Scalable integration and pre-processing of sensor data streams

M.Sc. in Computing Science

March 2014

Student: Marco Andres Tello Guerrero

RuG First Supervisor: dr. Alexander Lazovik

RuG Second Supervisor: Prof. dr. ir. Paris Avgeriou

TNO Supervisor: ir. Bram van der Waaij
To Efrén and Carmen, my loving parents. Their encouragement and unconditional support made this work possible.

To Rafael, my brother, who was always there motivating me during all this process.

A Efrén y Carmen, mis queridos padres. Su aliento y ayuda incondicional hicieron este trabajo posible.

A Rafael, mi hermano, que siempre estuvo allí motivándome durante todo este proceso.
Acknowledgments

I would like to thank Bram van der Waaij, my supervisor at TNO, for his guidance and daily supervision. His support contributed to carry out and conclude this research.

I would also like to thank Alexander Lazovik, my first supervisor at RuG, for his intellectual support. His advice helped to improve the content of this thesis from the academic point of view.

I would also like to thank Elena Lazovik and Edwin Harmsma, my colleagues at TNO, for their ideas to improve this work, and for their humor, making this work an enjoyable journey.

I would also like to thank the SENESCYT, the Ecuadorian governmental organization that financially supported my master education.

I also want to thank Angelica and Michael, their encouragement and inspiration made this work an enjoyable experience. In addition, I want to thank my friends from Ecuador who were supporting me during my two years abroad.

Last but not least, I want to thank Theo and Daphne for their motivation and support during these last two years.
Abstract

Data from sensors is highly heterogeneous and coping with such heterogeneity is challenging and time consuming [3]. Each sensor network deployment has its own data formats. In addition, each deployment has its own system for gathering, processing, and publishing the data coming from the sensors, especially if they are managed by different authorities. Pre-processing of sensor streams is mandatory in order to translate the input data to a common representation within the entire processing pipeline [19]. Hence, creating a platform that integrates the inputs of several sensor networks is not a trivial task; creating a middleware that allows processing at scale in a partitioned, distributed setting is even more challenging.

In this thesis we proposed a sensor data integration solution that relies on Enterprise Integration Patterns (EIPs), a Distributed Message-Oriented Middleware (MOM), and a state of the art Stream Processing Framework (SPF). First we present the EIPs suitable for sensor data integration and the capabilities and limitations of current SPFs. Then we formalize the challenges of sensor data integration and pre-processing and present the definitions of the problems we need to solve and the algorithms for the proposed solution. Next, we present a prototype of the proposed solution implemented on top of Apache Kafka and Storm and we provide a rationale for choosing these specific platforms. Finally we evaluate our solution in terms of scalability, performance, and reliability. We present different test scenarios that include three datasets comprised of 5, 100, and 500 thousand sensor readings. The tests proved that it is easy to scale without losing accuracy and shows that about 75% in performance is gained when running in parallel.
## Contents

1 Introduction ................................................................. 1  
  1.1 Context .................................................................. 2  
  1.2 Research Questions .................................................... 3  
  1.3 Contribution .............................................................. 5  

2 Related Work ................................................................. 6  
  2.1 Data Integration .......................................................... 6  
    2.1.1 Overview of Enterprise Application Integration .............. 7  
    2.1.2 Integration Patterns ............................................... 7  
    2.1.3 Enterprise Service Bus ........................................... 12  
    2.1.4 Sensor Data Integration ........................................... 15  
  2.2 Stream Processing ....................................................... 18  
    2.2.1 Characteristics of Stream Processing Systems ............... 18  
    2.2.2 Evolution of Stream Processing Systems ...................... 21  
    2.2.3 Data Stream Processing vs. Complex Event Processing .... 23  

3 Problem Definition .......................................................... 25  
  3.1 Basic definitions ......................................................... 25  
  3.2 Problems definitions and proposed solutions ...................... 27  
    3.2.1 Data transformations .............................................. 27  
    3.2.2 Multiple Stream Aggregates ..................................... 28  
    3.2.3 Windowing Mechanisms .......................................... 30  
    3.2.4 Time Alignment ................................................... 34  

4 System Design ............................................................... 40  
  4.1 High level goals ........................................................ 40  
  4.2 Key Drivers ............................................................. 40  
  4.3 Requirements .......................................................... 41
4.3.1 Functional Requirements ........................................ 42
4.3.2 Non-Functional Requirements ................................. 43
4.4 System Architecture ................................................. 43
  4.4.1 Rationale ...................................................... 44
  4.4.2 Logical view .................................................. 44
     4.4.2.1 System Decomposition ................................. 45
  4.4.3 Process View ................................................ 47
     4.4.3.1 Data transformations ................................. 47
     4.4.3.2 Windowing Mechanisms ................................. 49

5 Implementation ....................................................... 51
  5.1 Software design decisions ...................................... 51
     5.1.1 Message-Oriented Middleware ............................ 51
     5.1.2 Stream Processing System ................................. 53
  5.2 Guaranteeing message processing .............................. 55
  5.3 Data transformations .......................................... 59
  5.4 Multiple Stream Aggregates ................................... 59
  5.5 Windowing Mechanisms ........................................ 61

6 Evaluation and Discussion .......................................... 63
  6.1 Evaluation Tests ............................................... 63
  6.2 Test 1. Single thread vs. Parallel topology ................. 65
  6.3 Test 2. Non partitioned vs. Partitioned topic ............... 66
  6.4 Test 3. Only Storm and different parallelism configurations ...... 67
  6.5 Test 4. Complete test using the last 15 minutes of data ....... 70
  6.6 Test 5. Reliability tests ...................................... 71

7 Conclusions .......................................................... 74

8 Future Work .......................................................... 76

A Input Datasets Examples ............................................ 78
  A.1 SAARecorder Version 4.86 ..................................... 78
  A.2 Cow Events File ............................................... 79
  A.3 Super sets ....................................................... 79

B Data Transformations Required at different levels .............. 81

Scalable integration and pre-processing of sensor data streams
## List of Figures

2.1 Basic Elements of an Integration Solution [23] .......................... 8
2.2 Channel Adapter ......................................................... 8
2.3 Message Translator ..................................................... 9
2.4 Message Translator use example ..................................... 10
2.5 Content Enricher ......................................................... 10
2.6 Content Filter .......................................................... 11
2.7 Normalizer .............................................................. 11
2.8 Splitter ................................................................. 11
2.9 Aggregator .............................................................. 12
2.10 Overview of an Enterprise Service Bus architecture ............... 13
2.11 Examples of windowing constructs .................................. 20

3.1 Logical view of the grouping aggregation ............................ 29
3.2 Physical view of the grouping aggregation ........................... 30
3.3 Example of the aggregator’s operation ............................... 31
3.4 Examples of count-based and time-based windows ................. 34
3.5 Time alignment of N *streams* ........................................ 36
3.6 Time alignment of N *streams* using windows of data .......... 37

4.1 High level architecture of the proposed integration framework . . 45
4.2 Internal view of the adapter component ............................. 46
4.3 Complete view of the system architecture .......................... 48
4.4 Activity diagram of the data transformation process ............... 49
4.5 Activity diagram of the window aggregation process .............. 50

5.1 Class diagram of the multiple aggregation using windows .......... 62

6.1 Different topology configurations for the evaluation tests ....... 65
6.2 Performance test with the 5k, and 100k datasets .................. 66
6.3 Performance test with the 500k dataset and different Kafka partitions 67
6.4 Performance test with different parallelism configurations with the 5k
dataset - only Storm ................................. 68
6.5 Performance test with the 100k dataset - only Storm ............... 69
6.6 Performance test with the 500k dataset - only Storm .............. 70
6.7 Performance test using 15 minutes of data ......................... 71
6.8 Average of a 1 minute window calculated every 30 seconds ....... 72
6.9 Average of a 1 minute window calculated every 30 seconds running in
parallel without preserving order .............................. 73
6.10 Average of a 1 minute window calculated every 30 seconds running in
parallel and order preserved ............................... 73
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Functional requirements</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Non-functional requirements</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>MOM Design Decisions</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>SPF Design Decisions</td>
<td>54</td>
</tr>
<tr>
<td>B.1</td>
<td>Transformation levels</td>
<td>82</td>
</tr>
</tbody>
</table>
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>canonical in-stream data integration application</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>Multiple aggregates</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Windowing</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Time Alignment</td>
<td>39</td>
</tr>
</tbody>
</table>
Listings

A.1 SAARecorder Sample File ........................................ 78
A.2 Cow Events Sample File ........................................ 79
A.3 Superset Sample File ........................................ 79
Chapter 1

Introduction

Academy and industry are working together to find solutions for problems that are affecting the society. For instance, improving building energy efficiency, creating awareness about energy consumption, monitoring the health of infrastructures, controlling the impact of natural disasters, and reducing traffic congestion, are some of the main concerns of governments, scientists and society today. Thus, real-time information gathered from the physical world is essential for a thoughtful decision making and effective response in case of undesirable events. On the other hand, technological developments have brought a diverse variety of sensor devices which nowadays are relatively cheap and can be placed almost anywhere. As such, ubiquitous sensor networks have become the straightforward approach for monitoring and understanding several physical phenomena and thereby achieve the aforementioned goals.

Data from sensors is highly heterogeneous and coping with such heterogeneity is challenging and time consuming [3]. Each sensor network deployment has its own data formats. In addition, each deployment has its own system for gathering, processing, and publishing the data coming from the sensors, especially if they are managed by different authorities. Pre-processing of sensor streams is mandatory in order to translate the input data to a common representation within the entire processing pipeline [19]. Hence, creating a platform that integrates the inputs of several sensor networks is not a trivial task; creating a middleware that allows processing at scale in a partitioned, distributed setting is even more challenging.

This research is part of the AnySense Project at the Dutch company TNO. It is aimed to contribute with a middleware for integrating and pre-processing data produced by heterogeneous, autonomously deployed sensor networks in near real-time. We propose an approach that combines the theory of data integration in order to cope with the
challenges imposed by the heterogeneity of the data sources, while exploiting the capabilities of the Stream Processing Systems (SPS) to achieve high throughput and low latency.

1.1 Context

When people working on sensor data analysis are also the owners of the sensor networks, integration is not a concern. They know in which format the data is being produced and they easily adapt the raw data to the format required by the analysis platform. However, when the sensor networks are managed by external parties, an agreement on data formats is more difficult to achieve. In the latter case, the parties interested in receiving the data have to handle with the heterogeneity of the data sources. In addition, they have to cope with the case that parties producing the input data suddenly change the format of the data that is being processed by the analysis platform.

Researchers are working on different problems. For instance, in the WATTALYST project, which is a consortium comprised by seven partners, they are working on modeling and analyzing demand and response systems [19]. The Department of Energy and Climate Change of UK is working on Energy Efficiency [13]. The University of Utrecht together with the Dutch government is working on monitoring farm animals to avoid infectious diseases in humans [55]. TNO is working on dike monitoring, health of buildings and underground infrastructures, farm animals’ conditions, and canal cruise fleet activity. The challenge in these examples is that the sensor networks deployed to sense the physical environment, are not managed by the project team. They receive the data from external partners. Each of those partners have their own systems for gathering and publishing the sensor data. Hence, they have to deal with such heterogeneity and pre-process the data to transform it into a predefined schema that can be understood by the analysis platform. In addition, sometimes they have to face with the fact that external parties producing the data suddenly change the agreed data formats. Hence, the analysis platform has to adapt to those changes and acts accordingly to such unexpected behavior.

In addition, one of the requirements in this kind of projects is the need for processing the sensor data on the fly. They require to produce results in near-real time, and therefore high-throughput and low-latency are desired features. Sensor networks usually comprise a few hundred or even thousands of sensors producing data at short
intervals. Without efficient programming models, the pre-processing stage might become the bottleneck of the whole analysis platform. This become even more dramatic when the number of data sources increases over the time.

At this moment, the common approach is to use a data integration framework, e.g., Apache Camel, OpenAdaptor, etc. For instance, researchers at TNO use OpenAdaptor for ingesting the data into the analysis platform. Even though the current implementation is working for the intended purpose, it is not giving the expected response. Researchers want to have a flexible mechanism for integrating new data sources into the analysis platform. Such mechanism should help them to reduce the deployment time when a new data source joins the system. In the current approach a processing pipeline has to be expressed in xml format and the schema of such xml file is quite complex. Hence, it requires a considerable effort to create, or maintain a pre-processing pipeline. In addition, OpenAdaptor by design does not support distributed parallel processing, which is a desirable property in order to process large volumes of data and produce faster responses. When performing sequential processing is more difficult to achieve the desired near-real time response.

### 1.2 Research Questions

From the context described in 1.1 we realize that our work falls within two different areas. The heterogeneity of the data sources providing sensor data imposes challenges related with data integration. On the other hand, the need for high-throughput and low-latency when processing large volumes of sensor data requires to explore new paradigms and programming models such as stream processing.

Concerning data integration, Apache Camel [49], MuleSoft [53], and other integration frameworks, are built on a rich theoretical foundation in terms of integration patterns. They provide out of the box components which implement most of the patterns proposed by Hohpe and Woolf in [23]. In addition, they provide stream processing capabilities by chaining such components to create a processing pipeline. Although these tools allow data integration, they do not provide built-in capabilities in terms of scalability, and parallel processing.

On the other hand, we have new frameworks and programming paradigms which allow processing large volumes of data at very high speed by providing horizontal scalability and distributed parallel processing. However, such frameworks were not designed with
support for integration in mind [46]. Hadoop [50], an open source implementation of the MapReduce paradigm, overcome these limitations, but it was intended for batch processing. There are different Hadoop’s extensions aimed to achieve low-latency responses, but those implementations result in compromised flexibility [4] and limited stream processing capabilities [35]. Today, there are new alternatives aimed to supplement Hadoop-based systems. S4 [25], Storm [41], and Samza [26] are the state-of-the-art stream processing systems. They are aimed for continuous processing of unbounded data streams, allowing what hadoop does in batch, but in near-real time.

From the aforementioned image we see that there is a plethora of systems aimed, from one side, to face the challenges imposed by data integration, and from a different perspective, the challenges imposed by the need for processing large volumes of data, as soon as data is being produced, with low-latency responses. Hence, the main research question is:

*Can stream processing techniques be applied for integrating and pre-processing data from heterogeneous sensor networks?*

In order to answer this question, the following subquestions are stated:

1. *How Stream Processing Systems can provide data integration capabilities?*. Integration Frameworks provide build-in components that facilitate integration. We need to identify which of those components are needed to ease sensor data integration, and how such functionality could be included in stream processing systems.

2. *How to adjust Stream Processing Systems in order to they cope with sensor data specifics?*. We need to identify which are the sensor data specifics and whether the current stream processing systems provide functionality to cope with the specifics of the sensor data domain.

3. *How could Stream Processing Systems reduce the deployment time of a new integration project?* Current Integration Frameworks use xml configuration files in order to define the stream processing pipeline. Such files sometimes result in complex structures tedious to define and modify. We need to find whether stream processing systems opt for a better approach for defining processing pipelines.
4. How to deal with the challenges imposed by scalability and parallel processing capabilities offered by the Stream Processing Systems? With the state of the art stream processing system we gain high throughput and low latency by means of horizontal scalability and parallel processing provided by them. However, these new features may result in associated challenges that developers need to cope with at implementation time.

1.3 Contribution

The remainder of this document is as follows. In chapter 2 we describe the concepts in the field of Enterprise Application Integration and Stream Processing which are the theoretical foundation on which we base our approach. In addition, we present the work made by other researchers in both fields. In chapter 3 we present the formal definitions of the problem that we want to solve and the proposed solutions. In chapter 4 we depict the design of a prototype that implement the proposed solutions described in chapter 3. In chapter 5 we explain the implementation phase of our prototype, the software used in our approach and a rationale about why we choose those technologies. In chapter 6 we evaluate our solution in terms of scalability, performance and reliability and we discuss our results. In chapter 7 we present the conclusions, and in chapter 8 we present the future work that could be done from our research.
Chapter 2

Related Work

In this chapter we present the theoretical background that is related with our work. First we present the theory in the field of Enterprise Applications Integration, which includes what other researchers have been doing in sensor data integration. Then we present the literature in the field of Stream Processing, and the evolution of the Stream Processing Systems (SPF). This conceptual foundation will contribute to answer the questions presented in chapter 1.

2.1 Data Integration

As Hohpe and Woolf state, data integration is a complex activity which impose different challenges. “Anyone who claims that integration is easy must be either incredibly smart (or at least a good bit smarter than the rest of us), incredibly ignorant (OK, let’s say optimistic), or they have a financial interest in making you believe that integration is easy” [23].

One of the drivers of our work was the need for integrating data produced by distributed, heterogeneous, and autonomously deployed sensor networks, and serve such data to other applications (e.g., analysis platform, persistent storage, etc.). Therefore, we need a theoretical background in order to understand which are the challenges of integration, and how to face them.
2.1.1 Overview of Enterprise Application Integration

Enterprise Application Integration (EAI) is a term used to describe different approaches to provide interoperability between disparate systems that comprise a typical enterprise infrastructure [54].

The IT infrastructure of a company comprises hundreds or even thousands of applications. Some of them are developed inside the company, others are acquired from a third party enterprise, others are part of a legacy system, or a combination of them. In order to support the day to day business processes, all these applications have to communicate and share data. This scenario becomes even more complex when applications from external partners have to be integrated. This type of scenario is called business-to-business integration [23].

The first attempt for EAI was the Point-to-Point integration. This approach proposed the implementation of a connector component between each pair of systems that must communicate. Such component has to handle with the communication, data transformations, and other messages related to the business operations. This approach was viable for small companies were the number of applications to connect was not big. For instance, a point-to-point integration for three systems needs only three connectors to provide interoperability among all the systems. However, if two more applications are added in the future, the number of connectors would increase to ten. Hence, the complexity of this approach would become inevitable unmanageable. This approach leads to what is known as spaghetti architecture.

Later on, Hohpe and Woolf presented different Enterprise Integration Patterns (EIP) in order to cope with the challenges of data integration. A pattern is an advice that describe general solutions to frequently recurring problems [23]. Hence, they propose that a set of patterns applied using asynchronous messaging systems may alleviate the associated complexity of integration.

2.1.2 Integration Patterns

Integration with Messaging and Patterns. Integration problems may be tackled using a message-oriented middleware (MOM) and several patterns that can be implemented on top of such messaging system [23].
Figure 2.1 shows the basic elements of an integration solution proposed in [23]. To integrate applications, data (Messages) has to be transferred from one application to another. We need an Endpoint, in order to connect an application to the integration system. Once the application is connected to the integration solution, we need a communication Channel in order to move data between applications. Additionally, we need Routing capabilities in order to send the messages to the correct applications that will process those messages. Such routing capabilities can be provided by a message-oriented middleware. One of the main issues of an integration project is the difficulty of agreeing in a common data format. Hence, we need a Translator to convert one application's data format into the other's. Now that all the pieces are in place, we need a Monitoring system to monitor the flow of data, and verify that all components are available.

The description of the basic elements of an integration solution gives an overview of the whole picture. In addition, we need to identify patterns which can be translated into components that facilitate handling more detailed integration problems. We studied the integration patterns proposed in [23]; and in the following paragraphs, we describe the patterns which facilitate sensor data integration.

Channel Adapter. In order to integrate different applications using a MOM, we need that the applications can send and receive messages. “A channel adapter can access the applications’ APIs or data and publish messages based on this data” [23]. Most applications were not designed to share data using a messaging infrastructure. Nonetheless, some of those applications expose an API in order to other applications can
use some of their functionality and access the data. There are applications that exchange data using files or database tables. Other applications use simple protocols for communication such as HTTP or TCP/IP. The examples cited before are just some cases where a channel adapter may be used to connect those applications with a messaging system.

**Message Translator.** A message translator is a generic abstraction used to depict a component that translates one application’s data format into another format. Applications produce data in a specific data format, and that data fits into a predefined schema for that particular application. Hence, if two or more systems want to share data, it has to be in a common format that all applications can understand. One solution could be to change each application to use a common data format. However, this approach sometimes is unfeasible because it might incur in several changes to inherent business functionality [23]. In addition, this approach would contrast with the main architectural principle in enterprise integration, loosely coupling. Even if all the applications produce data in the same format and with the same schema, the physical representation of the data might be still different. For instance, one application may use data using CSV files and other XML files.

There is a need for data transformation at different levels or layers of abstraction: Transport, Data Representation, Data Types and Application Layer [23]. Table B.1, in Appendix B, depicts those layers, and the transformation needs for sensor data streams. Usually such data translations need to be done at more than one level. For instance, in the sensor data analysis domain, sensors’ readings are represented as lines in a CSV file that have to be send over a message oriented middleware to the analysis platform, translated to XML or other format, and finally stored in the database. This scenario spans three levels. Transport changes from file to JMS, the data representation for changes from CSV to XML, and the data types level to comply with the constraints of the analysis platform. The benefits of the layered model is that we can treat the transformations differently at each layer. Creating one Message Translator for each layer allows us to reuse these components in other scenarios.

Figure 2.4 shows an example solution for the scenario described before. The dotted lines separate the transformations made at different layers. in this example, the File-
The message translator is a general solution to transform data from one format to another. However, it is common that integration solutions have to deal with more specific problems. The patterns described in the following paragraphs are variants of a message translator, and they are intended for more specific purposes.

**Content Enricher.** A content enricher is an operator that allows to augment information to the data being exchanged. In some cases, the data does not contain all the information required at processing time. Such information might be metadata or other information. In either case, the content enricher operator might be able to add the missing information to the data being transferred. The information to be added can be obtained by means of computation, from the environment, or from external systems.

- **Computation.** In some cases, the information to be added can be obtained from the information already present in the data, e.g., a calculated value. In this case, the content enricher provides the algorithm for such calculation.

- **Environment.** The content enricher must be able to retrieve information from the operating environment. For instance, if the data does not contain a timestamp, the content enricher might obtain the current time from the operating system and add it to the message.

- **External Systems.** This is the case when the information to be added has to be acquired from an external system, e.g., a database. The content enricher
operator has to retrieve that information from the external system and adds it to the message.

**Content Filter.** In contrast to the content enricher, a content filter operator allows removing unimportant elements from a message. There are different situations where this task is necessary. For example, consider that we want to preprocess sensor data about the canal cruise fleet activity in Amsterdam. However, we receive data with information from all the fleet activity in the Netherlands. In this case we may use a content filter operator to keep those records which contains only information about Amsterdam and filter out those unnecessary elements with data from other cities.

**Normalizer.** A normalizer is an operator that converts a message, or an element in a message, from one format to another. Depending on the needs of a particular scenario, a stream processing system might receive inputs from different data sources, and each of those sources may produce data in their own format. In our case, we are focused in the sensor data analysis domain, where the heterogeneity of the data sources is one of the challenges. A frequent use case for a normalizer is the need for transforming the timestamps into a common format, e.g., UTC.

**Splitter.** A splitter breaks a message into smaller elements each of which has to be processed in a different way. The splitter receives a message which is usually a list of elements each of which can be processed individually. For instance a list of sensor measurements where each element represents a different physical observed phenomena (e.g., temperature, pressure, humidity). The splitter produces a message for each single element or a subset of elements from the original message.
Figure 2.9: Aggregator. In contrast to the splitter, the aggregator allows to combine individual but related messages in order to process them as a whole. An aggregator “collects and stores individual messages until a complete set of related messages has been received.” Then, the Aggregator’s output will be a single message created from the individual messages [23]. The previous operators were stateless, they process messages one by one and do not need to keep any information between messages. On the contrary, the aggregator is a stateful operator. It needs to store each incoming message until all the related messages have been received. When implementing an aggregator, the following items have to be taken into account.

- Correlation. Which messages are related to each other.
- Completeness Condition. When the aggregator is ready to produce an output.
- Aggregation Algorithm. How do combine the individual messages into a single output.

2.1.3 Enterprise Service Bus

Mason in [32], defines an Enterprise Service Bus (ESB) as an architecture which comprises “a set of of rules and principals for integrating numerous applications together over a bus-like infrastructure”. The concept of an ESB evolved from the needs to combine message-oriented middleware, web services, routing intelligence, and transformation capabilities in order to facilitate integration [34].

The ESB is related to many of the patterns described before. The ESB requires that all the applications using the bus to share data use the same Canonical Data Model. In order to route the message to the appropriate destination the ESB may relies on different Message Routers. The applications that were not designed to interface with MOMs need a Channel Adapter in order to interface with MOM used by the bus.

Figure 2.10 shows an overview of an ESB architecture. As we can see from the picture, the ESB is a concept used to decouple applications. Such decoupling is usually achieved using a message-oriented middleware. The data that flows on the bus need to be transformed into a canonical data format, and those transformations are managed by the ESB adapters. The ESB adapter is responsible for communicating
with the back end applications and convert the data from the applications’ format to the bus format.

Figure 2.10: Overview of an Enterprise Service Bus architecture

**ESB features.** Menge in [34], presented different features that an ESB architecture have to provide in order to serve as a robust integration solution.

- **Invocation.** An ESB has to be able to send requests and receive responses from integration services and integrated resources. It has to support different standards including SOAP, JMS, and other MOMs. In addition, an ESB must support communication mechanisms such as TCP, UDP, JBI, RMI, HTTP, SSL, FTP, among others. This is related with the channel adapter pattern described in section 2.1.2.
• Routing. An ESB should allow to decouple the data sources from the final destination by means of routing capabilities. The decision to which destination a message has to be sent can be based on both pre-configured rules and dynamically created requests. An ESB usually offers different types of routers that implement the patterns proposed in [23]. All those routers can be combined in order to create complex processing workflows.

• Mediation. This refers to the data transformation capabilities that an ESB has to have. These transformation are needed to translate specific data formats into a canonical format understood by the bus. The patterns described in the section 2.1.2 are usually implement in an ESB in order to provide mediation capabilities.

• Monitoring & Administration. The aim of a ESB architecture is to provide a simple solution for data integration. Hence, an ESB should provide an easy method of monitoring the performance of the system, the flow of messages through the ESB architecture, and a simple means of managing the system in order to deliver its proposed value.

ESB considerations. Even though an ESB is the proposed solution for the current data integration concerns, there are different considerations to be taken into account before its adoption.

• There is overhead associated with the ESB architecture itself [14, 33]. First, defining a canonical data model that fits the needs of all the applications to integrate is not an easy task. A common interface for all the applications is not easy to achieve, some applications may need more parameters or functions, others than the ones defined in the common interfaces. Typically there is something in common for all the applications and then additional payload depending on the variability of each application. Second, the overhead may be at the ESB adapters. Architects have to create one ESB adapter for each application. Hence, it is desirable to design an standard adapter that can be reused and adjusted easily and quickly for new applications.

• The bus may be the single point of failure, but this can be overcome using redundancy [14]. There are open source ESB solutions such as Mule ESB, and Apache ServiceMix that provide all the capabilities described before. However, the open sourced versions do not include features such as fault-tolerance and
high availability. These features can be obtained with special subscriptions or
with the commercial versions of the same tools [52].

- Developers need to learn and get familiar with the concepts behind the ESB
  architecture. Hence, the learning curve associated to its adoption have to be
  considered.

2.1.4 Sensor Data Integration

In the previous sections we presented an overview in terms of Data Integration. Since
our work is particularly related to sensor data integration, we present what other
researchers have been doing in that field.

Global Sensor Networks (GSN). GSN [47] is one of the first attempts on sensor
data integration. The aim of GSN is to provide a reusable software platform for
processing data streams produced by heterogeneous sensor networks. GSN uses event-
based and pull-based models for data acquisition. In the first case, data is sent by
the source and a GSN method is called when it arrives. In the pull based mode, GSN
periodically asks the source for new data.

"The internal behavior of GSN is of a content-based publish-subscribe system, on
which subscribers (virtual sensors) subscribe (using SQL queries) to publishers (wrappers). Content (sensor data) is described as timestamped tuples and stored in tables
in a relational database" [47].

GSN relies on an abstraction so-called “Virtual Sensor”. A virtual sensor abstracts
the implementation details of the data sources to a common data schema. It may
receive input data streams from different data sources, either a physical device, or
another virtual sensor, and produce exactly one output stream in a predefined format.
The format produced is based on the input streams and instructions specified in the
java programing language.

The specification of a virtual sensor is provided in a XML file called Virtual Sensor
Description file (VSD). The VSD file provides the information required for deploying
and using a virtual sensor. The information provided within a VSD includes metadata
used for identification and discovery, the details of the data streams which the virtual
sensor consumes and produces, an SQL-based specification of the stream processing
(filtering and integration) performed in a virtual sensor, the processing class which
performs the more advanced and complex data processing (if required) on the output stream before releasing it, and functional properties related to persistence, error handling, life-cycle, management, and physical deployment.

GSN is a good approach to overcome the challenges of integrating heterogeneous sensor networks. Sensor networks and data streams can be specified using XML, and SQL is used as query and data manipulation language. GSN allows to add and reconfigure sensor networks and facilities dynamically and with no downtime [3].

Although GSN includes interesting features, it does not offer capabilities such as parallelism and fault tolerance. GSN allows distribution since GSN nodes can be placed at different physical locations. Virtual Sensors can receive data streams coming from different sources and pass the streams between distributed GSN servers. However, it does not support parallel processing. It does not allow creating a cluster of machines which execute the same tasks. In addition, the fault tolerance is only at node level; i.e., if GSN detects a faulty virtual sensor or wrapper, it undeploys it and releases the associated resources [3]. This means that in case that one of the GSN nodes in the pipeline breaks down, the whole processing activity would be interrupted due to the lack of automatic failover.

Dong and Schendel, in [12], proposed an Enterprise Service Bus (ESB) architecture to handle the heterogeneity of the data sources. They propose that proper data handling mechanisms are needed and they describe three main challenges of sensor data integration: how to efficiently handle large amounts of data, how to serve the sensor data to different client applications, and how to cope with different data formats. Hence, they propose that data has to be handled while it is being transferred to the client applications. In order to do that, they propose an ESB using a Message-Oriented Middleware to pre-process and transform the sensor data on the fly. They point out that this approach allows distribution, scalability and robustness. They mention that failover mechanisms could be implemented using backup servers but they do not state if such mechanisms are implemented. The authors describe some of the Enterprise Integration Patterns that are implemented within the ESB. It is remarkable that the authors used architectural patterns to cope with the integration challenges, as we presented in section 2.1.2, the integration patterns are proven solutions for recurring integration problems [23]. However, they mention that this approach was not intended for near-real time responses, so latency could be a new
issue. They also state that there is no implementation to prove the concepts of the design.

Koga and Medeiros, in [27], propose a combination of ESB and Complex Event Processing. ESB to handle the heterogeneity of data sources and Event Processing for dealing with sensor data. They use an Event Processing Engine to apply functions such as split, filter, aggregation, projection, etc., over the sensor data streams. They tested their approach in a real scenario with thirteen weather stations producing data.

Busemann et al., in [6], propose the integration of sensor networks with a SOA approach. In addition, they present the results of the evaluation of their proposed architecture. In their architecture they propose two components which deal with data handling. First, the Gateway, provides an interface between the sensor network and the rest of the system. It removes the device-specific details and offers an standardized version of the data. The second component is the Backend. It deals with data processing and data management functionalities. The Backend was designed following the ESB architectural style. When a new sensor reading is pushed to the Backend, that data is consumed by the Data Parser. The parser transforms the data into a valid format and pushes the data back to the ESB. Then, the translated data is consumed from the ESB by other components of the system. This approach facilitates decoupling, since each component of the system is independent and they communicate each other through the ESB. However, as stated by the authors, since one backend has to deal with many gateways and clients in the upper layers connected to it, it may become the bottleneck of the system. In the evaluation, the authors show that as the number of requests to the backend increases, its response time is affected, and it reaches a saturation state at some point.

Motwani et al., in [37] besides an ESB architectural style, they propose the use of open standards for enabling discovery of disperse sensor networks. They integrate the Sensor Web Enablement suite of standards in their work. The authors suggest that real and global scale integration could be achieved if the majority of teams working with sensor data adopt such standards.
2.2 Stream Processing

Stream Processing is a topic that born because of the limitations of traditional processing infrastructures to meet the application’s requirements of real-time responses when processing high volumes of continuous flows of data [45]. In order to determine whether SPFs could help researchers, working on sensor data analysis, to speed up the integration and pre-processing stage we need to understand the theory behind SPFs. We need to identify which are the characteristics that SPFs should have, how SPFs need to be adjusted to handle sensor data specifics, and whether SPFs introduce new challenges that have to be handled by the developers of stream processing applications.

2.2.1 Characteristics of Stream Processing Systems

There are different characteristics that set apart Stream Processing Systems (SPS) from the traditional interactive and batch processing systems [5]. In the following paragraphs we present the features that SPS should have according to the literature in this field.

**Edge adaptation.** SPS process data from different sources and forward the results to different end consumers or sinks. Such consumers may be other systems or visualization tools that allow human interaction. Hence, SPS should provide built-in integration mechanisms with external applications. In other words, SPS should provide adapters to connect sources or sinks with the processing platform, or a programming model that allow developers to create such adapters. The ability of SPS to provide such functionality was coined by Andrade et al. in [5] as “edge adaptation”.

**Data-in-motion and long-running analytics.** This characteristic refers to the ability of SPS to process data streams on the fly, i.e., as the data flows through the system. It is desirable that SPS do not rely on costly storage operations (e.g., database commits) in order to process the incoming data. This ability of processing information as data flows allows SPS adapt to changes, and it facilitate them to provide results in near-real time. In addition, SPS are designed to perform long-running analytics, i.e., they have to function automatically and never stop its execution. Hence, SPS
have to include adaptive resource management techniques such as load balancing, load shedding, and they also have to be resilience to transient failures [5].

**Query model and operators for continuous processing.** SPS can used to perform real-time computations based on the input streams, or to detect interesting patterns in the streams. In both cases SPS should provide a query model and primitive operators that allow to perform processing on continuous data streams. The query model (SQL) of traditional database management systems was intended to be executed over the whole data residing in a persistent storage. However, the intrinsic continuous flowing of data within SPS impose the need of defining new constructs and stream-specific operators [9, 45]. To determine which portions of data need to be considered during the execution of operators SPS need to include the concept of *windows*. Windows are not operators but rather a language construct that can be applied to operators to limit the scope of their action [9]. SPS should provide different types of windowing mechanisms. The general windows constructs are the time-based and count-based windows [20]. In the first case, the bounds are defined as a function of time, e.g., an operation will be computed only over the elements that arrived during the last hour. In the latter case, the bounds are based on the number of messages, e.g., an operation will be applied only over the last 10 messages. The most common windows mechanism are sliding and tumbling windows [9, 20]. Sliding windows advance the lower and upper bounds when new messages enter the system. Sliding windows evict the old messages to maintain the windows size, and they produce an output every time it advances certain sliding interval. For example, figure 2.11a depicts a sliding window that keep the last 5 messages and process them every time that 2 new messages arrive. Tumbling windows are a special type of sliding windows where the window size and sliding interval are the same, i.e., lower and upper bounds move by \( k \) positions, as \( k \) messages arrive, and all the messages are flushed every time it produces an output. Figure 2.11b depicts an example of tumbling window with \( k=5 \). As we can see, depending on the size and sliding parameters, the windows can be overlapping or disjoint.

**Historical and streaming data integration.** It is desirable that SPS allow processing both, historical and streaming data. It should facilitate seamlessly to switch from historical data to live data feeds [45]. This is particularly needed when researchers use current and historical data streams to create prediction models. Another use
of this capability is when researchers need to test their algorithms. They can use historical data to see how well their algorithms work. After the algorithms are validated, SPS should just switch to live data streams without changing the application code.

**High performance and scalability.** SPS born because of the need for processing high volumes of streaming data with low latency. Expressed quantitatively, this means that SPS should process tens to thousands of messages per second with a latency in the range of micro to milliseconds [45]. To meet such requirements, SPS should scale up and out, i.e., they should provide vertical and horizontal scalability. Vertical scalability means increasing the performance of the system as the computational capacity of the machine where the system runs increase. Hence, SPS should allow multi-threaded operations to take advantage of the current multi-processor computer architectures. SPS also should take advantage of additional bandwidth and memory when these resources become available. Horizontal scalability means increasing the performance of the system by splitting the execution of the application over multiple machines. SPS should support data parallelism to take full advantage of this horizontal scalability. This is useful to avoid bottleneck operators when the volume of input streams or the complexity of processing increases. SPS should provide built-in parallel processing capabilities without the developers having to write low-level code. SPS should allow developers only to specify where and how much data parallelism is needed and the system automatically should partition the incoming data, instantiate wires and redirect the data streams to the appropriate replicated operators, and finally merge the results.

**High Availability and Fault tolerance.** SPS should provide data integrity for the mission critical information in order to avoiding disruptions in its execution. Industry and academy using SPS require that their applications stay up and running all the time, even in the presence of transient failures. Restarting servers and recover from a
backup is something not acceptable in real-time analysis because this will incur in too much overhead. Hence, high-availability is an essential feature in stream processing applications. However, “developing fault-tolerant applications, even with the help of a fault-tolerant SPS, is potentially a much more challenging problem ” [5]. Active replication and checkpointing are the two techniques most common used to build fault-tolerant applications [5,45]. In the active replication mode the application, or the critical parts of it are running in a separate hosts, an the SPS provide the ability to switch from a faulty replica to a normal one in the presence of failures. When checkpointing is used, the application state is periodically checkpointed and in the presence of failures the state is restored from a previous stable snapshot. When designing a stream processing application, it is necessary to evaluate this features provided by the SPS middleware and made trade-off depending on the particular needs. Such trade-offs are needed because the solutions to achieve high-availability may incur additional overhead. So performance might be affected by the selection of one particular approach.

2.2.2 Evolution of Stream Processing Systems

SPS. There are different solutions in the field of Stream Processing Systems. The first contributions in this field were Aurora/Borealis [1,2], STREAM [36], and TelegraphCQ [7]. These systems were focused on scalability, but their input consisted on homogeneous sources with a predefined schema [46]. Hence, integration was not a requirement for those systems. These systems provided the foundations for the next generation of stream processing frameworks.

The next generation of SPS introduced a tighter relation between the concepts of streams and relational databases. STREAM’s Continuous Query Language (CQL), an extension to SQL:1999, introduced the concept of “stream” in addition to “relation”. Moreover, besides relation-to-relation operations, it introduced stream-to-relation and relation-to-stream operations [46]. Later on, different commercial systems adopted this SQL-based language to combine data from multiple sources in order to infer events from patterns in the data streams. This evolution brought what is known as Complex Event Processing Systems, e.g. StreamBase CEP, Oracle CEP, SQLstream, and Esper.

The last generation of SPS is focused on providing high level APIs in order to make it easier for developers to focus on the logic of the application rather than in the
associated complexity of distributed programing (e.g., managing state, data transport, distribution of application logic, etc.). The state of the art stream processing systems are Storm [41], S4 [25], and Samza [26].

**Storm.** It is defined as a “distributed realtime computation system” [41]. It was driven by the need of achieving near-real time response while processing high volumes of data. Storm provides inherent support for parallelization, fault-tolerance, and guaranteeing message processing. The main abstractions of storm are *tuple, stream, spout, bolt and topology*. A tuple is the main data structure in Storm. A tuple is a named list of values, where each value can be any type. A stream is an unbounded sequence of tuples. A spout is a producer of streams. It reads tuples from an external source (e.g., a WebService, Messaging Systems, etc.) and ingest those tuples into the system. Bolts are the basic processing unit in Storm, they can do functions like filtering, aggregations, joins, database access, etc. A topology is a graph of spouts and bolts that represent the logic of a stream processing application, i.e., topologies allows to assemble all these components together. Storm uses two kind of nodes to run a topology. A master node runs a daemon called Nimbus. It is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures. Each worker node runs a daemon called the Supervisor. The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it [31]. These two processes are coordinated through Zookeeper [17].

**S4.** It is a general purpose stream processing system inspired by the MapReduce model. Its design is a combination of MapReduce and the Actors Model, and it was driven by the need of large scale applications for data mining and machine learning [38]. The main capabilities of S4 are scalability and high availability, but it offers a partial fault-tolerance [38]. Their model is based on two abstractions, the “Event” and the “Processing Elements (PEs)”. Events are represented as a tuple in the form \((K, A)\), where K is the event’s key and A are the attributes associated to that event. Processing Elements are the basic computational unit in S4, PEs may execute one or both task: 1) emit one or more event which may be consumed by other PE or 2) publish results. Each instance of a PE has four components. *Functionality*, which is defined by a java class and associated configuration. The *type of events* it consumes, the *key* of the events, and the *values* corresponding to the keyed attributes. A special
type of PEs are those which are keyless. They consume all the events of the type to which they are associated. This kind of PEs comprises the input layer of a S4 cluster, where all the events are assigned a key. Processing Nodes (PNs) are the logical containers of PEs. S4 routes events to PNs based on the keyed attributes of the events. In this way all the events with a particular value for the key attribute are guaranteed to arrive at a particular corresponding PN. S4 uses Zookeeper for coordination between nodes in the S4 cluster. In order to create S4 applications, developers has to write java code for tasks to be executed by the PEs. Developers have to implement an input event handler processEvent and an output function output to specify the output events. Finally, the PEs are assembled into applications using the Spring Framework.

**Samza.** It is another distributed stream processing system created at LinkedIn, and now open sourced as an Apache Incubator Project. Samza offers managed state, fault tolerance, message durability, scalability and resources isolation. The core concepts of Samza are streams, jobs, partitions, tasks, data flow graphs, and containers. Streams are immutable messages of the same type. A job is a code that performs transformations on a set of input streams to append output messages to set of output streams. Partitions, are small portions of a stream, where each partition is a a totally ordered sequence of messages, and each position in that sequence has a unique identifier. A task is the unit of parallelism of the job. A job is itself distributed by breaking it into multiple tasks. The number of task a job can have is fixed and it is determined by the number of partitions, and the partitions assigned to a task will never change. A Samza application can be composed by multiple jobs as a data flow graph where the nodes are streams containing data and the edges are jobs performing transformations. Samza is a layered system comprised by a streaming layer, an execution layer, and a processing layer. Apache Kafka [16] for the streaming layer, YARN [15] for the execution and the Samza API for processing. The creators of Samza say that those three layers are pluggable, hence developers can use other systems instead of YARN and Kafka if the need for that arises.

### 2.2.3 Data Stream Processing vs. Complex Event Processing

Data Stream Processing (DSP) and Complex Event Processing (CEP) are two similar concepts that refer to systems that fall into the category of Stream Processing Systems (SPS). Even though both type of systems are used to process unbounded flows of data
as it flows from the sources to the system, we consider that it is important to specify
the difference between them.

Data Stream Processing is a generic concept used to denote systems that focus on
processing flows of data and perform data transformations [9]. On the other hand,
Complex Event Processing systems treat data streams as a stream of events. These
systems are focused on matching complex patterns in the streams and notify about
the occurrence of particular events to the interested parties [9, 42].

Although there is a clear distinction between both types of systems, at least concep-
tually, in practice there are many frameworks and tools that integrate capabilities of
both domains. We can see that evolution of the SPS presented in section 2.2.2 include
features to facilitate data processing, data transformation, events detection, distrib-
uted processing and so forth. These convergence of domains have motivated some re-
searchers to propose a common terminology to group all these kind of systems [9, 44],
and others to categorize the state of the art stream processing systems [11].

Some SPS provide capabilities that other does not provide and vice versa; nonetheless,
all of them are useful in different scenarios. Hence the adoption of one specific type
of system depends on the particular needs of the users.

From the contents in section 2.2.1 we find out that not all the tools described in the
overview offer all those characteristics. For instance, a windowing mechanism is a
feature that not all the tools have but the tools offer the abstractions for implement
such mechanism. Some of the tools which offer such capabilities require user to pay
for it. On the other hand, the tools described here were not designed with the need
for integration in mind. Hence, the data integration capabilities is something that
has to be added. The tools which are intended for data integration present more
complexity when parallel processing is required, and they do not offer fault-tolerance.
Once again, those tools which offer such capabilities require especial subscriptions.
Therefore, we could use one of those tools and make some adjustment in order to
they fit for data integration and pre-processing of sensor data.
Chapter 3

Problem Definition

Our main research question is whether stream processing techniques may be used for integrating and pre-processing data from heterogeneous sensor networks. Therefore first we need to understand the terminology used in the field of stream processing. Then in section 3.2.1 we try to answer the question of how stream processing systems can provide data integration capabilities. In the sections 3.2.2 and 3.2.3 we try to answer our second and fourth research questions: How to adjust Stream Processing Systems in order to cope with sensor data specifics and how to deal with the challenges that the parallelism offered by Stream Processing Systems impose.

Hence, in this chapter we present the semantics of data streams, the definitions of the problems we need to solve and the algorithms for the proposed solution.

3.1 Basic definitions

In this section we present the basic terminology to understand the context of our problem, and adapt some terminology to the sensor data domain.

Definition 1. A raw input dataset $R$ contains the data produced by sensor devices at a particular instant of time $t$. There are different representations of the input dataset based on observations about how the data is structured within it.

- **Set.** This is the most common representation of an observed input dataset. $R$ is a set of values, and each value represents a different reading produced by a sensor device. It can be denoted as:
\[ R = \{ r_1, r_2, ..., r_n \} \]

- **Matrix.** In this case \( R \) is a matrix of \( M \times N \), where each row represents a device with many sensors attached to it, and those devices are grouped in such a way that every time an output is produced all of them provide a sensor reading. The matrix contains all the measurements produced by all sensor devices. It can be denoted as:

\[
R_{m,n} = \begin{pmatrix}
  r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\
r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\
  \vdots & \vdots & \ddots & \vdots \\
r_{m,1} & r_{m,2} & \cdots & r_{m,n}
\end{pmatrix}
\]

- **Superset.** In this case \( R \) is a collection of the values of the non overlapping subsets \( W_i \). It can be denoted as:

\[ R = \{ w_1, w_2, ..., w_n, W_1 \cup W_2 \cup ... \cup W_n \} \]

where \( w_i \) is the number of elements of the subset \( W_i \), and \( \{ W_1 \cup W_2 \cup ... \cup W_n \} \) is a pairwise disjoint collection of subsets \( W_i \), i.e., \( \bigcap_{i \in I} W_i = \emptyset \).

**Definition 2a.** The time domain \( T \) is an ordered, infinite set of discrete time instants \( \tau \in T \).

**Definition 2b.** Given \( j \in \mathbb{N} \) where \( j \) is an increasing sequential integer, a *stream tuple* \( (A^T_j) \) is an instance of data produced at the time unit \( t \), and represented as a tuple comprised of timestamp, key, and value. It can be denoted as:

\[ A^T_j = (\tau_j, k_j, V_j) \]

where \( V_i \) may be a data structure of any type.
Definition 3. Given \( i \in \mathbb{N} \) where \( i \) is an increasing sequential integer, a stream \((S_i^\varphi)\) is an unbounded sequence of tuples \((A_j^\tau)\) with the same schema \(\varphi\), being continuously produced every \((t)\) time interval. It can be denoted as:

\[
S_i^\varphi = A_1^\tau, A_2^\tau, ..., A_\infty^\tau.
\]

3.2 Problems definitions and proposed solutions

Now that the basic definitions were presented in section 3.1, we need to define the problems that we are trying to solve and the proposed solutions to overcome such problems.

3.2.1 Data transformations

Our research was motivated by the need for integration of data streams coming from heterogeneous sensor networks. Hence, such heterogeneity is the first problem that we need to cope with. From literature presented in section 2.1, we observed that we need to transform the data being produced by the sensor devices into a canonical data model that can be understood by the applications to be integrated. The first problem is to define a processing workflow that depicts the function or a set of functions required to achieve such data transformations.

Problem 1. A data transformation function or set of functions need to be defined in order to transform the input dataset into stream tuples. It can be denoted as:

\[
R \rightarrow A_j^\tau
\]

At the data source side, the sensor devices physically located at the sensed areas are continuously producing input datasets \((R)\). Such datasets have to be ingested into the sensor data integration platform and transformed to stream tuples \((A_j^\tau)\) before to be transferred to the analysis platform. This generates different streams \((S_i^\varphi)\) that flow through the system.
**Proposal 1.** Create a stream processing application that receives an *input dataset* \((R)\), passes it through a set of *data transformation* operators \((\rightarrow)\) and produces an *stream tuple* \((A^\tau_j)\) as an output.

The data transformation operators should implement the integration patterns presented in section 2.1.2. The entire application is a composite DFG (Data Flow Graph) that encapsulate fine grained functionality.

Algorithm 1 shows a pseudo code of a canonical stream processing application.

```python
# Algorithm 1: canonical in-stream data integration application

Input: A raw input dataset \(R\)

Output: A stream \(S^\phi_i\)

1. \(input \leftarrow R\)
2. \(output = DataTransformation(input)\)
3. \(return\ output\)
```

### 3.2.2 Multiple Stream Aggregates

As we could observe high performance and scalability are important characteristics of the SPSs [5]. In addition, we observed that current SPSs allow to create distributed applications which exploit the parallel processing capabilities provided by the new multi-processor computer technologies, and the fact that they can be deployed in a cluster of machines [38,41]. With the parallelism provided by current SPSs we gain performance since the applications can process multiple *streams* \((S^\phi_i)\) at the same time. However, these new capabilities impose new challenges.

One common operation when processing data streams is the use of aggregates [56]. However, how we guarantee that the data to be aggregated arrives at the same place when our data is split and distributed to different processors or even different physical machines. In other words how to ensure that our aggregation algorithms will receive the relevant information.

Some SPSs provide built-in grouping mechanisms that allow easily to specify where to send the *stream tuples* \((A^\tau_j)\), but they still leave to developers to deal with the fact that different streams may arrive to the same task (thread). Therefore, they have to create a mechanism that allows them to keep separately aggregates of *stream tuples* that belong to different *streams*. 
Definition 4. An aggregation function $\text{Agg}$ is a function that performs a computation over a set of values and returns a single value or a list of values which satisfy some given condition. It can be denoted as:

$$S^\varphi_i \rightarrow S^\varphi_k$$

For illustration purposes let’s assume that we have one aggregation function $F$ running in parallel in two threads within one machine. This means that our function $F$ becomes $f_1$ and $f_2$ at runtime. In addition, we assume that the SPS provides the grouping mechanisms.

Problem 2. Given the streams $S^\varphi_1$, $S^\varphi_2$, $S^\varphi_3$, $S^\varphi_4$; its corresponding stream tuples arrive at any time and in any order to the aggregation function. Because of the grouping capabilities provided by the SPS, the stream tuples belonging to $S^\varphi_1$ and $S^\varphi_3$ goes always to $f_1$; and the stream tuples belonging to $S^\varphi_2$ and $S^\varphi_4$ goes always to $f_2$. The application should allow developers to create just one aggregation function, and automatically instantiates it as many times as stream tuples which belong to different streams ingress to the function.

\begin{center}
\begin{tikzpicture}
    \node [state, initial by arrow] (A) {$S^\varphi_i$};
    \node [state, right of=A] (B) {$S^\varphi_{i-agg}$};
    \node [state, below of=A] (C) {$S^\varphi_i$};
    \path[->] (A) edge node {$\text{Agg}$} (B);
\end{tikzpicture}
\end{center}

Figure 3.1: Logical view of the grouping aggregation

Figure 3.1 shows the logical view of the aggregation. Developers only need to create one aggregation function that can be applied to different streams. On the other hand, figure 3.2 shows how the aggregation is mapped to physical processes. The parallelism and data partition is managed by the SPS; however, the multiple instantiation of the aggregate function (boxes above Agg $f_1$, and below Agg $f_2$) have to be implemented in order to provide a seamlessly partition of the data streams.

Proposal 2. Create an aggregation mechanism that maintains a list of active aggregates until a completeness condition is met, then apply the aggregation function and produce the results.

The aggregator should keep a list of active aggregates, i.e., aggregates for which the aggregator has received stream tuples already. Every time a new stream tuple arrives,
the aggregator has to check if a stream tuple belonging to the same stream has arrived before. If so, it adds the stream tuple to the existing aggregate. Otherwise, the aggregator creates a new aggregate and then adds the incoming stream tuple. After a stream tuple is added to an aggregate, the aggregator has to check if the completeness condition is met. If the condition holds, the aggregator should apply the aggregation algorithm and publish the results. If the completeness condition is not met, the aggregator should wait for subsequent incoming stream tuples.

The algorithm 2 depicts the pseudo code of the algorithm for a multiple streams aggregation. On initialization we create an empty list of the current aggregates, line 3. The second part of the algorithm deals with a new stream tuple arrival, lines 4–10. Finally, the last part is responsible for checking the completeness condition and publishing the results, lines 12–19.

Figure 3.3 shows an example of how the aggregator operates. In the example, stream tuples coming from two different streams have to be aggregated separately. The tuple $\langle t_j, k_j, V_j \rangle$ in the picture represents the timestamp, key, and value of the arrived stream tuple as described in definition 2b.

We assume that the key attribute of the stream tuple is the correlation id. In addition, we assume that the aggregation algorithm has to be executed when at least three stream tuples have arrived. Only after that, the results of the aggregation can be forwarded as an output.

### 3.2.3 Windowing Mechanisms

We can see a streaming sensor data integration application as an continuous flow of stream tuples being processed as they flow from the data sources to the destination.
Algorithm 2: Multiple aggregates

\textbf{Input}: Stream tuples $A_j$
\textbf{Output}: Stream tuple $A_k$

1. On Init
   
   $Map < ID, List < StreamTuple > > \ currentAggregates \leftarrow \{\}$

2. On stream tuple arrival
   
   if currentAggregates contains key then
   
   add the stream tuple to the corresponding list
   
   else
   
   create a new entry in currentAggregates
   
   add the stream tuple to the new entry

3. After a new stream tuple is added
   
   // check completeness condition
   
   if isComplete = true then
   
   apply aggregation algorithm
   
   publish results

   else
   
   do nothing

Figure 3.3: Example of the aggregator’s operation

Since streams are infinite sequences of stream tuples (See definition 3), it is not possible to analyze or execute queries on the entire sensor data stream. Therefore, we need to rely on a data model that allows to capture finite and relevant subsets of the entire stream. The common approach to process finite subsets of data streams is the use of Windows [9, 10, 20, 56].

Definition 5. A window $W$ with window length $L$ and window size $\delta$, is a subsequence of a stream with well defined scope, i.e., it has lower and upper bounds. $L$ refers to the timespan, or the capacity in number of items covered by the window (e.g., one hour, or ten tuples). On the other hand $\delta$ refers to the current number of
items in the window.

In problem 2, we presented the need of keeping the state of multiple aggregates until a condition is met. From definition 5 we see that each of such aggregates can be modeled as a window. Hence, a window allows to determine the completeness condition for an aggregation operation every time that the upper bound is reached.

**Definition 6.** Given \( \tau \in \mathbb{T} \) where \( \tau \) is an increasing timestamp of a stream tuple \( A_\tau^j \), a **time-based Window** \( W_\tau \) is a window which bounds are defined in function of time, i.e., the stream tuples that have arrived in the last \( (\tau) \) time units belong to \( W_\tau \), (e.g., tuples that have arrived in the last hour). It can be denoted as:

\[
W_\tau = \{ A_1^\tau, A_2^\tau, ..., A_j^\tau \}
\]

**Definition 7.** Given \( \eta \in \mathbb{N} \) where \( \eta \) is an increasing counter, a **count-based Window** \( W_\eta \) is a window which bounds are defined in function of the number of stream tuples that have arrived, i.e., the last \( \eta \) stream tuples that have arrived belong to \( W_\eta \), (e.g., a window of the last fifty tuples). It can be denoted as:

\[
W_\eta = \{ A_1^\tau, A_2^\tau, ..., A_{\eta-1}^\tau, A_\eta^\tau \}
\]

**Definition 8.** A **sliding window** \( W_s \) is a window (e.g., \( W_\tau \) or \( W_\eta \)) which updates its lower and upper bounds every time a new stream tuple arrives, and **window length** = **window size** (i.e., \( L = \delta \)). It discards old stream tuples in order to maintain the window length, and produces an output every time a condition is satisfied.

**Definition 9.** A **tumbling window** \( W_r \) is a window (e.g., \( W_\tau \) or \( W_\eta \)) which keeps a buffer of the incoming stream tuples until the windows is full, i.e., \( L = \delta \). Then it produces an output and flushes the buffer to start all over again.

**Problem 3.** Given a stream \( S_\tau^\phi \), create a window mechanism that allows to implement \( W_\tau \) or \( W_\eta \) in order to keep aggregates of the last stream tuples arrived, and produce an output when a trigger condition is satisfied.
Proposal 3a. Implement a sliding window $W_s$ such that given a $window$ $W$ with $\text{window's length} = L$ and a trigger condition $\Omega$, keeps a buffer with the last $L$ stream tuples and produce an output every time $\Omega$ is satisfied. Notice that $L$ and $\Omega$ can be time-based or count-based, and they are independent from each other, i.e., a stream tuple that belongs to the window $W_L$ may also belong to the window $W_{L+1}$ if the trigger condition is reached again before the old stream tuple is evicted. In other words, there may be overlapping windows.

In the case that $\Omega$ is count-based, an additional trigger condition $(\Gamma)$ might be added in order to avoid that the window becomes a bottleneck waiting indefinitely for incoming tuples. Since we are dealing with highly distributed applications and parallel processing, some tuples can fail and being replayed. Other tuples might be delayed because of the difference in computational capabilities of the servers hosting the application. Therefore, we may include an external signal that will trigger the execution of the operation and force to produce an output.

In addition, it is important to discuss how to define the eviction policy, i.e., discard old stream tuples and maintain the window’s length constant. In the case that $L$ is count-based, the definition of the eviction policy is straightforward. Each time the number of stream tuples is equal to the windows length (i.e., $\delta = L$), we have to advance the window. When $L$ is time-based, additional considerations have to be taken into account. The eviction policy could be based on explicit or implicit timestamps $(t_i)$. An explicit timestamp is inserted by the source when the stream tuple is created. An implicit timestamp is inserted by the application when a stream tuple arrives to the aggregator. Let’s consider a window with a length $L = t_{\text{milliseconds}}, t_0$ the timestamp in milliseconds of the first tuple of the window, and $t$ the timestamp in milliseconds of the current tuple. Then, $\delta = t - t_0$. In both cases, explicit or implicit timestamps, a window is considered full when $\delta \geq L$. Notice that when timestamps are implicit, we need additional memory space to store the timestamp assigned to each tuple.

Figure 3.4 shows an example of how the proposed implementation of the eviction policy works in the case of count-based and time-based windows. Figure 3.4a depicts a count-based window with a window length $L = 5$ stream tuples. The window size $\delta$ is a counter that keep track of the number of stream tuples already in the window. Each time $\delta = L$ the oldest stream tuple has to be discarded and the new one appended to the window. Figure 3.4b depicts a time-based window with a window length $L = 60 \text{seconds}$. The time stamps could be explicit or implicit. The window’s size $\delta$ is the difference between the current time stamp and the timestamp of the first
**stream tuple** in the window. The bullets (●) represent the **stream tuples** inside the window. Notice that in the case of count-based windows, the number of **stream tuples** is always the same for each window. On the other hand, in time-based windows the number of **stream tuples** may vary for each window, as shown in 3.4b. The number of **stream tuples** that fall within a window depends on the rate at which the **stream tuples** arrive to the aggregator component.

![Image](image-url)

(a) Count-based window. \( L = 5 \)  
(b) Time-based window. \( L = 60 \text{ seconds} \)

Figure 3.4: Examples of count-based and time-based windows

The algorithm 3 shows the pseudo code of the Sliding Window implementation. The algorithm receives the **stream tuple**, the windows length and the completeness condition. On initialization, creates the window’s buffer and set the external termination signal to false, lines 2–6. When a **stream tuple** arrives, the size of the window is calculated depending on whether the window is count or time based, lines 8–13. Then it checks if the window is full. In that case, the head of the window buffer is removed and the incoming **stream tuple** is added to the buffer, lines 16–20. When a termination signal is received, the termination variable is set to true, lines 22–23. Finally, the algorithm checks for the completeness condition or the termination signal. If any of those conditions hold, the variable for termination signal is set to false and the window aggregation is returned, lines 26–29.

**Proposal 3b.** Implement a tumbling window \( W_\tau \) such that given a window’s size \( L \) keeps a buffer with the last \( L \) **stream tuples**. The same conditions for \( L \) than those proposed in 3a apply here. Moreover, an external signal could be also implemented in this case to avoid bottlenecks when \( L \) is count-based.

### 3.2.4 Time Alignment

The sensor networks producing **raw input datasets** comprise multiple sensors devices measuring different physical phenomena. Researchers working on sensor data analysis
Algorithm 3: Windowing

Input: Stream tuples $A_j^T$
Output: List<Stream tuple>

1. On Init
2. initialize the windows length $L$
3. initialize the completeness condition
4. List<StreamTuples> window = {} 

5. On stream tuple arrival
6. if $windows.type = countBased$ then
7. \hspace{1em} size = window.size
8. else if $windows.type = timeBased$ then
9. \hspace{1em} $t_0 = window[head]\text{.timestamp}$
10. \hspace{1em} size = st.timestamp - $t_0$

11. window.add(st)

12. if $size \geq length$ then
13. \hspace{1em} window.remove[head]

14. After a new stream tuple is added
15. //check completeness condition
16. if $isComplete = true$ then
17. \hspace{1em} return window;
18. else
19. \hspace{1em} do nothing

Scalable integration and pre-processing of sensor data streams
need to correlate the sensor data to create models based on those sensor readings in order to discover certain events [28]. However, the sensor devices do not capture the data at the same time. Likewise, the values produced by each sensor device may not be equally distributed over time. Therefore, a time alignment mechanism is necessary in order to correlate the data coming from different sensor devices.

**Definition 10.** An alignment function $F_{\text{align}}$ is a function such: given $N$ streams $(S_{1}^{\varphi}, S_{2}^{\varphi}, ..., S_{N}^{\varphi})$ having stream tuples generated at different intervals, produces a stream $S_{\text{aligned}}^{\varphi}$ which stream tuples are in the form $A_{\text{aligned}}^{\varphi} = \langle t_{\text{aligned}}, K, V_{\text{correlated}} \rangle$; where $t_{\text{aligned}}$ is a target timestamp at which the input streams have to be aligned, $K$ is a key, and $V_{\text{correlated}}$ is a set with the values, of each stream, at the time $t_{\text{aligned}}$.

Figure 3.5 depicts the alignment function applied to $N$ input streams. The bullets ($\bullet$) represent the stream tuples at the time they were sampled. As can be seen in the image, each stream have stream tuples sampled at different time. For simplicity, we omitted the word “aligned” in the image and $t_0, t_1, ..., t_i$ represent the target timestamp $t_{\text{aligned}}$ at different time intervals. On top of the image, we depict the desired output which is a stream tuple with a single timestamp for all the correlated values ($\times$), one from each stream, at the time $t_i$.

![Figure 3.5: Time alignment of N streams](image-url)
Problem 4. Given N streams, $S_1^\phi$, $S_2^\phi$, ..., $S_n^\phi$, that need to be aligned, create an alignment function $F_{align}$ that allows to correlate values based on a target timestamp $t_{aligned}$, as described in Definition 10.

Proposal 4. Implement an alignment mechanism that keeps a time-based tumbling window $W_\tau$ of length $L$ for each input stream and every time a trigger condition $\Omega$ is met, an alignment function is applied over the window to estimate the values at the time $t_{aligned}$ for each input stream and produce stream tuples in the form $A^\tau_{aligned} = \langle t_{aligned}, K, V_{correlated} \rangle$.

From figure 3.5 we can observe that there is no value sampled at the target timestamp $t_i$ (represented as $\times$). Therefore, we need to use an interpolation method to estimate a value for the unseen time points. In order to estimate the missing values we need a discrete set of known data points, and apply an interpolation method over those data points [18,22]. Hence, we create windows with lower and upper bounds $\tau_j$, $t_j+k$ respectively, that span the target timestamp $t_i$, i.e., $\tau_j \leq t_i \leq t_{j+k}$. Such windows will keep the stream tuples required to estimate the missing values, and the alignment function use those values to produce the desired output.

![Diagram of time alignment](image)

Figure 3.6: Time alignment of N streams using windows of data

Scalable integration and pre-processing of sensor data streams
Figure 3.6 depicts the proposed solution using a windows mechanism. A time-based window of arbitrary size is defined in order to keep data points used to estimate missing values. The only condition is that the target timestamp \( t_i \) should be contained in the window \( (t_j \leq t_i \leq t_{j+k}) \). In order to interpolate the sampled values, we assume that their timestamps are strictly increasing. Hence, if that is not the case, a re-ordering algorithm need to be applied in a step prior to the alignment function.

The algorithm 4 shows the pseudo code of the algorithm for time alignment implementation. On initialization it creates a map that will keep the windows for each input stream, and the list for the correlated values, lines 2–4. Every time a stream tuple arrives check if there is a entry in the map for that input stream. If not, first it creates a entry in the map, otherwise it pass the tuple to the TumblingWindow algorithm, lines 6–12. If the trigger condition is met, it iterates the map. If the window contains a stream tuple with the target timestamp, it adds the tuple to the list of correlated values, otherwise it pass the window to the interpolation method in order to estimate the missing value, lines 14–21. Finally it return the aligned stream tuple, line 23.
Algorithm 4: Time Alignment

**Input:** Stream tuples $A_j^\tau$

**Output:** Stream tuple $A_k^\tau$

1. **On Init**
   
   2. Map$<$StreamID, Window$>$ $W = \{\}$
   3. List$<$values$>$ correlatedValues = \{\}

5. **On stream tuple arrival**

6. //check if other tuples of the same stream already arrive

7. **if** $W$ not contains StreamID **then**
   
   8. create a new entry in the map
   9. Map.add(Window(StreamTuple,length, complete))

11. **On completeness**

12. **while** $i \leq Map$.size **do**

   13. **if** $W[i]$ contains a tuple with the target timestamp **then**
      
      14. correlatedValues.add(StreamTuple.value)

   16. **else**
      
      17. estimatedValue $\leftarrow$ interpolation($W[i]$)
      
      18. correlatedValues.add(estimatedValue)

   18. $i \leftarrow i + 1$

20. **return** aligned Stream Tuple
Chapter 4

System Design

In chapter 3 we defined the problems that we try to solve in this research and described the proposed solutions. In this chapter we present the design of a system which is a prototype of our proposed solution. In addition, we present the system’s goals, the key drivers, the functional and non-functional requirements and finally we describe the components of our system architecture.

4.1 High level goals

The In-Stream Integration System has two main goals. First, reduce the deployment time when a new sensor network needs to be integrated into the analysis platform. Second, provide inputs to the analysis platform in near-real time, by increasing the rate at which sensor data is being pre-processed.

Our system supports these goals by first, providing components that facilitate integration tasks, and allowing developers to assemble such components using a declarative language; and second, by exploiting the parallel processing capabilities of SPF s which allow to deploy our system in a cluster of machines with commodity hardware.

4.2 Key Drivers

Based on the high level goals, we found the most important quality attributes of our application to be performance, scalability and robustness. In the next paragraphs we present the rationale of why we chose these attributes as key drivers.
Performance. The system’s goals are both related with performance. The aim of the system is to provide responses in near-real time. The pre-processing middleware should not be the bottleneck of the whole analysis platform. Hence, performance is essential.

Scalability. In order to achieve better performance, the system should allow to scale vertically and horizontally, and thereby exploit the parallel processing capabilities of the stream processing systems. The system should allow that the number of new external partners providing sensor data increases over the time. It will also allow that the number of sensors that comprise a sensor network increases over the time. Any of these two situations should not hinder the system’s performance.

Robustness. The system should be resilience to transient failures. There are several reason why the system might fail during its execution time, however it is desirable that the system include fault-tolerance mechanisms to avoid cope with such failures and avoid a disruptions during runtime.

4.3 Requirements

In this section we present the functional and non-functional requirements of our system. The requirements were prioritized using the MoSCoW method [8]. The MoSCoW method propose the following categories to prioritize requirements.

Must. Describes a requirement that have to be satisfied in the final deliverable in order to consider the solution as successful.

Should. Describes a requirement that is important for the project success, but it does not necessarily need to be included in the current delivery time schedule to consider the project a success. In other words, it is not time-critical or it allows workarounds, so these kind of requirements can be satisfied in a future delivery.

Could. These are requirements which are less critical than the ones within the two first categories. They are nice to have but not essential, and usually they increases user satisfaction with little development costs.
Will not. Describes a least-critical requirement, or a requirement that is not appropriate at that time. They are not included in the current delivery schedule but can be considered for future delivery timeboxes. Nonetheless, it does not make them less important.

### 4.3.1 Functional Requirements

Table 4.1 depicts the functional requirements of the system to be.

**Table 4.1: Functional requirements**

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR-01</td>
<td>The system must connect with external sensor data producers.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-02</td>
<td>The system should inject raw input datasets to the pre-processing platform.</td>
<td>Should</td>
</tr>
<tr>
<td>FR-03</td>
<td>The system should persist the raw input datasets for a t period of time, which is parameterizable.</td>
<td>Should</td>
</tr>
<tr>
<td>FR-04</td>
<td>The system should replay the input datasets when needed.</td>
<td>Should</td>
</tr>
<tr>
<td>FR-05</td>
<td>The system should allow push and pull incoming data and ingest it in the pre-processing platform.</td>
<td>Should</td>
</tr>
<tr>
<td>FR-06</td>
<td>The system should transform input datasets into stream tuples</td>
<td>Should</td>
</tr>
<tr>
<td>FR-07</td>
<td>The system should allow to persist the data during the transformation process if needed.</td>
<td>Should</td>
</tr>
<tr>
<td>FR-08</td>
<td>The system must publish translated data (i.e., stream tuples) to allow it can be used by external consumer applications.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-09</td>
<td>The system must serve translated data (i.e., stream tuples) directly to external consumer applications.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-10</td>
<td>The system must provide a time-based, sliding/tumbling window mechanism which allow to handle multiple aggregations over data streams.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-11</td>
<td>The system must provide a count-based, sliding/tumbling window mechanism which allow to handle multiple aggregations over data streams.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-12</td>
<td>The system must provide a time synchronization mechanism to join data streams based on timestamp.</td>
<td>Must</td>
</tr>
<tr>
<td>FR-13</td>
<td>The system should provide a mechanism to gracefully redeploy the system if necessary.</td>
<td>Should</td>
</tr>
</tbody>
</table>
4.3.2 Non-Functional Requirements

Table 4.2 depict the non-functional requirements of the system to be.

Table 4.2: Non-functional requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR-01</td>
<td>The system should have a latency of at most 500ms</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-02</td>
<td>The system must scale in the number of external partners providing sensor data.</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-03</td>
<td>The system must scale in the number sensors attached to a sensor network.</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-04</td>
<td>The system should allow integration with different messaging oriented middlewares.</td>
<td>Could</td>
</tr>
<tr>
<td>NFR-05</td>
<td>The system should allow integration with different database management systems, including Relational and NoSQL databases.</td>
<td>Could</td>
</tr>
<tr>
<td>NFR-06</td>
<td>The system should provide abstractions that facilitate the creation of pre-processing pipelines.</td>
<td>Should</td>
</tr>
<tr>
<td>NFR-07</td>
<td>The system must provide built-in parallel processing capabilities.</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-08</td>
<td>The system must provide built-in fault-tolerance mechanisms that allow automatic failover.</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-09</td>
<td>The system must provide guaranteed data processing, i.e., at-least-once semantics is essential.</td>
<td>Must</td>
</tr>
<tr>
<td>NFR-10</td>
<td>The system should provide exactly-once semantics for mission critical operations.</td>
<td>Should</td>
</tr>
<tr>
<td>NFR-11</td>
<td>The system must be easy to deploy, requiring a minimum of setup and configuration to get up and running.</td>
<td>Must</td>
</tr>
</tbody>
</table>

4.4 System Architecture

In this section we present the rationale of our proposed architecture, the logical view of the system, the process view of the system, and finally we describe the software design decisions for the system to be.
4.4.1 Rationale

From section 2.1.2 we noticed that a data integration solution can be build on components that implement the EIP proposed in [23]. In addition, we realized that an ESB is the current solution for integration (see section 2.1.3). It provides built-in functionality in terms of patterns and leave to developers only to program custom logic. We also observed that other researchers use different open source ESB solutions combined with CEP, SOA, or other technologies in order to tackle the challenges of sensor data integration (see section 2.1.4).

Although ESB solutions offer the aforementioned features, the current open source solutions do not provide built-in capabilities in terms of fault-tolerance, performance, scalability, parallel processing, and other important quality attributes. Hence, in order to obtain all the benefits provided by the different -ilities of software engineering, developers need to add all those features by themselves, or face with cumbersome configurations, which could together, may result in error prone applications. Otherwise, they need to pay for special subscriptions or acquire commercial versions of the software products [24,51].

Therefore, we propose that in order to provide a sensor data integration solution which also provides fault-tolerance, scalability, parallel processing and other quality attributes, we can design a solution based on the concepts on which ESBs are built, and exploit the capabilities of current SPFs.

We propose a solution that relies on Enterprise Integration Patterns, a Distributed Message-Oriented Middleware, and a state of the art Stream Processing System. EIPs are proved solutions for solving recurring problems. A MOM allows decoupling applications by means of asynchronous messaging exchange. In addition, current open source MOMs are inherently distributed, offer high scalability, and guaranteed delivery of messages which are desired features in our system. Moreover, current SPFs offer built-in fault-tolerance, parallel processing, guaranteed processing, declarative semantics that ease the creation of a pre-processing pipeline, and other capabilities.

4.4.2 Logical view

This view shows the structural elements to represent the system’s functionality. Figure 4.1 depicts the high-level architecture of the proposed solution.
4.4.2.1 System Decomposition

The main components of the system are the Raw Data Producer, the Input/Output Brokers, and the Adapter (see figure 4.1).

**Raw Data Producer.** This component is responsible for connecting the data sources with the pre-processing platform. It adapts the data sources in order to they can exchange data using a MOM.

This component comprises the Channel Adapter and Publisher components.

- **Channel Adapter.** This component is responsible for adjusting the data produced by the sensor network applications in order to the data can be exchanged using a messaging middleware (see section 2.1.2). For instance it adjusts sensor data stored in CSV files to allow it can transferred as a message.

- **Publisher.** The publisher receives the adapted data from the Channel Adapter and send it to the Input Broker.

**Input/Output Brokers.** These components are part of a message-oriented middleware responsible for temporary storage of messages and routing the messages produced at the data source to the right destinations. The *input broker* routes messages that contains the raw input datasets. The *output broker*, on the contrary, routes the transformed data, i.e., *stream tuples*, and serves that data as an output for external systems. Both, the *input* and *output* brokers provide a guaranteed ordered messaging delivery. This ensure that every time a message reaches one of the brokers and it is acknowledged, the brokers guarantee that those messages will not be lost. Additionally, the brokers guarantee that the consumers will receive the messages in the same order that they were produced.
**Adapter.** This component is responsible for translating raw input datasets $R$ into stream tuples $A^p_i$. In addition, this component will handle the windows mechanisms and the time synchronization described in chapter 3. This component can connect with persistent data storages through the IPersistent Storage Interface. It also can publish data to a message broker, or serve the data it produces directly through the IOutput Interface.

This component comprises the Polling Consumer, Message Translator, Aggregator, Time Synchronizer, and Publisher components. Figure 4.2 show the internal decomposition of the adapter component.

![Figure 4.2: Internal view of the adapter component](image)

- **Polling Consumer.** This component is the entry point of the adapter. It requests messages to the input broker and forwards the messages to the next components in the pre-processing pipeline. Notice that this component uses a pull strategy, i.e., it will not receive messages unless it explicitly request them. For this component we had to make a choice between push or pull based strategies for message consumption. We opted for pull because of the following reasons.

  1. A pull strategy allows the consumer to control the rate of consumption, which means it will not be overwhelmed in case its processing speed falls below the rate of production. This allow the consumer falls behind the producers rate and catch up when it can. This property is an automatic load shedding mechanism which is a desirable adaptive resource management technique as described in the subsection 2.2.1.
2. In case of failures the system should apply automatic failover mechanisms, which means that the Polling Consumer might be restarted in a new machine. Hence, it is easier for the consumer to find the output broker after a failure than the output broker keep track of which machine the consumer was restarted on.

- **Message Translator.** This component is responsible for transforming the incoming raw input datasets into stream tuples. This is a generic component, therefore different types of message translators can be implemented in order to provide the functionality of the patterns described in section 2.1.2.

- **Publisher.** This component receives the stream tuples from the message translator and send them to the output broker in order to they be consumed by external interested applications.

- **Aggregator.** This component provides the functionality to keep multiple aggregates of data, and the windows mechanisms described in the sections 3.2.2 and 3.2.3.

- **Time Synchronizer.** This component provides the functionality to align different streams based on timestamps as described in section 3.2.4.

Figure 4.3 depicts the complete view of our system architecture, it shows all the components described in this section.

### 4.4.3 Process View

In this section we present the activity diagrams that describe the most significant operations of the system: Data Transformations and Windowing Mechanism.

#### 4.4.3.1 Data transformations

The data transformation process is the starting point of the system. Before sensor data is analyzed it has to be pre-processed. In this stage multiple transformations take place until it finally is converted into stream tuples. Figure 4.4 depicts the sensor data transformation process from raw input datasets to stream tuples.

The process start at the Raw Data Producer component. First, it reads and validate a configuration file which has the initial settings that will be used for the transformation
process. If the validation of the configuration file fails, the system stops since the initial configuration allows the system to run properly. If the validation succeeds, it continues to the next step. Notice that this step is executed only once, when the system runs for the first time.

After the configuration validation succeeds, the input data file is read. Internally this action splits the input data file into lines which are passed to the next step. Then the raw input dataset is generated and published in the Input Broker.

While all these steps where being executed the adapter component, concurrently, started to request raw input datasets. This is achievable due to the MOM used to decouple data producers and data consumers. The Request Dataset action showed under the adapter component in figure 4.4, is being executed specifically by the polling consumer component described in the section 4.4.2. The polling consumer is continuously requesting new data. However, to avoid that the polling consumer end polling data in a tight loop if there is no new data arriving, the implementation should allow the consumer to block in a long poll waiting until new data arrives. When a new raw input dataset arrives it is routed to the specific task depending on the type of dataset. It could be a set, matrix or superset, as described in the section

Scalable integration and pre-processing of sensor data streams 48
3.1. Finally the raw sensor data is processed and a *stream tuple* is published in the *output broker* for further consumption by interested applications.

![Activity diagram of the data transformation process](image)

**Figure 4.4:** Activity diagram of the data transformation process

### 4.4.3.2 Windowing Mechanisms

The *Message Translator* component translates the *raw input datasets* into *stream tuples* and then publishes the translated data in the *output broker*. Additionally, the *Message Translator* can pass the *stream tuples* to the *aggregator* in order to allow keeping windows aggregates of data that may be served to the analysis platform. Figure 4.5 shows the process that creates aggregations of data using a windowing mechanism.

The process starts at the *aggregator component* which initializes the windows’ buffers and the variables used for the different computations. Then the component starts to accept incoming *stream tuples*. Each time a new tuple arrives, it checks the correlation ID to verify if there is already a window keeping track of that type of *stream tuples*. If a window for that *stream tuple* already exist, the component checks the windows...
size. If the size $\geq$ length, then stream tuple at the head of the window is removed. Then the new stream tuple is added to the window. If there is not window for that kind of stream tuples, a new window is created and then the stream tuple is added to the window. Finally, the completeness condition is checked. If the condition is satisfied, the aggregator returns the window as an output. Otherwise, it continues receiving the next incoming stream tuples.

Figure 4.5: Activity diagram of the window aggregation process
Chapter 5

Implementation

In chapter 3 we presented the problems that we try to solve with this work. In chapter 4 we described the software architecture of our proposed solution. In this chapter we describe the implementation of a prototype that materializes the architectural concepts and aims to solve the problems presented in chapter 3. We present the software design decisions, a rationale about why we opted for a specific tool to implement a component, and how the main components of the system were implemented.

5.1 Software design decisions

In this section we justify the decision made regarding the particular software that we used for the system to be. As we stated in section 4.4, our proposed solution relies in two types of software, a messaging oriented middleware and a stream processing framework.

5.1.1 Message-Oriented Middleware

The decisions for choosing a particular MOM solution was made based on the key drivers, the functional requirements FR-03, FR-04, and the non-functional requirements NFR-01, NFR-02, NFR-08, NFR-09, NFR-10. Table 5.1 depicts the design decisions for this software component.
Table 5.1: MOM Design Decisions

<table>
<thead>
<tr>
<th>Name</th>
<th>Message-Oriented Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Selection of the MOM for the implementation of the Input/Output Brokers components</td>
</tr>
<tr>
<td>Issues</td>
<td>The raw input datasets have to be ingested to the system and routed to the pre-processing platform in order to be transformed into stream tuples. In addition, once the data is translated, it has to be routed to the analysis platform and other interested client applications.</td>
</tr>
<tr>
<td>Decision</td>
<td>Apache Kafka</td>
</tr>
<tr>
<td>Status</td>
<td>Approved</td>
</tr>
<tr>
<td>Considered Options</td>
<td>JMS. The Java Message Service is a Java API that allows applications to implement messaging features, i.e., create, send and receive messages. It provides reliability and allows asynchronous messaging. [39]</td>
</tr>
<tr>
<td></td>
<td>ActiveMQ. It is an open sourced message broker implementation of JMS 1.1 as part of the J2EE 1.4 specification [48]. It provides high-availability, performance, scalability, reliability and security [43].</td>
</tr>
<tr>
<td></td>
<td>RabbitMQ. It is a message broker written in Erlang. It offers reliability, clustering, high availability, management UI, etc. [40]. RabbitMQ include persistence and it allows users to make trade offs between performance and reliability which involves choosing from different alternatives for publishers and delivery acknowledgements.</td>
</tr>
<tr>
<td></td>
<td>Apache Kafka. It is a real-time publish-subscribe system [21], intended to provide high-throughput, low-latency, support for partitioned, distributed, real-time processing, and guarantee fault-tolerance [16].</td>
</tr>
</tbody>
</table>

**Arguments for choosing Apache Kafka.** One of the reasons that made us opt for Apache Kafka was that the key drivers of its architecture were in line with the key drivers of our proposed system. It was designed to scale transparently with no downtime, and to operate fast (performance) [16].

In addition, Kafka’s design decisions also motivated us to use it as part of our proposed solution. Unlike other MOMs, Kafka’s mechanism to manage persistence allows it to offer durability and delivery guarantees, without sacrificing performance. Kafka
achieves this by using the filesystem to persist data and making use of the pagecache management provided by the OS. This technique allows Kafka to provide features that usually other MOMs do not offer. For instance, Kafka does not delete a message after it is consumed, but it keeps messages for a specified period of time, which can be set by configuration. Kafka, as most of the MOMs, use the Topic abstraction to group messages of the same type. However, those topics can be divided into multiple partitions, which are the unit of parallelism in Kafka. Hence, each partition must fit in the server where it is hosted, but a topic may have many partitions allowing to scale arbitrarily. Each partition is an ordered, immutable sequence of messages that is continually appended to a file in the system. Every time a message is appended, it is assigned a sequential id number that uniquely identifies each message within the partition. Such number is called offset. Kafka by design, leaves the consumers to keep record of the offsets they consume, which allows the consumer to reset the offset to an older value in order to reprocess the messages when needed. These features provided by Kafka partially contributes to meet the functional requirements FR-03 and FR-04.

Kafka provides replication to a configurable number of servers. This mechanism tolerates N-1 server failures with guaranteed no data loss. Kafka offers different guarantee deliveries. By default it offers at-least-once semantics, but it provides the mechanisms to achieve at-most-once or exactly-once semantics if required. This contributes to meet the non-functional requirements NFR-08, NFR-09, and NFR-10.

5.1.2 Stream Processing System

The decisions for choosing a particular SPF solution was made based on the key drivers, the non-functional requirements NFR-04, FR-05, NFR-06, NFR-07, NFR-08, NFR-09, NFR-10, NF-11. Table 5.2 depicts the design decisions for this software component.
Table 5.2: SPF Design Decisions

<table>
<thead>
<tr>
<th>Name</th>
<th>Stream Processing System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Selection of the SPF for the implementation of adapter component</td>
</tr>
</tbody>
</table>

**Issues**

We need a middleware that allows to process data on the fly with low latency. In addition, it should allow scale in and out, and thereby exploit the capabilities of a cluster of machines to process different raw input datasets in parallel.

**Decision**

Apache Storm

**Status**

Approved

**Alternatives**

Apache S4. It is a general purpose stream processing system inspired by the MapReduce model. Its design is a combination of MapReduce and the Actors Model, and it was driven by the need of large scale applications for data mining and machine learning [38]

Apache Sanza. It is a distributed stream processing system created at LinkedIn, and now open sourced as an Apache Incubator Project. Sanza offers managed state, fault tolerance, message durability, scalability and resources isolation [26].

Storm. It is defined as a “distributed realtime computation system” [41]. Its design was driven by the need of achieving near-real time response while processing high volumes of data [31].

**Arguments for choosing Storm.** One of the reasons for choosing Storm was that its capabilities are in line with the key drivers of our architecture. It offers scalability, guaranteed data processing, fault-tolerance, flexibility to integrate with other systems, and a simple API that allows easily to define processing pipelines without having to deal with complex xml structures as in the case of S4.

Regarding guaranteed processing semantics, Storm by default offers at-least-once semantics. However, trident, which is a high-level API implemented on top of Storm, allows to create stateful applications and provides exactly-once semantics. S4 only offers at-most-once semantics. In contrast to Storm, it uses a push based model and if the receiver’s buffer is full messages are dropped.

On the other hand, Storm provides built-in components that allows integration with different MOMs and databases. For instance it allow integration with Kestrel, RabbitMQ, Kafka, Cassandra, MongoDB, HBase, etc.
Unlike S4 and Samza, Storm is more mature and it has a larger number of adopters (e.g., Twitter, Yahoo, The Weather Channel, etc) [41]. Storm has a very active community of developers who provide useful insights for new storm users. The community nowadays has about four thousand members \(^1\). Likewise, Storm has a reach documentation which allow to understand its internal architecture [31]. S4 and Samza do not provide documentation and clear examples of how to create applications atop them.

5.2 Guaranteeing message processing

One of the challenges that we had to face during implementation is how to deal with failures. In this section we present the guarantees that Kafka and Storm offer and explain the trade offs between performance and reliability in order to create a robust application.

First we describe what such guarantees mean.

- **At most once**: Messages might be lost but they are never redelivered.
- **At least once**: Messages are never lost but they may be redelivered.
- **Exactly once**: Each message is delivered once and only once.

**Kafka Guarantees.** To analyze the guarantees provided by Kafka we have to see from the producer and consumer point of view. This is the guarantees when a message is being published and the guarantees when consuming a message.

From the producer point of view Kafka uses a notion of “commit” when publishing messages to the cluster. Kafka by default provides “at least once” semantics from the producer perspective. It allows the producer to specify the “request.required.acks” option when publishing a message. There are three options to set this parameter. Zero (0), it provides the lowest latency but the weakest guarantees. It means that the producer will not expect any acknowledgement after sending a message. One (1), it provides better guarantees because the producer will wait until the replica leader acknowledge the message. With this option only the messages that were written to the leader but not yet replicated will be lost. Minus one (-1), it means the producer will wait until it receives the acknowledgement of all in-sync replicas. This option

\(^1\)https://groups.google.com/forum/#!members/storm-user
provides the best durability but it adds some latency. Another important option to take into account is “message.send.max.retries”, which determine the number of retries that the producer will try in the presence of failures. It is important to notice that this option might cause message duplication when a network failure occurs after a message is sent but the acknowledgement is lost. Ideally, the producer should send each message with an associated key that makes the retry operation idempotent, with this feature the producer could retry until it receives the acknowledgement and be sure that the message was published exactly once. However, this option is not currently implemented.

From the consumer point of view, Kafka lets the consumer to control the position in the log from which to read messages. In this way, the decision of which semantics to implement lies on the consumer application (Storm in our case). Hence, depending on the order the consumer processes the messages and save its current offset, the application could have any of the three delivery guarantees.

- **At most once**: The consumer can read the message, save its position, and finally process the message. In this case if the consumer crashes after saving its position but before processing the message, the process that takes over processing will start from the saved position but the messages that were not processed will be lost.

- **At least once**: The consumer reads the message, process the message and then save its position. In this case if the consumer crashes after processing but before saving the offset, the taking over process will receive some message that were already processed by the now crashed consumer.

- **Exactly once**: This is not a feature of Kafka but rather a coordinated procedure with the storage system of the consumer’s output. This guarantee can be achieved by letting the consumer stores its offset in the same place as its output.

**Storm Guarantees.** Storm provides abstractions to achieve the three previous mentioned guarantees. First, by default, Storm provides “at most once” semantics because it will not replay failed tuples unless the spout implementation provides this capability and the bolts implement reliable mechanisms appropriately.

The core abstractions of Storm facilitate the implementation of “at least once” semantics. In order to achieve this guarantee, the spout has to implement the `ack` and `fail` methods to handle the successful and failed tuples respectively. In addition, a
link in the tuple tree has to be created by adding an “id” when emitting the output
tuple inside the execute method in a bolt. This is called anchoring. Finally, Storm
needs to know when a tuple is fully processed and this is achieved by acking the
input tuple. When a tuple is acknowledged, the ack method in the spout is called.
Likewise, when a tuple times-out Storm will call the fail method in the spout, which
implements the replay logic. Additionally, An output tuple can be anchored to more
than one input tuple. This is useful when doing streaming joins or aggregations. A
multi-anchored tuple failing to be processed will cause multiple tuples to be replayed
from the spouts. Multi-anchoring is done by specifying a list of tuples rather than
just a single tuple in the emit method.

To achieve “exactly once” semantics, Storm relies on Trident, an abstraction imple-
mented atop of it. Trident base its logic in the following assumptions in order to
provide exactly once semantics.

- Tuples are processed in small batches
- Each batch is given a unique id (txid) and every time a batch is replayed is
given the same id.
- State updates are ordered among batches. That is, updates for batch with txid
2 will not be applied until the update for the batch with txid 1 succeed.

Additionally Trident relies on the implementation of Transactional and Opaque Trans-
actional spouts and states. A Transactional Spout has the following properties.

- Batches for a given txid are always the same. Replays of batches for a txid will
replay the exact same set of tuples as the first time that batch was emitted for
that txid.
- Tuples are in one batch or another, never multiple, i.e., tuples do not overlap
different batches.
- Every tuple belongs to a batch.

A transactional state implementation that has exactly-once semantics for transac-
tional spouts relies on the fact that any given txid is always associated with the exact
same set of tuples. Hence, suppose that we need to compute word count and store
the counts in a key/value database. If we store only the counts there is no way to
know if that word was already counted, however a transactional state additionally
stores the txid together with the count. Then, when updating the count, we need to
compare the transaction id in the database with the transaction id for the current batch. If they are the same, the count is not updated. If they are different, the count can be incremented.

On the other hand, an opaque transactional spout has the following property.

- Every tuple is successfully processed in exactly one batch. However, it is possible for a tuple to fail to process in one batch and then succeed to process in a later batch.

With this kind of spout is not possible to skip the state update if the transaction id in the database is equal to the transaction id for the current batch because the batch may have changed between state updates. However, it is possible to store more state in the database. Hence, additionally to the transaction id, and the value, the previous value is also stored. Then it is possible to achieve exactly once semantics updating the state based on the previous and current values.

Without Trident is more challenging to achieve “exactly once” semantics. Storm offers what is called Transactional Topologies. However, this implementation is deprecated since it evolves to Trident. In order to achieve “exactly once” semantics with the core version of Storm, the incoming tuples should be processed in small batches, and the computation of a batch should be divided in two phases.

- The processing phase: this is the phase that can be done in parallel for many batches.

- The commit phase: The commit phase for batches are strongly ordered. So the commit for a batch with txid 2 is not done until the commit for batch with txid 1 has been successful.

The two phases together comprise a transaction. Many batches can be in the processing phase at a given moment, but only one batch can be in the commit phase. In the case of failures in the processing or commit phase for a batch, both phases should be replayed. If “exactly once” semantics want to be achieved with basic Storm, developers should manage the state, and coordinate the transactions. That is the transaction id and the metadata with the information for each batch, and the mechanism to determine which transactions should be processing or committing at certain time.

Due to time constrains the implementation of our prototype implements “at least once” semantics which is the default guarantee of Kafka and achievable using the core functionality of Storm.
5.3 Data transformations

In section 3.2.1 we described the need for a mechanism that allow transforming raw sensor data into a predefined schema to be used by the processing platform. In this section we describe the implementation details of the module that transform the raw input datasets $R$ into stream tuples $A_i^ϕ$.

The implementation of this functionality was expanded in two components. The first one deals with adjusting the raw input dataset to be transferred in the form of messages through Kafka. This component implements a channel adapter proposed in [23] and presented in chapter 2. In our prototype this component first reads the raw sensor data from the filesystem (csv files), serializes the data using Json, and publishes the data to a Kafka topic to be consumed by the Adapter component described in section 4.4.2.1.

Once the data is available for consumption the Adapter component consumes the raw input data from Kafka. The adapter component is implemented as a Storm topology that is where all the transformations take place. The polling consumer presented in our system architecture is implemented as a Storm’s spout. It is responsible for reading the data out of Kafka and pass it to the next component in the topology which implements the message translator pattern. There are different patterns implemented as subcomponent used to transform the raw input streams. For instance, a Time Normalizer is implemented to handle the timestamps of the sensor readings. In the case the timestamp is not present in the raw data, a Content Enricher deal with it and generates a timestamp. Finally a Splitter is used to break the original input which contains readings from different sensors into single measurement which became a stream tuple $A_i^ϕ$ in the form of $⟨t_i, k_i, V_i⟩$. Finally this component emit two output streams, one used to publish the transformed data into a new Kafka topic, and other to continue processing of windows aggregates. This capability of Storm of producing outputs to more than one stream allows to pass the results of a computation to other stages in the preprocessing pipeline.

5.4 Multiple Stream Aggregates

Another of the problems to solve is the need for keeping multiple aggregates of stream tuples as described in section 3.2.2. Storm itself does not provide this functionality,
however, the abstractions provided by storm allow us to implement it. The multiple stream aggregator was implemented as a Bolt within a Storm topology. In Object Oriented Programming, the Dependency Inversion Principle (DIP) says that high level modules should always depend on interfaces and not in actual implementations [29]. We follow this principle and our implementation of the multiple stream aggregate depends on three different interfaces, IWindow, IWindowFactory, and IAggregationFunction. Figure 5.1 depicts the class diagram of this part of the system.

IWindow exposes the functions of a Window mechanism, described in the next section. Such mechanism will keep aggregates of each different stream tuple that arrives to our aggregator component. One way to comply with the DIP is by means of Abstract Factories, another design principle used to allow that dependencies on concrete classes exist in one, and only one, place. Hence, the IWindowFactory provides expose such functionality. Finally the IAggregationFunction exposes a function that receives a list of objects as an input, perform some computation over the elements of the list and produces an the result of the computation as output. In our prototype we implemented two aggregation functions Sum, and Average.

The aggregator bolt uses a $HashMap(key, Window)$ in order to keep different aggregates for each stream tuple that arrives. On arrival of a new tuple to the aggregator bolt, it first checks whether there is window already for that stream tuple. If not, it creates an entry for that new key and then it calls the $insert$ method. Then it checks whether $isProcessingTime$, which is a function that returns a boolean if the window has to be processed. If that is the case the Bolt calls the method $getWindow$ and pass the list of values corresponding to the window at that time to the $aggregationFuction$ (e.g. Sum, Avg, etc.). Such function produce the result of the aggregation and emit a new tuple with the result.

Additionally we implemented a special version of Aggregator which instead of applying a specific function over a window of data, keeps aggregates of data in a sorted window. This is implemented as a Sorting Bolt component. We realized that when exploiting the parallelism of Storm, we lost the notion of order. Hence this component is placed before the Aggregator Bolt to ensure that the Stream Tuples will arrive in order to it.
5.5 Windowing Mechanisms

One way of keeping aggregates of data is using a window abstraction. Based on the definitions presented in section 3.2.3 we implemented two different types of windows, Sliding Window and Tumbling Window. However, depending on the policies to discard old tuples from the window and the policies that will trigger an output, there are multiple windows implementations.

This fact plus the aim to comply with the Dependency Inversion Principle, led us to opt for a window factory. A factory should be used when a system has to be configured with one of multiple families of products. In this case the eviction policies, Count-Based and Time-Based, originate four different types of Sliding Windows and two types of Tumbling Windows. Hence, we used the Abstract Factory pattern to implement the window functionality. Figure 5.1 depicts the classes used to implement this part of the system.

The IWindowFactory is an abstract factory that exposes the method `createWindow`. There are two concrete factories, one for Sliding Windows and other for Tumbling Windows. These factories implement the `createWindow` method and returns a concrete window implementation based on the parameters passed at construction. All the implementations of the Windows inherit from the IWindow interface. This interface exposes the functionality used by the Aggregator component described in the previous section.
Figure 5.1: Class diagram of the multiple aggregation using windows.
Chapter 6

Evaluation and Discussion

In chapter 5 we described the implementation of the prototype used as a proof of concept of our proposed solution. Such implementation would not have validity if we do not evaluate it. In this chapter we described different test scenarios that we used to evaluate our solution. We present some statistics and discuss the results of the evaluation.

6.1 Evaluation Tests

We evaluated our prototype in terms of performance and reliability. For the evaluation we used different scenarios which are described in the next sections. All the tests were run in a single machine with an Intel Core i7 processor and 8GB of ram. We used four different configuration of Storm topologies for our test. All of them were executed in local mode. Figure 6.1 depicts the different topologies used in the evaluation. Each topology first parses the raw datasets, then calculates a moving average in a Window Aggregator and finally emits an output with the results. the Topology A (6.1a) is the simplest configuration. It is a single threaded topology which does not use the parallelism provided by Storm. Topology B (6.1b) exploits the parallel processing capabilities provided by Storm. This topology was configured with a parallelism hint of 2,4,2 for the ParserBolt, AggregatorBolt, and OutputBolt respectively. This indicates the number of instances that will be created for each component at runtime. Topology C (6.1c), uses the same parallelism hint than the previous one, however, in this case the spout also run in parallel with a parallelism hint of 4. This is possible because the spout consumes tuples from a partitioned topic from Kafka. Finally, Topology D (6.1d) includes the SortingBolt with a parallelism hint of 4 in order to
ensure that the data arriving to the Aggregator arrives in the same order than it was produced.

The tests were performed using three different datasets. All were csv files where each record represent a sensor reading executed every 9 seconds. Each reading contains 24 sensor measurements. From now on we will call the datasets as 5k, 100k, and 500k, which contains 5 thousand, 100 thousand and 500 thousand records respectively. The 5k dataset represent about 1 day of data, the 100k, about 10 days and the 500k about 2 months. These datasets were used just to perform a stress test and see how well the system performs, however, they do not represent a realistic scenario. This system is intended to integrate sensor data in near-real time, hence a more realistic scenario is a smaller dataset which contains the freshest data. Therefore, a final performance test was executed using the last 15 minutes of data. In this dataset each record has 1000 sensor readings, which simulates a fiber optic cable with multiple sensors attached to it.

The base configuration for Kafka was a single partition with a replication factor of 2 and required acks -1, the variants are explained within each particular test. For details about these parameters refer to section 5.2.
6.2 Test 1. Single thread vs. Parallel topology

This test was executed using the topologies A and B, and the 5k and 100k datasets. The processing time was calculated since the producer started to send data and until the last tuple was processed by the OutputBolt. Figure 6.2 depicts the results of this test. We observed that in both cases the execution in parallel outperformed the single threaded one. There was a gain in performance in the case of 37.5% for the 5k dataset, and a 14.6% for the 100k.
6.3 Test 2. Non partitioned vs. Partitioned topic

This test was executed using the topologies A, B and C, and the 500k dataset. The processing time was calculated since the producer started to send data and until the last tuple was processed by the OutputBolt. Figure 6.3 depicts the results of this test. The first part of this test was exactly like the previous one, observing that the parallel configuration outperformed the single threaded one by 19.76%.

Since we noticed that the time for processing the entire dataset substantially increased, we decide to repeat the test but this time with a partitioned Kafka topic. Instead of producing and consuming tuples from a single Kafka partition we used a topic with 4 partitions. For this attempt we used the topology C. We expected to gain in performance in this case since the spout will also run with parallelism hint of 4. However, the results showed that the time for processing the entire dataset increased with respect to the non partitioned version. Nonetheless, the performance was still better than the one produced with the topology A (single thread). We attributed that the small lose in performance regarding the non partitioned test to the fact that we were using strong guarantees for the raw data producer (required acks -1) and with a replication factor of 2. Hence, every time a message was produced it had to be replicated and this for the 4 partitions of the topic. Then, we decided to do the
last test but this time avoiding the producer wait for acknowledgements from the Kafka brokers (required acks 0). The results were not considerably better than those produced with the Parallel Configuration A. With this configuration the performance increased a 15.43%. However the performance gained with this configuration over the topology A (Single thread) was 32.1%, the processing time decreased from 13.93 to 11.78 minutes.

Figure 6.3: Performance test with the 500k dataset and different Kafka partitions

6.4 Test 3. Only Storm and different parallelism configurations

So far we calculated the time of processing the whole dataset since the first message was produced until the OutputBolt produces the last output. Since we observed that the difference in performance when using a non partitioned vs. a partitioned Kafka topic, we decided to test the performance of Storm only. For that we used the topologies A and B again, but this time instead of producing new messages, we start reading the messages that were already in the Kafka brokers. Figure 6.4 depicts the results of this test. Additionally in this test we decided to see if different configuration of the parallelism hint produces a remarkable difference. What we noticed is that the
results were much satisfactory with regard to those produced in the Test 1, when the test started from publishing the messages to Kafka. The time needed to process the entire 5k dataset decreased from about 28.8 seconds to 4.63 seconds in the case of a single threaded topology. Likewise for the case of the topology B, it decreases from 18 to about 4.77 seconds. In addition, for this test we duplicated the parallelism hint from 2-4-2, used in Test 1 to 4-8-4 instead. However, we obtained similar results in both cases, 4.63 for the 2-4-2 configuration against 4.77 for the configuration 4-8-4.

In addition, we executed similar performance tests using the 100k and 500k. The time needed to process the entire datasets reduces considerably, with regard to the Test 1 and Test 2, when measuring only the performance of Storm.

Figure 6.5 shows the result of the test using the 100k dataset. For this test we made three trials and then we averaged the results. For the same dataset with a single thread topology the result was 4.03 minutes (see figure 6.2), with this test we obtained a result of 48.7 seconds. There was a gain in performance of 79.85%. For the topology B with parallel configuration, the increase in performance was 81.53%.

Figure 6.6 shows the results of the test using the 500k dataset. For this test we made three trials as well and the we averaged the results to give a final assessment.
Figure 6.5: Performance test with the 100k dataset - only Storm

In this case the parallel topology outperforms the single-threaded one by 19.61%. The single-threaded topology processed the entire dataset in 3.57 minutes and the one with parallelism in 2.87 minutes.
Figure 6.6: Performance test with the 500k dataset - only Storm

6.5 Test 4. Complete test using the last 15 minutes of data

As we stated before, the performance test executed so far were intended to check how well the system performs with big datasets. However, this situation does not represent a real scenario where sensor data is being produced, pre-processed and analyzed. The datasets 5k, 100k and 500k used before represent historical data, or let’s say data that was being stored for further processing while the processing platform was down. Even in that case, the 5k dataset is more realistic, it contains about 1 day of data which was being produced every 9 seconds. That means that the pre-processing platform has to catch up after one day of crash. The other datasets represent 10 days and 2 months of data, which is simply not acceptable. The system should not be down that amount of time in any circumstances.

This dataset contains the freshest 15 minutes of data, which is more realistic because normally a sensor network keeps the data on a server for a couple of minutes and then transfer it for further analysis. For this test, the processing time was calculated since the first record was being produced until the last output was published at the end point. Once again, the topology using the parallel processing capabilities of Storm
outperformed the single threaded one. However, this time the performance gained was 79.5%. The single thread topology processed the 15 minutes of data in 46.84 seconds while the parallel one processed it in 9.60 seconds. Figure 6.7 shows the results of this test.

![Figure 6.7: Performance test using 15 minutes of data](image)

### 6.6 Test 5. Reliability tests

Finally we performed a reliability test to evaluate the results produced by the Window Aggregation algorithms. First we ran the topology A, configuring the AggregatorBolt to use a Sliding Window. The sliding window was configured to keep windows of 1 minute and produce the average every 30 seconds. This algorithm was executed over the data of the 15 minutes dataset. Figure 6.8 shows the outputs produced by the algorithm. In order to check if the output produced by the algorithm was correct, we pre-calculate the output and we compared against it. In this case both output datasets matched.

We ran the same algorithm but this time using the topology B. We realize then, that the output produced by the topology B was not producing the same output than the one produced using a single thread topology. We realize that it was due
Figure 6.8: Average of a 1 minute window calculated every 30 seconds

to the parallelism introduced in this topology. The fact that we were parsing and aggregating different stream tuples in different instances of the components, caused that the ordering of the inputs was altered. Figure 6.9 shows the mismatch between the results produced by the topologies A and B. This shows that if not analyzed correctly, the parallelism might introduce inaccuracy.

Therefore, we introduced the SortingBolt, which is described in section 5.4. The topology D includes this component between the ParserBolt and the AggregatorBolt, ensuring that the data arrives to the AggregatorBolt in the same order that it was produced. Figure 6.10 shows the results of the topology D compared against those produced by the topology A. As can be seen in the figure the two lines perfectly match.
Figure 6.9: Average of a 1 minute window calculated every 30 seconds running in parallel without preserving order

Figure 6.10: Average of a 1 minute window calculated every 30 seconds running in parallel and order preserved
In this thesis we try to answer the question *Can stream processing techniques be applied for integrating and pre-processing data from heterogeneous sensor networks?*. We presented a thorough literature research which help to understand the challenges of data integration, the capabilities that a stream processing system should have, and which are the contributions made by other researchers in this field.

Based on the findings in the literature we propose that in order to provide a sensor data integration solution which also facilitates performance, scalability, fault-tolerance and other quality attributes, we have to design a solution based on the concepts on which ESBs are built, and exploit the capabilities offered by current Stream Processing Systems.

Our solution relies on Enterprise Integration Patterns (EIPs), a Distributed Message-Oriented Middleware (MOM), and a state of the art Stream Processing Systems (SPS). We based our proposal on different facts. EIPs are proved solutions for solving recurring problems. A MOM allows decoupling applications by means of asynchronous messaging exchange. In addition, current open source MOMs are inherently distributed, offer high scalability, and guaranteed delivery of messages. On the other hand current SPSs offer built-in fault-tolerance, parallel processing, guaranteed processing, and declarative semantics.

We created a prototype of our solution using the Apache Kafka message oriented middleware and Storm stream processing system. We implemented some features relevant for data integration (i.e., enterprise integration patterns) and others necessary to handle sensor data specifics (e.g. windowing mechanisms). Finally we evaluated our solution in terms of performance, scalability and reliability. Kafka and Storm
allow to scale horizontally and vertically, so both are suitable system for creating an scalable application. The performance tests presented in chapter 6 shows that our solution, using the parallel processing capabilities of Storm, outperformed by 75% to a traditional single threaded application. In addition, the reliability tests shows that using an additional component for keeping the order of the input data, the output of the system is correct and equal to the output of a traditional system where only one process is handling the whole load.

Therefore, the research question is answered. Stream processing techniques can be used for sensor data integration and pre-processing. However some adaptations need to be done in order to tackle the challenges of data integration and sensor data specifics.
Chapter 8

Future Work

In this research one of the major limitations was the time. Hence, there are some topics that may be studied with more detail. In addition, there are other topics that may be covered which arise from the challenges encountered during the execution of this work.

One important direction for this research might be to study how to restart the execution of the system ensuring that the current processing will finish smoothly. Storm provides internal command to stop a working topology specifying the time the system will wait until the topology is actually stopped. This time is intended to let storm to finish its current work. One main limitation is how to be sure that such time that you pass as parameter is the right time needed to finish processing the data already in the system?. Due to time constrains we could not investigate in that direction, so it might be an interesting work for future research.

Another important direction for future research might be how to guarantee that the order of the input data is maintained given the fact of network delays and parallel processing. We found that dealing with the order of the incoming data is not straightforward. As we explained in chapter 5 we introduced a component to sort the input data and ensure that the data arrives in increasing order to the aggregation algorithm. The sorting component keep windows of data which is sorted upon arrival. However we did not made a detailed investigation about the size of the window required to order the data. We noticed that with windows of arbitrary size the result were not the same than those produced by a single thread version of the application. Hence, the size of the window is important to ensure the correctness of the results.
Finally, another important topic to cover might be how to integrate a system like the one proposed in this work to a lambda architecture [30]. A lambda architecture is a concept depicted by the creator of Storm, Nathan Marz. It is intended to integrate batch processing with real time systems and provide more accurate results. We think that creating a platform for historical and current sensor data analysis might be an interesting work for further research.
Appendix A

Input Datasets Examples

A.1 SAARecorder Version 4.86

In this example we can see a file which contains metadata and timestamp in the file’s header (lines 1–13), column headers (line 14), and the data structure is a Matrix dataset (lines 15–25).

<table>
<thead>
<tr>
<th>Listings A.1: SAARecorder Sample File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SAARecorder Version 4.86</td>
</tr>
<tr>
<td>2 Array Name: SAAF_ABC_XYZ</td>
</tr>
<tr>
<td>3 Reference Filename: None</td>
</tr>
<tr>
<td>4 XYZ data generated from live data.</td>
</tr>
<tr>
<td>5 Current Date (YYYY-MM-DD): 2013-03-13</td>
</tr>
<tr>
<td>6 Current Time (HH:MM:SS): 14:18:00</td>
</tr>
<tr>
<td>7 Number of Vertices: 11</td>
</tr>
<tr>
<td>8 Output data is expressed at each vertex (or physical joint of the array), starting with vertex 0.</td>
</tr>
<tr>
<td>9 Reference = FAR</td>
</tr>
<tr>
<td>10 Array Azimuth Angle = 0.00 deg</td>
</tr>
<tr>
<td>11 Location Name: V5</td>
</tr>
<tr>
<td>12 Software Averaging set to 4 samples</td>
</tr>
<tr>
<td>13 302 raw data samples available for averaging.</td>
</tr>
<tr>
<td>14 Vertex #, X(mm), Y(mm), Z(mm), AccX(g), AccY(g), AccZ(g), Rx(deg), Ry(deg), Temperature(degC)</td>
</tr>
<tr>
<td>15 0, 3000.00, 950.00, 5.00, 0.0105, -0.0049, -0.9981, -0.28, -0.60, 18.50</td>
</tr>
<tr>
<td>16 1, 2998.42, 950.74, 154.99, -0.0520, -0.0015, -1.0051, -1.73, 3.58, N.A.</td>
</tr>
<tr>
<td>17 2, 3006.22, 955.99, 304.70, -0.0259, -0.0015, -1.0017, 1.92, -1.49, N.A.</td>
</tr>
<tr>
<td>18 3, 3010.10, 956.21, 454.64, -0.0281, 0.0083, -1.0017, 1.92, -1.49, N.A.</td>
</tr>
<tr>
<td>19 4, 3014.31, 954.96, 604.58, -0.0301, -0.0066, -1.0051, 0.56, 0.12, N.A.</td>
</tr>
<tr>
<td>20 5, 3018.82, 955.95, 754.51, -0.0179, -0.0045, -1.0006, 0.12, -0.70, N.A.</td>
</tr>
<tr>
<td>21 6, 3021.51, 956.62, 904.48, 0.0054, -0.0174, -1.0000, -0.74, -1.33, 18.50</td>
</tr>
<tr>
<td>22 7, 3020.71, 959.23, 1054.46, 0.0170, -0.0094, -1.0092, 0.43, -0.67, N.A.</td>
</tr>
<tr>
<td>23 8, 3018.15, 960.64, 1204.43, 0.0769, 0.0304, -0.9990, 2.28, -3.43, N.A.</td>
</tr>
<tr>
<td>24 9, 3006.62, 956.08, 1353.92, 0.1428, 0.0910, -1.0018, 3.48, -3.78, N.A.</td>
</tr>
</tbody>
</table>
A.2 Cow Events File

In this example we can see a file with column headers (line 1) and the data is structured as a set.

Listings A.2: Cow Events Sample File

<table>
<thead>
<tr>
<th>Cow</th>
<th>LifeNumber</th>
<th>Event</th>
<th>Date</th>
<th>Number</th>
<th>Remark1</th>
<th>Remark2</th>
<th>Remark3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>HeatDetected</td>
<td>06-04-2013 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>InHeatAttention</td>
<td>NaN,NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>HeatDetected</td>
<td>13-02-2013 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>InHeatAttention</td>
<td>NaN,NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>DryOff</td>
<td>14-06-2013 00:00:00</td>
<td>10</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>Disease</td>
<td>14-06-2013 08:01:00</td>
<td>NaN</td>
<td>Preventief</td>
<td>Overigen</td>
<td>NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>PregnancyCheck</td>
<td>27-12-2012 00:00:00</td>
<td>NaN</td>
<td>Pregnant</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>PregnancyCheck</td>
<td>10-01-2013 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3466</td>
<td>NL 1111</td>
<td>PregnancyCheck</td>
<td>16-01-2013 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>57</td>
<td>NL 2222</td>
<td>HeatDetected</td>
<td>21-02-2013 00:00:00</td>
<td>NaN</td>
<td>InHeatAttention</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>57</td>
<td>NL 2222</td>
<td>Transfer</td>
<td>28-02-2013 12:03:45</td>
<td>NaN</td>
<td>Departure</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>522</td>
<td>NL 3333</td>
<td>MilkSeparation</td>
<td>03-05-2013 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>522</td>
<td>NL 3333</td>
<td>Calving</td>
<td>03-05-2013 23:49:16</td>
<td>2</td>
<td>Colostrum</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>522</td>
<td>NL 3333</td>
<td>HeatDetected</td>
<td>03-12-2012 00:00:00</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

A.3 Super sets

In this example we can see a file without column headers, the timestamp is in unix epoch format and it is included in every line, and the data is structured as a superset. The numbers at the columns 2–5 represent the number of elements of each subset.

Listings A.3: Superset Sample File

<table>
<thead>
<tr>
<th>1356027305.4797</th>
<th>10 4 4 6</th>
<th>1519.8058</th>
<th>1524.8834</th>
<th>1529.7096</th>
<th>1534.9249</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2094</td>
<td>1544.7793</td>
<td>1549.9689</td>
<td>1554.7560</td>
<td>1559.7152</td>
<td>1565.4524</td>
</tr>
<tr>
<td>1540.0070</td>
<td>1549.0894</td>
<td>1560.0162</td>
<td>1530.2763</td>
<td>1540.3879</td>
<td>1550.0993</td>
</tr>
<tr>
<td>1559.7679</td>
<td>1520.4464</td>
<td>1530.3482</td>
<td>1540.2388</td>
<td>1549.9614</td>
<td>1559.9347</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1356027305.4807</th>
<th>10 4 4 6</th>
<th>1519.8068</th>
<th>1524.8834</th>
<th>1529.7099</th>
<th>1534.9234</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2086</td>
<td>1544.7787</td>
<td>1549.9701</td>
<td>1554.7582</td>
<td>1559.7152</td>
<td>1565.4506</td>
</tr>
<tr>
<td>1540.0057</td>
<td>1549.9007</td>
<td>1560.0170</td>
<td>1530.2762</td>
<td>1540.3875</td>
<td>1550.1012</td>
</tr>
<tr>
<td>1559.7698</td>
<td>1520.4451</td>
<td>1530.3481</td>
<td>1540.2381</td>
<td>1549.9626</td>
<td>1559.9359</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1356027305.4817</th>
<th>10 4 4 6</th>
<th>1519.8050</th>
<th>1524.8834</th>
<th>1529.7078</th>
<th>1534.9249</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2078</td>
<td>1544.7787</td>
<td>1549.9689</td>
<td>1554.7560</td>
<td>1559.7153</td>
<td>1565.4524</td>
</tr>
<tr>
<td>1540.0045</td>
<td>1549.9912</td>
<td>1560.0157</td>
<td>1530.2763</td>
<td>1540.3890</td>
<td>1550.0993</td>
</tr>
<tr>
<td>1559.7679</td>
<td>1520.4464</td>
<td>1530.3500</td>
<td>1540.2374</td>
<td>1549.9633</td>
<td>1559.9324</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1356027305.4827</th>
<th>10 4 4 6</th>
<th>1519.8076</th>
<th>1524.8834</th>
<th>1529.7096</th>
<th>1534.9229</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2094</td>
<td>1544.7775</td>
<td>1549.9707</td>
<td>1554.7574</td>
<td>1559.7135</td>
<td>1565.4524</td>
</tr>
<tr>
<td>1540.0070</td>
<td>1549.9912</td>
<td>1560.0162</td>
<td>1530.2763</td>
<td>1540.3879</td>
<td>1550.1012</td>
</tr>
<tr>
<td>1559.7679</td>
<td>1520.4477</td>
<td>1530.3482</td>
<td>1540.2388</td>
<td>1549.9633</td>
<td>1559.9328</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1356027305.4837</th>
<th>10 4 4 6</th>
<th>1519.8058</th>
<th>1524.8834</th>
<th>1529.7081</th>
<th>1534.9247</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2094</td>
<td>1544.7793</td>
<td>1549.9689</td>
<td>1554.7564</td>
<td>1559.7152</td>
<td>1565.4523</td>
</tr>
<tr>
<td>1540.0051</td>
<td>1549.9894</td>
<td>1560.0170</td>
<td>1530.2744</td>
<td>1540.3879</td>
<td>1550.0993</td>
</tr>
<tr>
<td>1559.7679</td>
<td>1520.4464</td>
<td>1530.3482</td>
<td>1540.2370</td>
<td>1549.9614</td>
<td>1559.9340</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1356027305.4847</th>
<th>10 4 4 6</th>
<th>1519.8068</th>
<th>1524.8834</th>
<th>1529.7078</th>
<th>1534.9231</th>
</tr>
</thead>
<tbody>
<tr>
<td>1540.2086</td>
<td>1544.7781</td>
<td>1549.9685</td>
<td>1554.7578</td>
<td>1559.7169</td>
<td>1565.4523</td>
</tr>
<tr>
<td>1540.0057</td>
<td>1549.9910</td>
<td>1560.0165</td>
<td>1530.2745</td>
<td>1540.3875</td>
<td>1550.0993</td>
</tr>
<tr>
<td>1559.7698</td>
<td>1520.4441</td>
<td>1530.3483</td>
<td>1540.2381</td>
<td>1549.9629</td>
<td>1559.9336</td>
</tr>
<tr>
<td>1356027305.4857</td>
<td>10 4 4 6</td>
<td>1519.8060</td>
<td>1524.8851</td>
<td>1529.7078</td>
<td>1534.9234</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>1540.2075</td>
<td>1544.7787</td>
<td>1549.9689</td>
<td>1554.7574</td>
<td>1559.7153</td>
<td>1565.4542</td>
</tr>
</tbody>
</table>
Scalable integration and pre-processing of sensor data streams
Appendix B

Data Transformations Required at different levels

Table B.1 depicts the type of transformations needed at different levels. This table was proposed in [23]. We adapted it to show some of the transformations needed in the sensor data analysis field and particularly at the TNO Use Case.
<table>
<thead>
<tr>
<th>Level</th>
<th>Deals with</th>
<th>Transformation Needs (Examples)</th>
<th>Tools/Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structures, (Application Layer)</td>
<td>Entities, associations, cardinality</td>
<td>Condense many-to-many relationship into aggregation. Structural Mapping, Patterns, Custom Code</td>
<td></td>
</tr>
<tr>
<td>Data Types</td>
<td>Field names, data types, value domains, constraints, code values</td>
<td><strong>TNO:</strong> Change timestamp from YYYY-MM-dd to UTC format, concatenate date and time to create the new field timestamp</td>
<td>EAI visual transformation editors, XSL, Custom Code</td>
</tr>
<tr>
<td>Data Representation</td>
<td>Data Formats, e.g., XML, name-value pairs (ID,Timestamp,Value), CVS, JSON, etc.</td>
<td>Parse data representation and render in a the new format. Serialization/Deserialization as necessary. <strong>TNO:</strong> CVS and JSON files to tuple (ID,Timestamp,Value)</td>
<td>XML Parsers, EAI parsers, Custom Code</td>
</tr>
<tr>
<td>Transport</td>
<td>Communications Protocols: TCP/IP sockets, http, SOAP, JMS, FTP</td>
<td>Move data across protocols without affecting message content. <strong>TNO:</strong> FTP to JMS, HTTP to JMS, etc.</td>
<td>Channel Adapters, EAI Adapters</td>
</tr>
</tbody>
</table>
References


[14] Eduardo B. Fernandez, Nobukazu Yoshioka, and Hironori Washizaki. Two patterns for distributed systems: Enterprise Service Bus - (ESB) and


[41] Storm Project. **Storm - Distributed and fault-tolerant real time computation.** http://storm-project.net/.


