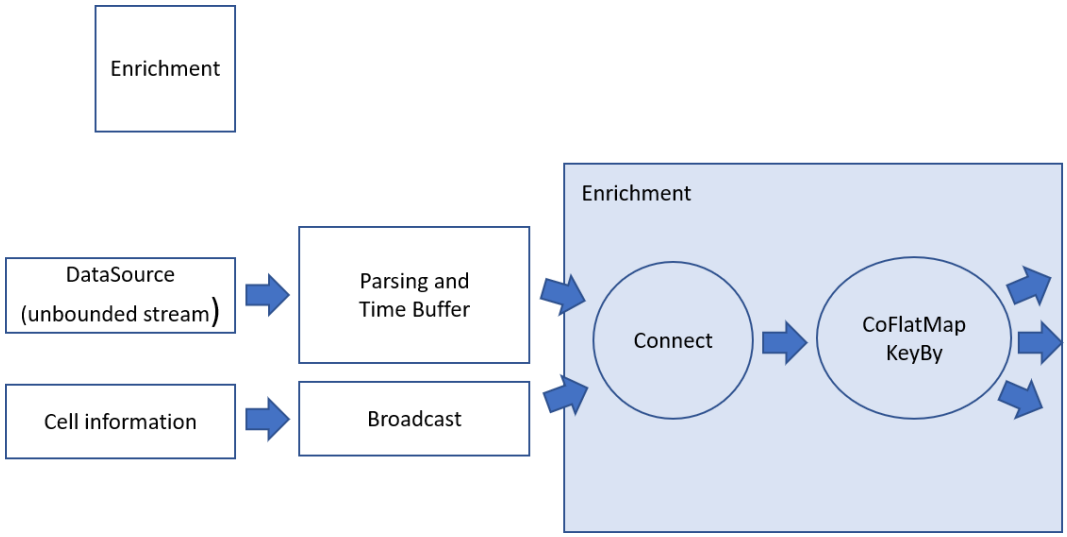

Figure 33: *Detail view of the Enrichment transformation.*

The first thing to do on cell information is to broadcast them to all Flink operators. This way, all the Flink operators have the same information about all the cells towers. With the broadcast done, it is time to connect the cell information stream with the data stream that resulted from the previous transformation describe in section 3.3.1 (*DataStream<Tuple8<userId, timestamp, cellId1, distCell1, cellId2, distCell2, cellId3, distCell3>>*).

With the two streams connected, it is possible to perform the process, using *CoFlatmap*. This is a user defined function similar to *Flatmap* that has two *flatmap* methods instead of one. One of these methods (*flatmap1*) receives the cell information as a stream of strings. Then performs the parsing and stores the information into a *HashMap<Integer, Point.2D>* (cellId, position). The other (*flatmap2*) receives and performs the enrichment of the effective data stream. This enrichment accesses the cells towers *HashMap* to get the effective position. With the three positions and respective distances it is possible to compute the user position, using triangulation (listing 14).

```
/**
* @params p1, p2, p3 (longitude, latitude) of the cell towers.
* @params d1, d2, d3 distances from the cell towers to the intersection point.
* @return the intersection point (longitude, latitude).
*/
public  Point getIntersectionPoint(p1,p2,p3,d1,d2,d3){
        Point[] twoCircleIntersectPoints = new Point[2];
        twoCircleIntersectPoints = getTwoCirclesIntersections(p1,d1,p2,d2);
        if(getDistanceBetweenPoints(p3, twoCircleIntersectPoints[0]) == d3)
                return twoCircleIntersectPoints[0];
        else
                return twoCircleIntersectPoints[1];
}
```

Listing 14: *Pseudo code to perform triangulation.*

Once the position is determined, it is possible to map the users with previous defined country zones. This mapping will help latter to treat users and areas from one zone into the same Flink operator for consistency purposes.

Since, it is necessary to keep the cell information in a *HashMap*, the *CoFlatmap* function must implement the *CheckpointedFunction*. This function provides two more additional methods (listing 15).

```
snapshotState(FunctionSnapshotContext context)

initializeState(FunctionInitializationContext context)
```

Listing 15: *CheckpointedFunction methods.*

The logic in *snapshotState* function is to store the *HashMap* that keeps the cell information into state. The *initializeState* is responsible to initialize the state descriptor and get the respective state. This function must test the context in case of a restoring context. To fill the *HashMap* with the information of the checkpointed state. Through this the function it is now stateful and fault tolerant.

With this implementation, it is now possible to change the parallelism and therefore making the application scalable. The choice of type of state redistribution is

very important. For instance, *Union Redistribution* should be chosen in order to scale up. This will allow all Flink operators to receive the same state information in case of context restoring (from a checkpointor save point), due to an increase of parallelism. The last operation is a *KeyBy* to split the stream. This division is made taking care of the country zones.

3.3.3 Correlate Users and Areas with Country Zones

The users, as mentioned in the above section, are mapped with the respective country zone, after their position is known in *CoFlatmap* function that performs the enrichment. Relative to areas, the mapping with the country zone is the first transformation applied to areas stream and this transformation is a regular *Flatmap*. The logic used to determine area zones is similar to the one used to correlate users and zones. With the difference that it needs to be computed for all zone corners. This is because at the area limit, it can belong to four country zones. In this case, the same area is collected for each zone.

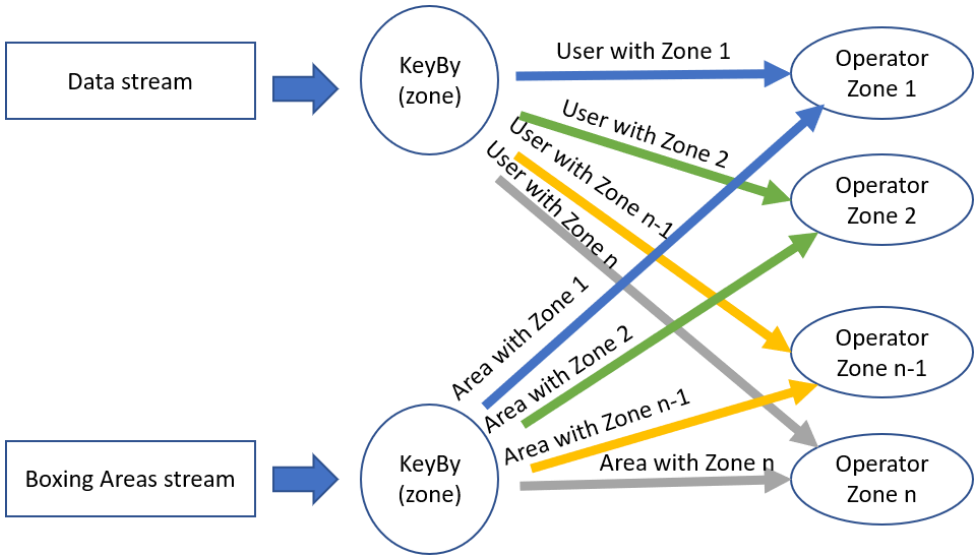

Figure 34: *Streams partitions by zones.*

This *Flatmap* transformation performs the processing taking care only of the current record. This way, it is not necessary to keep a state for recovery purposes.

After the *Flatmap* transformation, the next operation is a *KeyBy* to perform stream split by country zone like it was done in data stream after the user zone calculation. The aim

is to ensure that users and areas with same zone are treated by the same parallel Flink operator (figure 34).

In this particular case, and by default, the Portuguese map is divided into ten zones (figure 35).

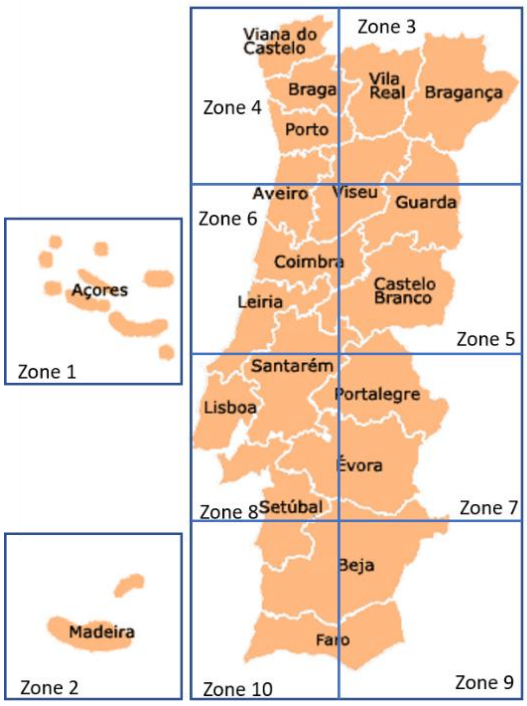

Figure 35: *Default country division by zones.*

There are two parameters to define the final number of zones. The number of division levels and the number of divisions zones. So, in this case (default), the level number is one, and there are eight divisions zones. Once these values are parameterized, it is possible to increase the number of zones. For example, if the number of levels is defined as 2 and the number of division zones stay in 8, each one of the default zone will be split into 8 sub-zones. So, in the end it leads to 80 zones.

The goal of this implementation is to make possible the correlation between zones and parallel Flink operators. This way, it is possible to adjust the application parallelism with the number of zones and vice versa. This leads to a substantial increasing of efficiency.

3.3.4 Correlate Users with Areas

After splitting the streams (data stream and areas stream) by zone, it is time to connect the keyed streams to perform the effective computation using a *CoFlatmap* user defined function (figure 36).

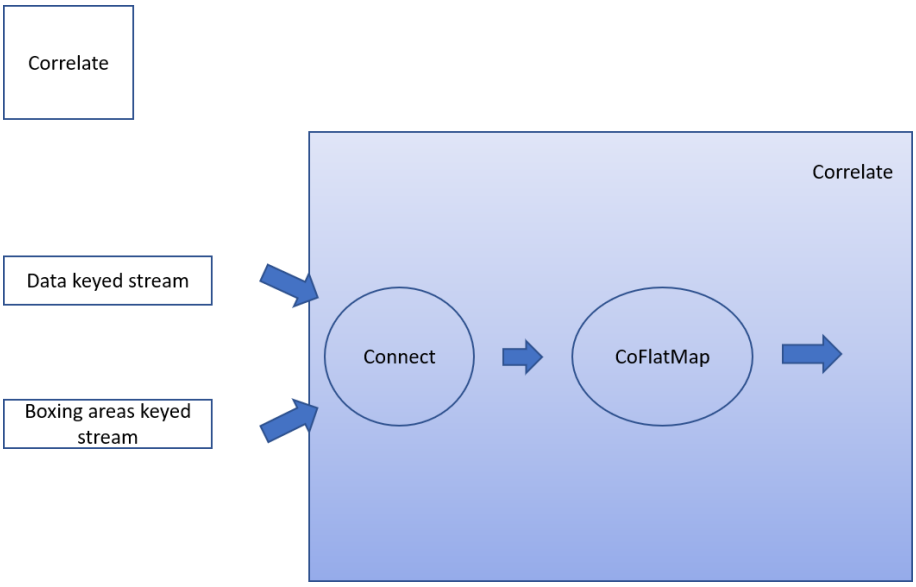

Figure 36: *Users and areas correlate transformation.*

As mentioned above, the *CoFlatmap* is a variant of the *Flatmap* but with two *flatmap* methods to treat each stream. One Flatmap is responsible to treat the areas stream and perform the insertion of new areas into areas *HashMap*, or the areas removal from the *HashMap*. An auxiliary *HashMap* is used to maintain users historic over each area.

The other *flatmap* has the responsibility to handle the keyed data stream. When receives an event, a check is made to see if the user is or not in all areas of the same zone.

If the user is inside an area, another test is done to see if it is the first time. In case of a first time, a *List<Tuple3<Long,Long,Long>>* is created to hold the user presences on area. The *Tuple3* holds check-in time, intermedium time and checkout in this order. After the creation of the list, a *Tuple3* is inserted with check in time. If the user already appears in this area, by checking the last *Tuple3* it is possible to infer if the user last time was inside or not the area. Depending on that, a *Tuple3* update or insertion is done. There is one *HashMap<Integer,List<Tuple3<Long,Long,Long>>* for each area containing the presence list for each user.

If the user is outside the area, one check is done to see if there is some earlier presence on this area. In affirmative case, the checkout time is inserted on the last *Tuple3* containing the early presence. In case, the last *Tuple3* has a checkout time, meaning that the user already made the checkout, nothing is done. As well if user never was on this area.

The areas metrics recalculation and respective outcome result is done every time that one of the *HashMap<Integer,List<Tuple3<Long,Long,Long>>* belong to an area is updated, or when an area is removed.

The necessity of keeping historic information (the *HashMap's* mentioned above) to perform a consistent metrics calculation forces the implementation of a stateful user defined function. This implementation is achieved using *Keyed State.* This is possible since both streams (data and areas) are keyed streams. Further, it is not necessary to implement additional functions to handle the state. This implementation is managed with the *Rich* function (extending the RichCoFlatmap function) that provides methods to access the state (*getRuntimeContext(), setRuntimeContext(), open() and close()*). With this implementation it is also possible to scale up or down the Flink solution.

3.3.5 Outcome Results

To perform the sink of the results into HDFS, Flink provides *Connectors*. The Hadoop connector used is the *BucketingSink*. This allows to specify the type of the *Bucketer* as well as other important configurations such *BatchSize*, *PartPrefix*, *inactivityTime* and *PendingPrefix*, among others. Once the writing is done on a continuous streaming, the type of *Bucketer* used is a *DateTimeBucketer("yyyy-MM-dd--HH")*. This leads to a new directory containing all part files by hour. It is possible to define the maximum size for the files using the *BatchSize* attribute. There is another important attribute that could be configured. The *inactivityTime:* if defined, a file is closed every time there is a time of inactivity on stream equal to the defined value.

With all those parameters, it is possible to ensure a good relation between file size (Hadoop is more efficient handling big files) and the delay time to have the last file available for consumption. In other words, files as large as possible in a minimum amount of time.

4. Evaluation

This chapter aims to perform the evaluation of the solution and briefly discuss the results.

4.1 Proof of Concept

The application goal was to perform a scalable, reliable and fault tolerant, metric calculation about people presences in geographical areas over a continuous stream. At this point the network data flows (source data) were still unavailable so, all the source data was simulated. Hence, the tests done at this point were for testing if the metrics had a consistent count. Other tests were performed to understand which kind of files showed the best throughput on Flink and some aspects of Flink configurations.

Relative to scalable and fault tolerance the tests done are to understand if the application is coherent in case of failures. Also, if it is possible to stop the application for scaling proposes (scale up or down due the needs).

4.1.1 Constrains

Due to the unavailable data from the network, it was necessary to simulate files containing flows with the identical attributes. To have a coherent user position, a random point is computed using the Cell Tower Information and the country map stored on PostGis DB. To do that, a random cell tower is determined and a random point is computed near to this cell tower. After the point is determined, a test is done using the Portugal map stored into PostGis DB to confirm if this is a valid point. With the valid point, it is possible to determine the other two nearest cell towers. In the end, a flow is made containing *userId, timestamp, cellTower1Id, distanceToCellTower1, cellTower2Id, distanceToCellTower2, cellTower3Id, distanceToCellTower3.* A similar logic was used for the areas files.

4.2 The Environment

There are various scenarios due to all the constrains inherent to the stream processing:

- Fault tolerance, reliable metric calculation after recover from a failure.
- Scalable, possibility to stop application execution and change the parallelism with coherent results.

- Out-of-order, ensure that the result is the same independent of order.
- Late incomings, consistency even when deal with late incomings files.

The above scenarios are external scenarios not controlled by the application. However, the application must be able to handle them. Relatively to the reliable metric calculation performed by the application, the tricky scenarios are about people movement over time:

1. A person can get into an area and stay there.
2. A person can get into an area and leave multiple times.
3. Various persons can get into an area and stay there.
4. Various persons can get into an area and leave multiple times.
5. Above tests with overlapping areas.

Since these tests are concerned with the reliability of the results, they were performed with small data sources contemplating all above scenarios.

Concerning Flink performance, the goal of the tests performed was to understand the best option relatively to the file type that will be consumed. To perform these tests and to have more reliable results the decision was to use the batch solution. The main reason for this was that the batch job is a finite job. Hence, the time spent to realize the job was more coherent.

To understand the best configuration on Flink, some tests were done with the batch solution (for the same reason). In this case with the same input and changing the number of Vcores by YARN container to different configurations of Task Managers and Task Slots.

To comprehend if the solution handles fails or is able to scale up or down: tests are done to simulate failure and restore execution from the checkpoint. The objective is to confirm if the result is the same as if there was no interruption. In the scale test, the parallelism is increased and decreased when the execution is interrupted.

All tests are executed over a cluster managed by YARN. The cluster has three nodes with 25 Gb of memory and 16 virtual cores (Vcores).

4.3 Testing

The aim of the functionality tests performed was to understand if the application developed accomplishes the purposed. To do that, a simulation of the tricky scenarios described in section 4.2 was done. The main objective was to be sure that metric calculation is coherent.

4.3.1 Scenario 1

The scenario reflects one selected area (rectangle over Forum Aveiro). And one mobile user that get in this area and never exit (figure 37).

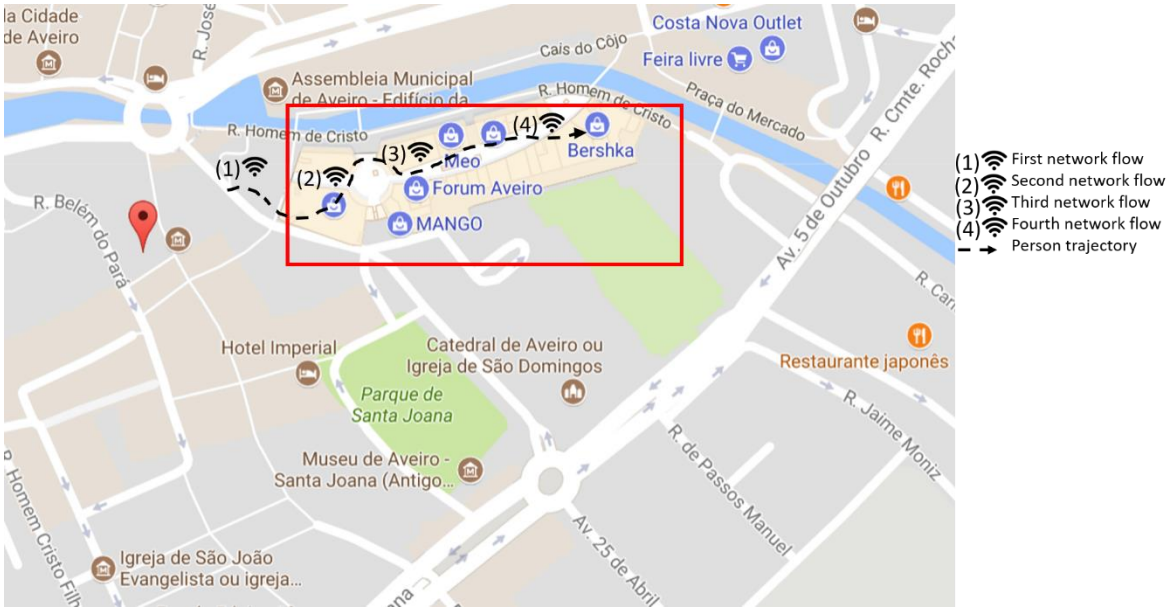

Figure 37: *One user appearance scenario.*

This scenario simulates a user that gets into a defined area and never exits from there. The result expected to this situation is just one presence in the area.

The result in HDFS (table 2):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
2	Zona_4	1	0	1	0
3	Zona_4	1	0	1	0
4	Zona_4	1	0	1	0

Table 2: *Result scenario 1 one user.*

As showed in the result from HDFS the count is according with the expected.

The same scenario was also tested for multiple persons. In this particular case, nine persons appeared only one time into the area. So, the expected result has nine total entries, zero total outs, nine distinct entries and zero distinct outs.

Results on HDFS (table 3):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
2	Zona_4	2	0	2	0
3	Zona_4	3	0	3	0
4	Zona_4	4	0	4	0
5	Zona_4	5	0	5	0
6	Zona_4	6	0	6	0
7	Zona_4	7	0	7	0
8	Zona_4	8	0	8	0
9	Zona_4	9	0	9	0

Table 3: *Result scenario 1 multiple users.*

4.3.2 Scenario 2

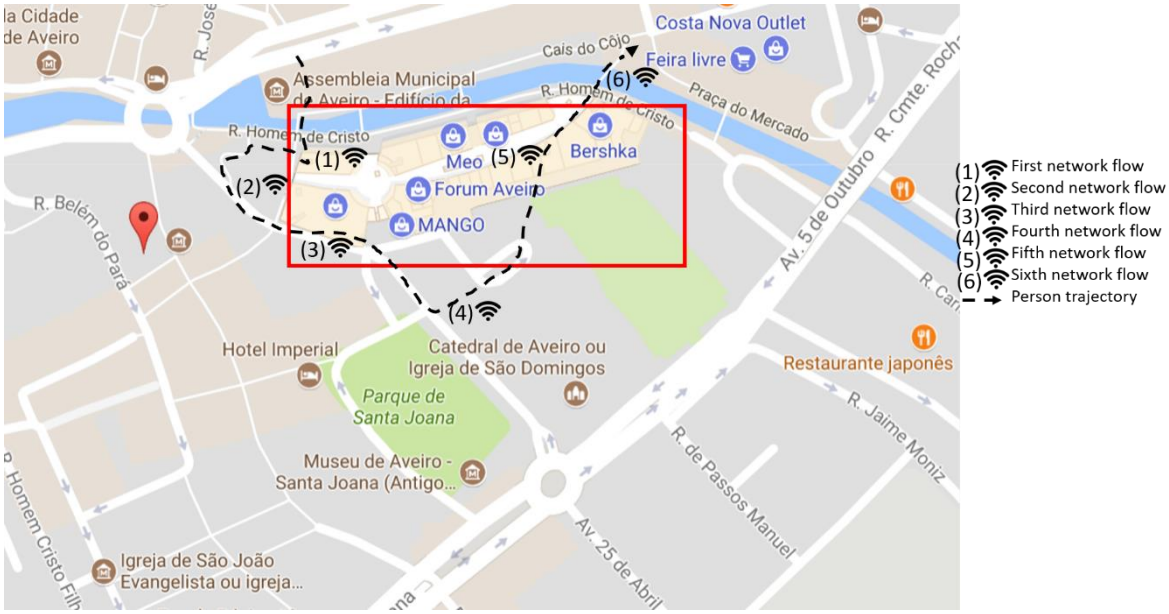

Figure 38: *One user multiple appearance and exits.*

This scenario shows one area (rectangle over Forum Aveiro). And one mobile user with multiple entries and exits over the area (figure 38).

In this scenario the person trajectory appears in the area three times with exits between the appearances. The result expected is one distinct person with three appearances and three exits from the area.

Results on HDFS (table 4):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
2	Zona_4	1	1	1	1
3	Zona_4	2	1	1	0
4	Zona_4	2	2	1	1
5	Zona_4	3	2	1	1
6	Zona_4	3	3	1	1

Table 4: *Result scenario 2 one user.*

As expected the result showed three ins and outs of the same person, just one distinct person.

This test was also done with multiple persons, in this particular case with seven persons that get in and out of the area.

Results on HDFS (table 5):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
2	Zona_4	2	0	2	0
3	Zona_4	3	0	3	0
4	Zona_4	3	1	3	1
5	Zona_4	4	1	4	1
6	Zona_4	5	1	5	1
7	Zona_4	5	2	5	2
8	Zona_4	6	2	6	2
9	Zona_4	6	3	6	3
10	Zona_4	6	4	6	4
11	Zona_4	7	4	7	4
12	Zona_4	7	5	7	5
13	Zona_4	7	6	7	6
14	Zona_4	7	7	7	7

Table 5: *Result scenario 2 multiple users.*

The result shows the seven persons get in and out as well the distinct same values.

4.3.3 Scenario 3

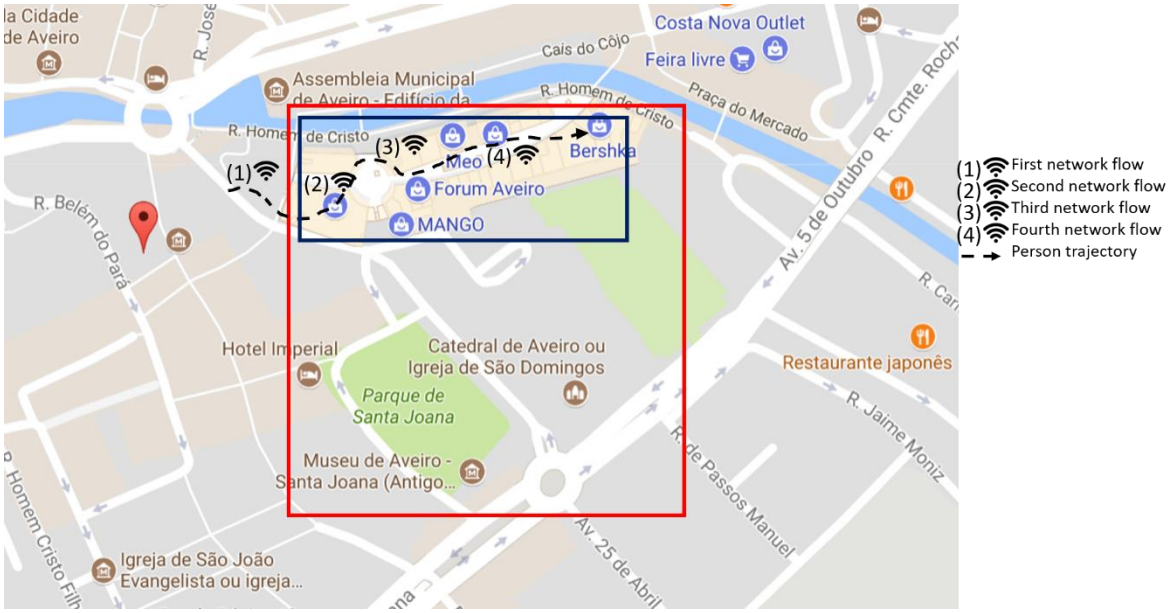

Figure 39: *Overlapping areas with one entry from one Person.*

On this test the goal is to be sure that the result is coherent considering two overlapping areas.

In this case one person that gets into both areas and never leaves (figure 39).

The expected result from this scenario is the same values from both areas, in this particular case one entry in both areas.

Result on HDFS (table 6):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
2	Zona_4	1	0	1	0
2	Zona_5	1	0	1	0
3	Zona_4	1	0	1	0
3	Zona_5	1	0	1	0
4	Zona_5	1	0	1	0
4	Zona_4	1	0	1	0

Table 6: *Result scenario 3 one user.*

The same test was done to multiple persons that only appeared once, in particular nine persons that get into both areas once.

Results on HDFS (table 7):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
1	Zona_5	1	0	1	0
2	Zona_4	2	0	2	0
2	Zona_5	2	0	2	0
3	Zona_4	3	0	3	0
3	Zona_5	3	0	3	0
4	Zona_5	4	0	4	0
4	Zona_4	4	0	4	0
5	Zona_4	5	0	5	0
5	Zona_5	5	0	5	0
6	Zona_4	6	0	6	0
6	Zona_5	6	0	6	0
7	Zona_5	7	0	7	0
7	Zona_4	7	0	7	0
8	Zona_4	8	0	8	0
8	Zona_5	8	0	8	0
9	Zona_4	9	0	9	0
9	Zona_5	9	0	9	0

Table 7: *Result scenario 3 multiple users.*

4.3.4 Scenario 4

This test aims to handle with the two overlapping areas and one person that gets out from one area staying in the other (figure 40).

In this particular case, the expected result is to have one area (the largest area) with only one appearance and the other one with multiple appearances, plus exits and just one distinct in and out.

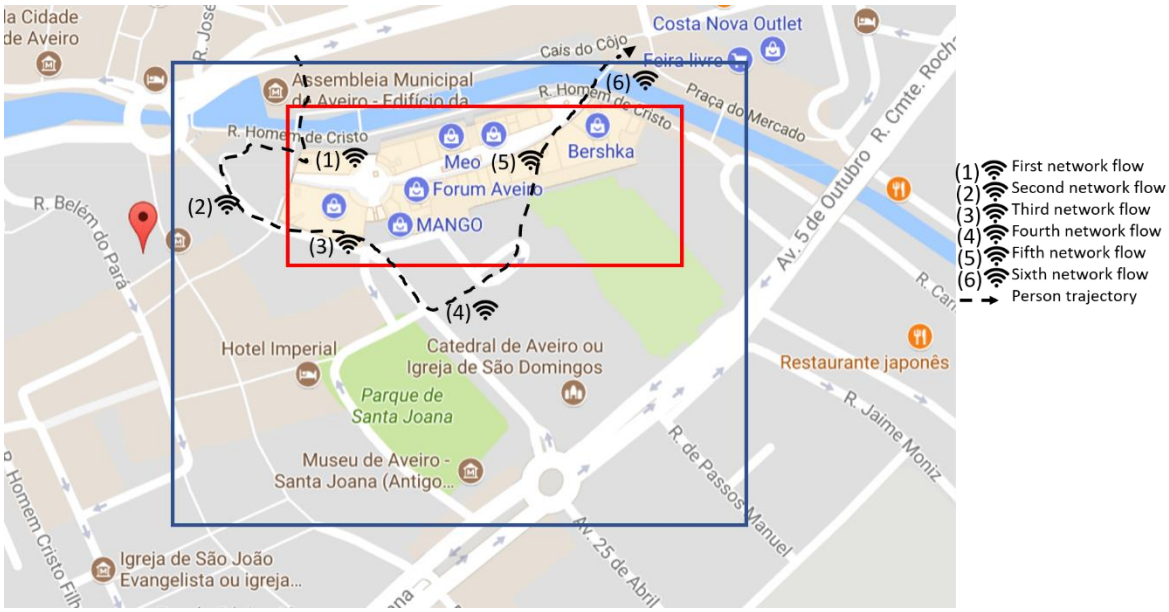

Figure 40: *Two overlapping areas and exits from one of them.*

Results on HDFS (table 8):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
1	Zona_5	1	0	1	0
2	Zona_4	1	1	1	1
2	Zona_5	1	0	1	0
3	Zona_4	2	1	1	1
3	Zona_5	1	0	1	0
4	Zona_4	2	2	1	1
4	Zona_5	1	0	1	0
5	Zona_4	3	2	1	1
5	Zona_5	1	0	1	0
6	Zona_4	3	3	1	1
6	Zona_5	1	0	1	0

Table 8 : *Result scenario 4 one user.*

4.3.5 Scenario 5

In the last scenario there are persons that get in and out from both overlapping areas (figure 41).

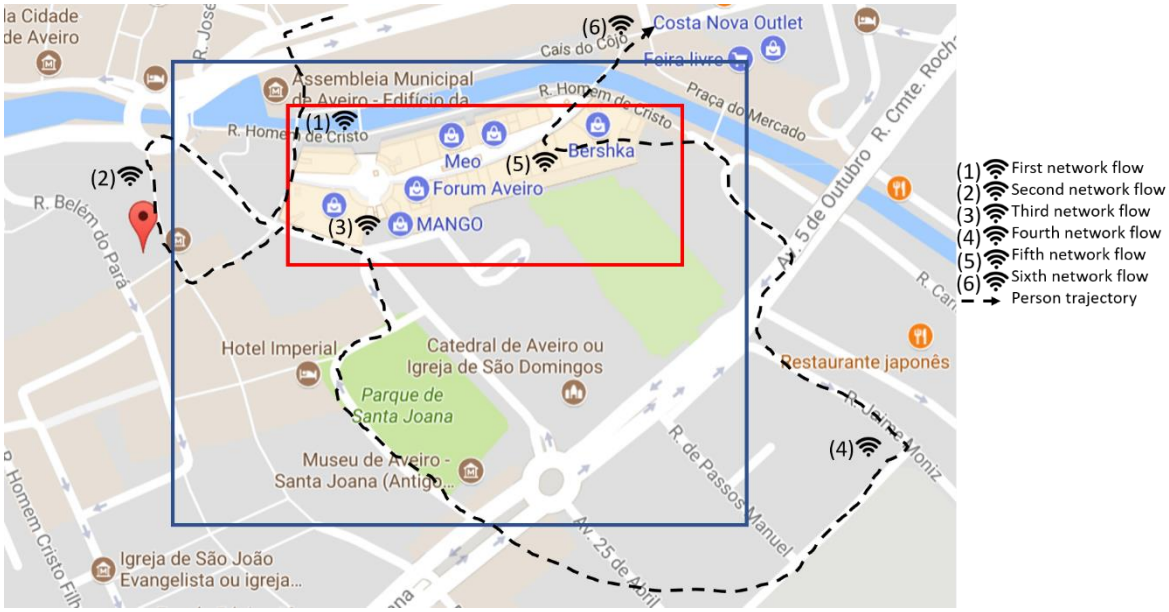

Figure 41: Multiple ins and out from both overlapping areas.

Considering just one person the result expected was multiple appearances and exits with only one distinct in and out.

Results on HDFS (table 9):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
1	Zona_5	1	0	1	0
2	Zona_4	1	1	1	1
2	Zona_5	1	1	1	1
3	Zona_5	2	1	1	1
3	Zona_4	2	1	1	1
4	Zona_4	2	2	1	1
4	Zona_5	2	2	1	1
5	Zona_5	3	2	1	1
5	Zona_4	3	2	1	1
6	Zona_4	3	3	1	1
6	Zona_5	3	3	1	1

Table 9 : *Result scenario 5 one user.*

The same scenario changing the number of persons, first with seven persons that get in and out from both areas. The result expected is seven appearances and seven exits with same distinct values.

Result on HDFS (table 10):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
1	Zona_5	1	0	1	0
2	Zona_4	1	1	1	1
2	Zona_5	1	1	1	1
3	Zona_5	2	1	2	1
3	Zona_4	2	1	2	1
4	Zona_4	2	2	2	2
4	Zona_5	2	2	2	2
5	Zona_5	3	2	3	2
5	Zona_4	3	2	3	2
6	Zona_4	3	3	3	3
6	Zona_5	3	3	3	3
7	Zona_4	4	3	4	3
7	Zona_5	4	3	4	3
8	Zona_4	4	4	4	4
8	Zona_5	4	4	4	4
9	Zona_5	5	4	5	4
9	Zona_4	5	4	5	4
10	Zona_4	5	5	5	5
10	Zona_5	5	5	5	5
11	Zona_5	6	5	6	5
11	Zona_4	6	5	6	5
12	Zona_4	6	6	6	6
12	Zona_5	6	6	6	6
13	Zona_5	7	6	7	6
13	Zona_4	7	6	7	6
14	Zona_4	7	7	7	7
14	Zona_5	7	7	7	7

Table 10 : *Result scenario 5 multiple users (1).*

In the last situation on this scenario not everyone gets out of both areas. In this case nine persons get into the areas and just seven get out.

As expected (table 11) on the end the result, it shows nine entries for both areas and just seven exits with the same value for distinct ins and outs, once these were different persons.

Results on HDFS (table 11):

flowNumber	areaName	ins	outs	distinctIns	distinctOuts
1	Zona_4	1	0	1	0
1	Zona_5	1	0	1	0
2	Zona_4	1	1	1	1
2	Zona_5	1	1	1	1
3	Zona_4	2	1	2	1
3	Zona_5	2	1	2	1
4	Zona_4	2	2	2	2
4	Zona_5	2	2	2	2
5	Zona_4	3	2	3	2
5	Zona_5	3	2	3	2
6	Zona_4	3	3	3	3
6	Zona_5	3	3	3	3
7	Zona_5	4	3	4	3
7	Zona_4	4	3	4	3
8	Zona_5	5	3	5	3
8	Zona_4	5	3	5	3
9	Zona_4	5	4	5	4
9	Zona_5	5	4	5	4
10	Zona_5	6	4	6	4
10	Zona_4	6	4	6	4
11	Zona_4	6	5	6	5
11	Zona_5	6	5	6	5
12	Zona_5	7	5	7	5
12	Zona_4	7	5	7	5
13	Zona_5	7	6	7	6
13	Zona_4	7	6	7	6
14	Zona_4	8	6	8	6
14	Zona_5	8	6	8	6
15	Zona_5	8	7	8	7
15	Zona_4	8	7	8	7
16	Zona_5	9	7	9	7
16	Zona_4	9	7	9	7

Table 11 : *Result scenario 5 multiple users (2).*

4.3.6 Testing File Types

This test was done aiming to understand how Flink deals with the compressing or uncompressing files. In this particular case the test was performed over gzip and txt files. Considering the different types and with the same parallelism (24) the idea was to change the number of files to see how Flink reacts in terms of throughput (events processed per second).

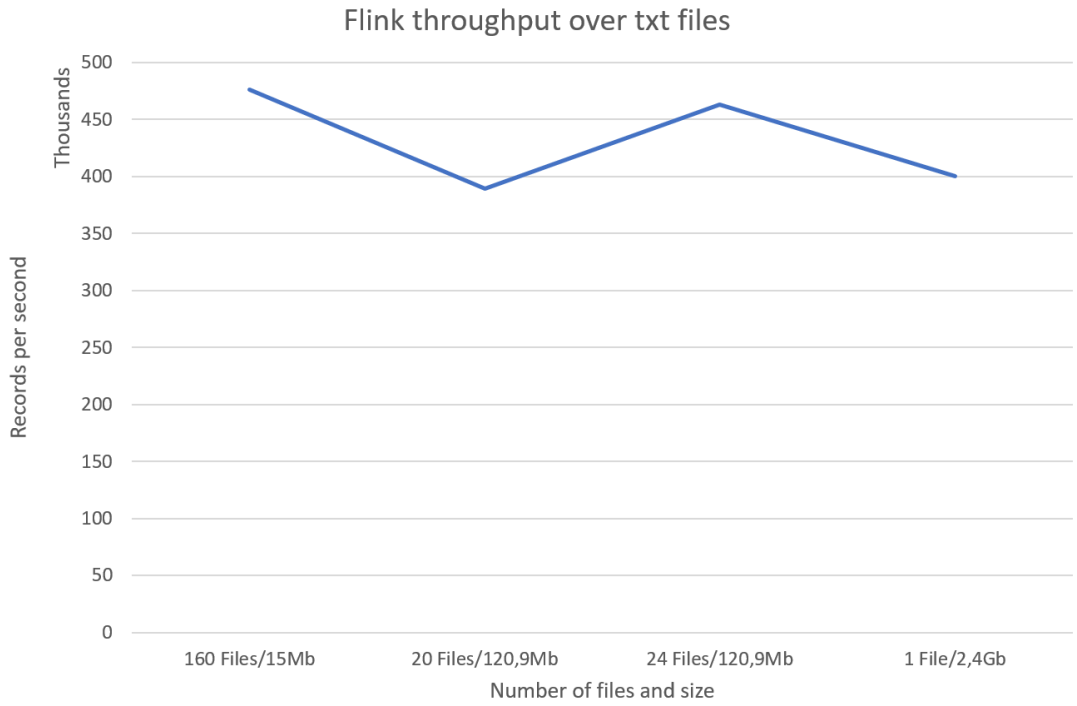

Chart 1: *Throughput over text files*.

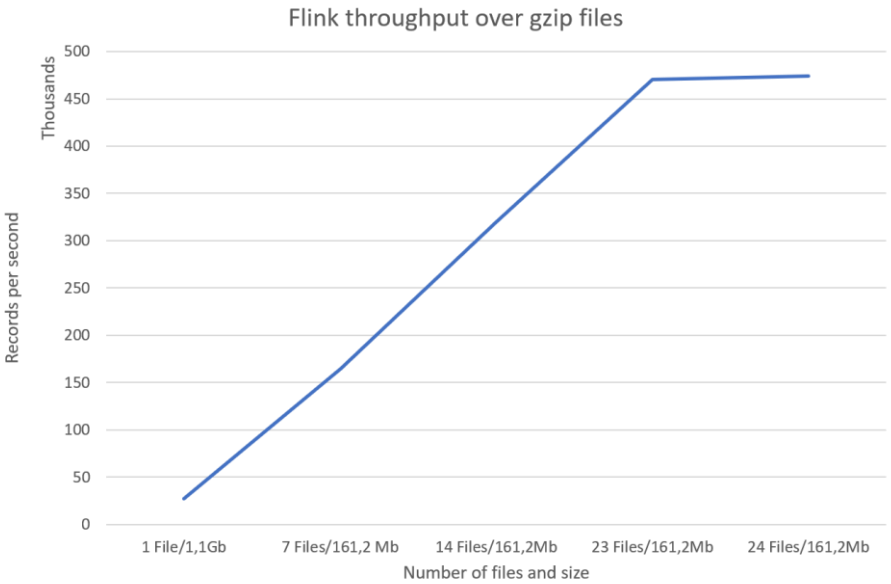

Chart 2: *Throughput over gzip files.*

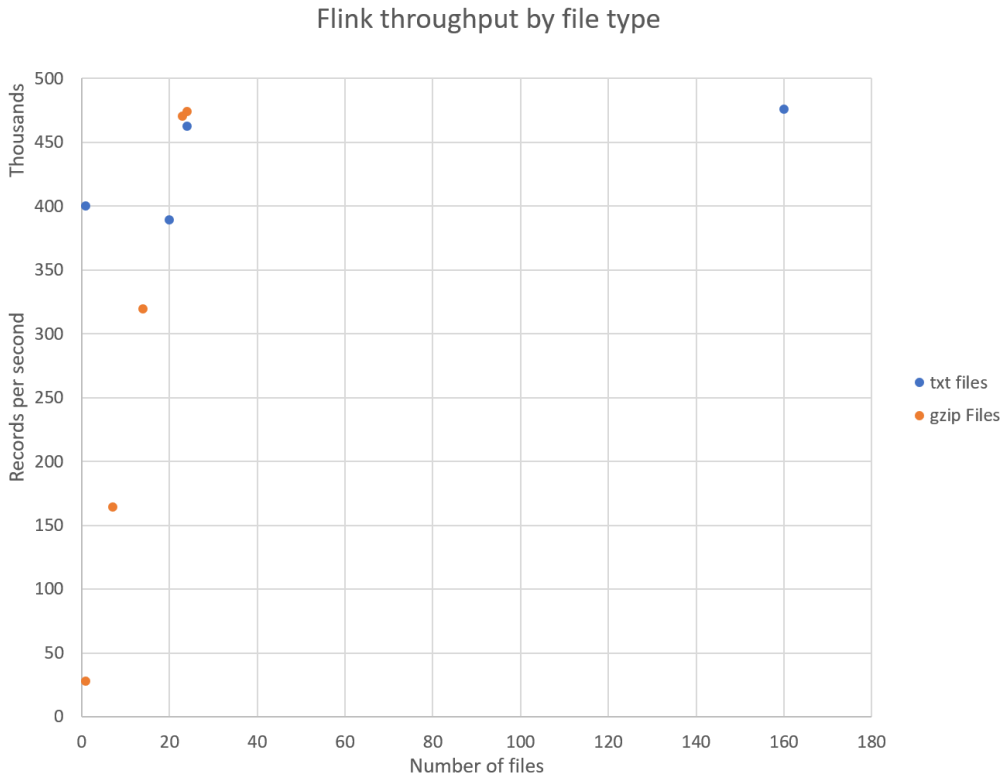

Chart 3: *Overall scenarios.*

Using parallelism 24 for both file types the result was the expected, since Flink *readFile* function only parallelized the reading on a single file if this was uncompressed. As showed on txt chart, the results were basically the same independent of the number of files (400000-450000 events/second). This happened because Flink parallelized the reading for one or various files, so the result is almost the same.

When dealing with compressed files, the test showed the expected. A low throughput (27000 events/second) occurs when there is just one file, because there is only one parallel instance reading the file. When there is an increase in the number of files to match the number of parallel instances, the result is basically the same to the one achieved in text files. This is because Flink is able to parallelize the reading of multiple compressed files.

Since, this test was more about how Flink reacts, the decision was to use the batch solution. The reason being, batch jobs are finite jobs so it is possible to know with more accuracy when a job starts and when it ends.

The tests are executed over a cluster managed by YARN. The cluster has three nodes with 25 Gb of memory and 16 virtual cores (Vcores) each node.

4.3.7 Testing Flink Configuration

To understand what type of Flink configuration is more pleasant on YARN cluster, the decision was to change multiple parameters, such ad number of TaskManagers (TM), TaskSlots (TS) and the number of Vcores for each YARN container. With the same file type (text in these cases) for each configuration the throughput was measured.

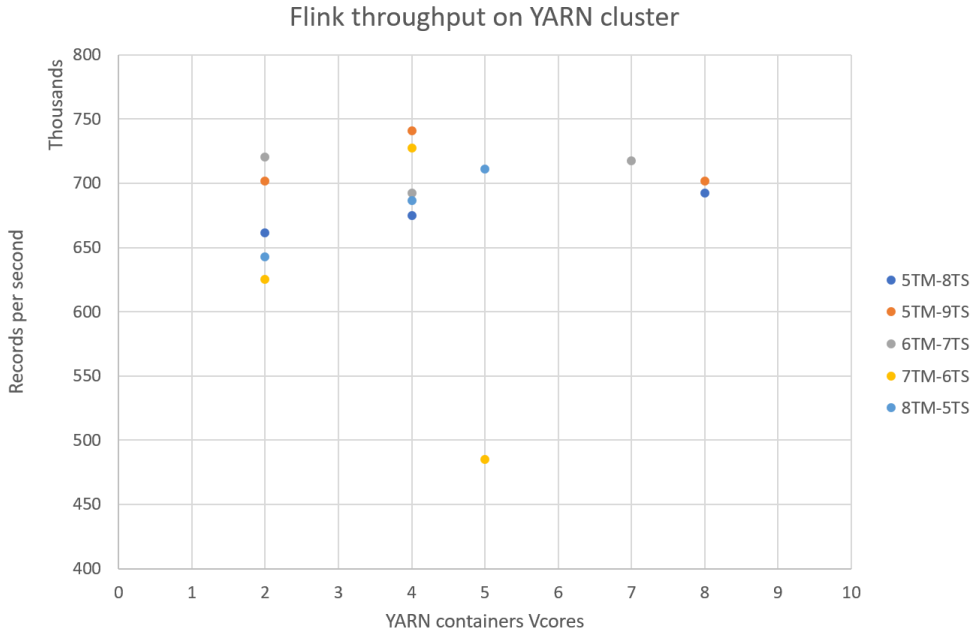

Chart 4: *Overall throughput on YARN cluster.*

On a YARN cluster each TaskManager (TM) is deployed on a container. In this scenario, the idea is to understand how Flink reacts with different Vcores in each container. The YARN cluster has 16 Vcores available for each node and 25 Gb of memory. Due to this the number of TaskManager that is possible to rise up is limited to eight due the memory available in each node. Through the results it is possible to see that the higher throughputs (0,7-0,75 million events per second) are related to the higher parallelism (TM*TS). And the desirable number of Vcores is the number of TaskSlots, in this way each TaskSlot is mapped with one Vcore.

4.3.8 Fault Tolerance and Scalability Tests

This test aims to prove that the solution is able to restore from a failure without impact in metrics calculation. With the same data stream the goal is to confirm that the

final result is the same with or without failure. First the execution is made normally with the data stream. The data stream is composed by two million (in two files) of records and ten defined areas. The result is shown below.

Result, 2 files, 2000000 Records processed, no fail (table 12):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 12: *Result fault tolerance test, no fail.*

The next test is done using the same data source. But now the execution is interrupted between the two files. And resumed from the earlier checkpoint.

Results, 2 files 2000000 records processed, fail between files (table 13):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 13*: Result fault tolerance test, fail between files.*

Now the interruption is performed during the processing of first file. The resume like the above scenario is done using the earlier checkpoint.

Results, 2 files 2000000 records processed, fail during the first file (table 14):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 14: *Result fault tolerance test, fail during first file.*

The results from all scenarios show that the solution is able to resume from failures with no impact in the outcomes. This confirms that the solution is fault tolerant. In order to test the solution scalability, similar tests are performed. However, in these tests (during the failure) the solution parallelism is increased and decreased. Hence, the resume is done from the earlier checkpoint and from the save point. The goal for these tests is to simulate overhead situations that lead to failure. It is necessary to increase the resources (parallelism) to deal with more data. Moreover, in a real world, it is necessary to upgrade the system. Due to this, the application must handle interruption and resumes with different configurations.

In the next test, the job is canceled and a save point is triggered between two files. Then the resume is done from the saved point with increased parallelism.

Results, 2 files 2000000 records processed, cancel job between files with save point and parallelism increased 10 → 20 (table 15):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 15: *Scalability test, increasing parallelism 10→20 from savepoint.*

The previous test shows that it is possible to increase the parallelism when resuming from a saved point. The next step is to test if it is possible to resume from a checkpoint with parallelism increase.

Results, 2 files 2000000 records processed, cancel job between files and resume from checkpoint with parallelism increased 10 → 20 (table 16):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 16: *Scalability test, increasing parallelism 10→20 from checkpoint.*

The above tests prove that it is possible to scale up the solution with no impact in results. Finally, the last test aims to prove that the solution is also able to scale down. For that, when the interruption is done, the parallelism is decreased and then the job is resumed.

Results, 2 files 2000000 records processed, cancel job between files and then resume from checkpoint with parallelism decreased 30 → 18 (table 17):

areaName	ins	outs	distinctIns	distinctOuts
Zona_1	77	2	77	2
Zona_2	67	1	67	1
Zona_3	63	0	63	0
Zona_4	39	0	39	0
Zona_5	33	0	33	0
Zona_6	42	0	42	0
Zona_7	49	0	49	0
Zona_8	108	1	108	1
Zona_9	58	1	58	1
Zona_10	174	0	174	0

Table 17: *Scalability test, decreasing parallelism 30→18 from checkpoint.*

As shown in the above tables of results, the framework is capable to scale up or down without impacts on results as intended.

5. Conclusion

This chapter present an overview of the work developed on this dissertation, ending with some considerations to a future work improvement.

5.1 Work Overview

The aim of this dissertation was to build a distributed framework that was scalable, reliable and fault tolerant. Also, that could perform metrics calculations about generic mobile users geographical information retrieved by the mobile network on a streaming environment. It was extremely important to maintain the privacy as well the anonymity of the mobile users. In this regard, the information about mobile users was previously submitted to a *one-way-script-obscure-function.* This way the work was done over anonymous data as intended. The implementation was achieved using Flink and as a proof of concept it was integrated into a location based system that determined persons presences in user defined areas.

The calculation of metrics about people geographical information in a streaming environment leads to multiple concerns. The main important concern is to treat each record in the same order that was produced in the Base Transceiver Station (BTS). Since the files came from multiples BTS, it was not possible to guarantee the order of the arriving files. Due to this the stream processor must be able to handle this concern.

The other important concern was to ensure that the stream processor worked in a fault tolerant way. In case of failure, the framework must be able to recover from an earlier safe and coherent state in a way that the coherency of results continues with the guarantee that no event is reprocessed or left behind (exactly once semantic). Also, the solution must ensure scalability. In other words, when the execution is interrupted (intentional or not) it should be possible to scale up or down the solution parallelism.

All the calculated metrics about persons' geographical positions were determined over the areas that are inserted into the system by users. So, the system was able to receive these areas and remove them. Once the areas are removed the system must stop the metrics calculation.

During the implementation, it was possible to subdivide the solution requirements into smaller tasks. Starting with the guarantee of the reception of late events and consequent ordering. This requirement was achieved using a time buffer. This buffer was implemented with the Flink *window* function that considered a late time for the late arrives. This means that the *window* is only closed after the time defined plus the late time, and the sort is done. At this point, it is known that the *window* considers the late arrival and future operations will occur over an order stream. Hence, it is

guaranteed that the early events are treated first. This guarantee is crucial to ensure a reliable metric calculation over geographical information.

Regarding to the fault tolerance requirement, having a stream processor means that the processing is done 24 hours/7 days. Obviously, fails will occur and the system needs to be capable to handle those fails and recover from them. Also, the recover must happen in a coherent and consistent way. Implementation was done using stateful Flink operators with externalized checkpoints. This way, it guarantees that all required Flink operators have a well done defined state that is periodically stored into a persistent storage, in this case HDFS. If a failure occurs, it is possible to restore the processing from the earlier stored checkpoint with the insurance of the exactly once semantic.

The functionality tests performed to the framework confirmed that the metric calculation was done in a consistent way, even on tricky scenarios as mentioned in section 4.3. Having the out order and late arrival problems solved.

Concerning the fault tolerance and scalability, to ensure and confirm the consistency, the streaming job was cancelled multiple times and restored from the previous checkpoint. In some of these cases, it was performed an increase and decrease of parallelism (scale up and down). In order to verify the exactly once semantic and consistent metrics count.

As another proof of concept, a web viewer was implemented to display persons presences over time. The implementation was done using HTML5 and gmaps with heatmaps. The representation is done over a time line and it is possible to see the movement, due to the different gradients of the heatmap.

5.2 Future Work

The aim of this dissertation was to make a reliable stream processor for geographical metrics calculation over user defined areas and the goal was achieved. At this point there is a Flink application that is able to perform the job in a consistent and fault tolerant way. The following suggestions aim to improve the solution:

- The web visualizer at this point only shows information from one day due the absence of real data. Desirable it should be linked to the results from the Flink stream processor.
- The areas are inserted and removed via files into HDFS. In future it would be more pleasant if this functionality is implemented on the web visualizer and via Kafka arrives to Flink.

- Once the streaming processor receives real data since the enrichment and metrics about areas is already done, it would be interesting to apply machine learning (ML) algorithms to infer persons behaves and patterns.

References

[1] "Twits." [Online]. Available: www.internetlivestats.com/twitter-statistics/.

[2] Z. Cheng, J. Caverlee, and K. Lee, "You Are Where You Tweet : A Content-Based Approach to Geo-locating Twitter Users," *Proc. 19th ACM Int. Conf. Inf. Knowl. Manag.*, pp. 759–768, 2010.

[3] J. Bao, Y. Zheng, and M. F. Mokbel, "Location-based and preference-aware recommendation using sparse geo-social networking data," *Proc. 20th Int. Conf. Adv. Geogr. Inf. Syst. - SIGSPATIAL '12*, no. c, p. 199, 2012.

[4] G. Lansley and P. A. Longley, "The geography of Twitter topics in London," *Comput. Environ. Urban Syst.*, vol. 58, pp. 85–96, 2016.

[5] M. Li, G. Sagl, L. Mburu, and H. Fan, "A contextualized and personalized model to predict user interest using location-based social networks," *Comput. Environ. Urban Syst.*, vol. 58, pp. 97–106, 2016.

[6] "OECD." [Online]. Available: http://www.oecd.org/sti/broadband/oecdbroadbandportal.htm.

[7] S. Aheleroff, "Applying Call and Event Detail Records to Customer Segmentation and CLV," vol. 3, no. 8, 2013.

[8] R. Becker *et al.*, "Route classification using cellular handoff patterns," *Proc. 13th Int. Conf. Ubiquitous Comput. - UbiComp '11*, pp. 123–132, 2011.

[9] S. A. Ríos and R. Muñoz, "Land Use detection with cell phone data using topic models: Case Santiago, Chile," *Comput. Environ. Urban Syst.*, vol. 61, pp. 39–48, 2017.

[10] "The Internet of Things: Seizing the Benefits and Addressing the Challenges," 2016. [Online]. Available: http://www.oecd.org/officialdocuments/publicdisplaydocumentpdf/?cote=DSTI/ICCP/CISP(2015)3/FINAL&docLanguage=En.

[11] "DataArtisans-StreamProcessing." [Online]. Available: https://data-artisans.com/what-is-stream-processing.

[12] M. Verrilli, "From Lambda to Kappa: A Guide on Real-time Big Data Architectures." [Online]. Available: https://www.talend.com/blog/2017/08/28/lambda-kappa-real-time-big-data-architectures/.

[13] "Flink." [Online]. Available: https://flink.apache.org.

[14] J. Ellingwood, "Hadoop, Storm, Samza, Spark, and Flink: Big Data Frameworks Compared." [Online]. Available: https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared.

[15] "Flink Time." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/event_time.html.

[16] "Batch is a special case of streaming." [Online]. Available: https://data-artisans.com/blog/batch-is-a-special-case-of-streaming.

[17] "Stream Processing Myths Debunked." [Online]. Available: https://data-artisans.com/blog/stream-processing-myths-debunked.

[18] "Flink programming model." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/concepts/programming-model.html.

[19] "Flink DataStream API." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/datastream_api.html.

[20] "Flink DataSet API." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/batch/index.html.

[21] "Flink Table API." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/table/index.html.

[22] "Flink Runtime." [Online]. Available: https://ci.apache.org/projects/flink/flink-

docs-release-1.3/concepts/runtime.html.

[23] "Flink Working with state." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/state.html.

[24] "Flink Checkpointing." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/checkpointing.html.

[25] "Flink Externalized Checkpoints." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/checkpoints.html#externalized-checkpoints.

[26] "Flink Savepoints." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/savepoints.html.

[27] K. Goodhope, J. Koshy, and J. Kreps, "Building LinkedIn's Real-time Activity Data Pipeline.," *IEEE Data Eng*, pp. 1–13, 2012.

[28] "Linkedin." [Online]. Available: https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin.

[29] "Kafka." [Online]. Available: https://kafka.apache.org/intro.html.

[30] "RFC Radius." [Online]. Available: http://www.rfc-base.org/rfc-2865.html.

[31] "PostgreSQL." [Online]. Available: https://www.postgresql.org/about/history/.

[32] "Opengeospatial." [Online]. Available: http://www.opengeospatial.org/standards.

[33] "NFLabs." [Online]. Available: www.zepl.com.

[34] "Apache." [Online]. Available: https://www.apache.org/.

[35] "Spark." [Online]. Available: http://www.spark.tc/moon-soo-lee-interview.

[36] "Zeppelin." [Online]. Available: https://zeppelin.apache.org/.

[37] "Apache Spark." [Online]. Available: https://spark.apache.org/.

[38] "Python." [Online]. Available: https://www.python.org/.

[39] "Java Database Connectivity." [Online]. Available: http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

[40] "Jupyter about." [Online]. Available: http://jupyter.org/about.html.

[41] "Folium." [Online]. Available: https://github.com/python-visualization/folium.

[42] "Ipyleaflet." [Online]. Available: https://github.com/ellisonbg/ipyleaflet.

[43] "GoogleMaps." [Online]. Available: https://github.com/googlemaps/google-maps-services-python.

[44] C. Lin and M. Hung, "A Location-based Personal Task Reminder for Mobile Users," *Pers. Ubiquitous Comput.*, vol. 18, no. 2, pp. 303–314, 2014.

[45] H. Assem, T. Buda, and D. O´sullivan, "RCMC : Recognizing Crowd Mobility Patterns in Cities based on Location Based Social Networks Data," vol. 8, no. 70, 2011.

[46] M. M. Al-Rajab, S. A. Alkheder, and S. A. Hoshang, "An Intelligent Location-Based Service System (ILBSS) using mobile and spatial technology: A proposal for Abu Dhabi petrol stations," *Case Stud. Transp. Policy*, vol. 5, no. 2, pp. 245–253, 2017.