SuNDroPS: 
SEMANTIC AND DYNAMIC DATA IN A PERVERSIVE SYSTEM

Context-ADDICT Revisited

Doctorsal Dissertation of:
Emanuele Panigati

Advisor:
Prof. Letizia Tanca

Co. Advisor:
Prof. Fabio A. Schreiber, Gianpaolo Cugola

Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Fiorini Carlo Ettore

2014 – XXVII
To My Family:
Roberto, Daniela and Mauro
“The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter – for the future. His duty is to lay the foundation for those who are to come, and point the way. He lives and labors and hopes.”

NIKOLA TESLA, Radio Power Will Revolutionize the World” in Modern Mechanics and Inventions, July 1934

“The progressive development of man is vitally dependent on invention. It is the most important product of his creative brain. Its ultimate purpose is the complete mastery of mind over the material world, the harnessing of the forces of nature to human needs. This is the difficult task of the inventor who is often misunderstood and unrewarded. But he finds ample compensation in the pleasing exercises of his powers and in the knowledge of being one of that exceptionally privileged class without whom the race would have long ago perished in the bitter struggle against pitiless elements. Speaking for myself, I have already had more than my full measure of this exquisite enjoyment; so much, that for many years my life was little short of continuous rapture”

NIKOLA TESLA, My Inventions (Chapter 1, “Early Life”), 1919
Acknowledgements

First of all I want to thank my family (my father Roberto, my mother Daniela and my brother Mauro) for their enthusiastic support during these three years of further studies.

Then I must thank all the professors who supported me in my academic experience, starting from my advisor, Prof. Letizia Tanca, always giving me the right advices and supporting me also during the most difficult moments of my journey. Prof. Fabio A. Schreiber and Prof. Gianpaolo Cugola, my coadvisors, helped me a lot in finding two very interesting research topics and in achieving useful results. Prof. Angelo Morzenti supported and still supports me with his experience and his grants, allowing me to reach the end of the Ph. D. and, together with Prof. Dino Mandrioli and Eng. Matteo Rossi, to work in some research projects that enhanced my understanding of how to work in this complex context.

I hope I have - at least partially - repaid them with my work in their team.

Eng. Giorgio Orsi, my master thesis co-advisor, deserves special thanks, since he made me understand how beautiful it is to be a researcher and thus transferred to me his passion for this amazing world.

I should also acknowledge all my colleagues, starting from my office-mates Emanuele Rabosio and Angelo Rauseo, always helpful with useful advice about my research and the Ph. D. life in general. I wish to thank also my other friends and colleagues Matteo Carminati, Luca Cassano, Mirjana Mazuran, Francesco Merlo, Antonio Miele and Paolo Ravanelli: working with friends makes things even easier than they really are.
A friend and colleague who deserves a special mention is Paolo Garza, who disclosed to me the foundations and some advanced aspects of *Data Mining*, and to all the other Ph. D. Candidates and research assistants working in the same research projects I am involved in (Andrea Bianchessi, Carlo Ongini, Daniele Rogora and Edoardo Vannutelli Depoli).

Thanks to Romolo Camplani whose coordinator role on the PerLa project I have inherited, for passing me also some of his experience.

Thanks to Alessandro Margara, a very valuable researcher, that also gave me a very good research opportunity I think I wasted because of some unfortunate circumstances. I hope he will achieve his career objectives, and also have the possibility to collaborate with him whenever possible.

Also some of my students need a special mention. Firstly I must acknowledge Angela Geronazzo for her great work during her thesis activity related to *MR-EClalT*: together we performed a big effort in designing the algorithm that she implemented, allowing me to concentrate only on the most theoretical aspects of this job. Matteo Rovero also produced a very good survey related to COP, which became the basis of the PerLaCOP design. Francesco Filippazzi and Guido Rota did a very great job with PerLa reengineering: it has been a great pleasure to work together with such valuable students and coders, always trying to point out pros-and-cons of each solution proposed and examined, without taking anything for granted. Also thanks to all the other students who worked with me: Stefania Crotti, Federico Monterisi, Riccardo Pincioli, Matteo Simoni.

Finally a big thanks goes to all my friends who supported me during this adventure: a particular mention goes to the guys of the technical staff of Cineteatro San Luigi, which I have managed till 2 years ago, trying to transmit them my passion for technical stuff.

Last, but not least, I have to thank Stefano Grassi (a friend and a Politecnico di Milano design student) for the design and realization of the front cover of this thesis work.

To the, and to all those I forgot to mention, I commit my final and kind "Thank you!"

Emanuele
Abstract

This thesis aims at developing a full-fledged system supporting pervasive information distribution and sharing. The key feature of this system is to allow to manage, integrate, analyze, and possibly reason on, a large and heterogeneous set of data. Part of this data set (e.g., user data, service data, network and vehicles static data) have to be stored in a traditional information system (e.g. based on a relational data structure) while another big part of them – those data that come from sensors, system logs and every other run-time, dynamic data – need to be treated as a data stream, which requires research beyond the current state-of-the-art. The system includes the use of context-aware techniques applied to data gathering, shared services, and information distribution; we also discuss how a context-aware approach applied to these tasks leads to the reduction of (noisy) information delivered to the users and to the personalized distribution of information. A privacy-safe approach to information distribution and advertising is adopted.

Within the overall architecture, the most relevant concern of this thesis is understanding how to manage the flow of information contained in the streams and how to extract useful knowledge from it. Due to the intrinsic dynamicity of the system, a large part of this analysis task has to be executed in real-time, otherwise its results might become outdated, and reasoning on them could lead to wrong conclusions. On the other hand, also historical data analysis is important since it makes it possible to understand some aspects, like performance degradation in time, that real-time analysis is not able to evince. The research activity will focus on methods
and techniques which allow to query this data stream in real-time, while at the same time preprocessing and loading data into a persistent data store for further off-line mining. Due to the large amount and to the simple structure of sensor-collected data, storage solutions like Cloud Databases and clusters of Parallel, Shared-Nothing, Distributed Database systems become very interesting.

The project, grounded in many results on the use of context-awareness and sensor-network data management already published by the same research group, aims at building a real-life system based on them. Prototypes of (parts of) the system are produced, within the application domains of car-sharing services and support for citizens’ mobility.
La tesi ha come scopo la progettazione e lo sviluppo di un sistema completo per il supporto della condivisione e distribuzione dei dati in sistemi pervasive. Tale sistema deve consentire la gestione, l’integrazione, l’analisi ed eventualmente l’inferenza basata su tecniche complesse come il “reasoning” automatico, su grandi insiemi di dati eterogenei. Parte di questi dati (ad es. i dati degli utenti, dei servizi, delle reti, …) è più opportuno che vengano immagazzinati in sistemi informativi tradizionali (cioè su sistemi basati su basi di dati relazionali) mentre un’altra grande parte di questi dati, costituita dai dati che provengono dai sensori, dai log di sistema e da tutti i dati dinamici registrati durante l’esecuzione di un sistema, può invece essere trattata come flussi di dati (“data stream”). Ciò richiede un’attività di ricerca che parte da, e va oltre, lo stato dell’arte. Il sistema include anche l’uso di tecniche di contestualizzazione da applicare alla raccolta dei dati, ai servizi condivisi e alla distribuzione delle informazioni. Si discuterà anche di come l’uso di un approccio contestuale in queste operazioni consenta di ridurre il numero di informazioni da distribuire agli utenti, personalizzando il processo di distribuzione stesso. L’approccio proposto consentirà inoltre un maggiore rispetto della privacy dell’utente, anche nelle attività di distribuzione personalizzata di informazioni commerciali e non.

Considerando l’architettura globale, questa tesi si focalizza su come i vari flussi di dati contenenti informazioni vadano gestiti in modo tale da consentire di estrarre informazioni utili. Data l’intrinseca ed elevata dinamicità del sistema, una grande parte di questi dati deve essere trattata in
tempo reale, altrimenti le informazioni risulteranno troppo “vecchie” e le analisi porteranno a conclusioni errate. D’altra parte, è necessaria anche un’analisi dei dati storici registrati dal sistema, per ottenere informazioni utili alla gestione del sistema stesso, quali ad esempio degli indicatori sul peggioramento delle prestazioni.

L’attività di ricerca si focalizzerà su tecniche e metodi che consentano di gestire i flussi di dati in tempo reale, mentre allo stesso tempo questi dati verranno preprocessati e memorizzati in una struttura persistente per altre analisi future. Data la grande mole e la struttura semplice dei dati provenienti dai sensori, soluzioni di memorizzazione alternative come i “cloud database” o i cluster di database paralleli e distribuiti devono essere tenute in considerazione.

La tesi, che ha le sue radici nei molti lavori e risultati pubblicati dello stesso gruppo di ricerca in merito a contestualizzazione e reti di sensori, ha come scopo quello di creare un sistema reale basato sulle tecniche che verranno descritte. Sono inoltre stati prodotti dei prototipi relativi ad alcune componenti del sistema, nel dominio applicativo dei servizi di car-sharing e del supporto alla mobilità dei cittadini.
1 Introduction .................................................. 1
   1.1 The SuNDroPS system: Context-ADDICT Revisited .... 3
   1.2 Application Case Study: The Green Move Project ....... 4

2 State of the Art ............................................ 7
   2.1 Context-Awareness ...................................... 8
       2.1.1 Communication .................................. 9
       2.1.2 Situation awareness and behavioral variations ... 10
       2.1.3 Context-Oriented Programming .................. 10
       2.1.4 Managing knowledge chunks ..................... 13
   2.2 Information Flow Processing .......................... 14
       2.2.1 Data Stream Management Systems and Languages .. 15
       2.2.2 Complex Event Processing Systems ................. 24
   2.3 Map-Reduce and NoSQL Databases ..................... 27
       2.3.1 The MapReduce Programming Model ............... 27
       2.3.2 Hadoop and Map-Reduce .......................... 27
       2.3.3 NoSQL Databases .................................. 28
   2.4 Data Mining and Frequent Itemset Extraction .......... 29
       2.4.1 Implementations of FIE in Distributed Environment .. 30
       2.4.2 Map-Reduce Implementations of FIE ............... 32

3 System Architecture Overview .......................... 35
   3.1 SuNDroPS: an Overview ................................ 36
   3.2 Overview of the Original Systems ..................... 40
# Contents

3.2.1 Preliminary Notions: the CDT Context Model  40  
3.2.2 The Context-ADDICT System Functionality and Modules  43  
3.2.3 Information Flow Processing: PerLa and Tesla/TReX  47  
3.2.4 PervAds: a Push Component for broadcasting Context-aware Notices  50  
3.3 Orchestrating the modules  51  

4 Original PerLa and Tesla/TReX  55  
4.1 The PerLa Language and Middleware  56  
4.1.1 Data Structures  57  
4.1.2 Language Features  58  
4.1.3 Context-aware PerLa  59  
4.1.4 A Stress-test for PerLa  67  
4.2 Tesla/TReX  68  
4.2.1 Data Structures  68  
4.2.2 Language Features  69  
4.2.3 Detection Algorithms  70  
4.3 Comparison of the Two Systems  73  
4.3.1 Examples  73  
4.3.2 Common Issues of the Systems  84  

5 Reengineering and Extending PerLa and TReX  87  
5.1 Reengineering the PerLa Middleware  88  
5.1.1 Middleware Refactoring  88  
5.1.2 Channel  94  
5.1.3 Script Engine  97  
5.1.4 Cassandra integration  97  
5.1.5 HTTP protocol integration in PerLa  98  
5.1.6 PerLa on TinyOS-based Motes  101  
5.2 Extending PerLa  104  
5.2.1 Query Performance Enhancements  104  
5.2.2 Distributed PerLa  105  
5.2.3 PerLa and COP  107  
5.3 Extending Tesla/TReX  116  
5.3.1 RDF-3X SPARQL Engine  117  
5.3.2 TReX Engine Extension  118  
5.3.3 Benchmarking the New Semantic Enhanced Engine  128
6  A parallel, Map-Reduce approach to association rule mining       131
6.1 Preliminary Concepts .............................................. 132
  6.1.1 MapReduce and Hadoop ....................................... 132
  6.1.2 NoSQL Databases ............................................. 133
  6.1.3 The Frequent Itemset Mining Problem ...................... 133
6.2 The Idea – Problem Statement .................................... 136
  6.2.1 Lattice Aided Itemset Enumeration .......................... 137
  6.2.2 Map-Reduce EClat Design .................................... 140
6.3 The framework .................................................... 141
  6.3.1 System Design ................................................ 141
6.4 Results and benchmarks .......................................... 147
  6.4.1 Experimental Results ........................................ 148
  6.4.2 Final Remarks .............................................. 152

7  The Green Move Project: an Application Example               153
7.1 The Green Move Project .......................................... 153
7.2 SuNDroPS in Green Move ........................................ 155
  7.2.1 Context-aware Reservations ................................. 155
  7.2.2 Green Move Dynamic Apps .................................. 158
  7.2.3 GMAdvisor .................................................. 160
  7.2.4 Context-aware Sensors ..................................... 163

8  Conclusions and Future Work                                   167
8.1 Future work ................................................................ 168
  8.1.1 Information Flow Processing ............................... 168
  8.1.2 Context Awareness .......................................... 169
  8.1.3 Data Mining ................................................ 170
8.2 Final Thoughts .................................................... 170

Bibliography ................................................................... 181

A  Relevant Publications .................................................. 191
A.1 Publications on Journals and Magazines .......................... 191
A.2 Publications on Congress Proceedings ........................... 191
A.3 Submitted Book Chapters ......................................... 193
A.4 Technical Reports .................................................. 193

B  PerLa Reengineering: Implementaiton Details ........................ 195
B.1 Middleware Reengineering ....................................... 195
  B.1.1 Functionality Proxy Component (FPC) .................... 195
  B.1.2 Channel .................................................... 196
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2 PerLa on Motes</td>
<td>200</td>
</tr>
<tr>
<td>B.2.1 Design Details</td>
<td>201</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Nowadays people are surrounded by a high quantity of data, coming in different and heterogeneous formats. However, humans cannot exploit the whole power of this data without the appropriate aid of digital means. The SuNDroPS system (Semantic and dyNamic Data in a Pervasive System) [98] described in this thesis aims at supporting the (mobile) users with a context-aware approach, allowing them to consider only a small set of data, automatically selected by the system itself according to their current interests. Part of this data (e.g., user data, service data, network and vehicles static data) is stored in a traditional information system (based on a relational data structure) while another big part of them—those data that come from sensors, system logs and every other run-time, dynamic data—need to be treated as data streams and dealt along with the other data, which requires research beyond the current state-of-the-art.

The SuNDroPS system empowers the Context-ADDICT (Context-Aware Data Design, Integration, Customization and Tailoring) [27] system by adding new features to manage high loads of dynamic, sensor-coming data, allowing the system to support all context-aware operations from the user queries to specific setup of sensors, together with data mining algorithms
useful for extracting unknown information from the data flowing in the system.

The main concern of this thesis is to present the overall architecture to manage the flow of (possibly semantically enriched) information contained in these streams and to study how to extract useful knowledge from it and from the more traditional, static datasets. Due to the large amount and to the simple structure of sensor-collected data, we have adopted as storage solution a NoSQL database like approach, adopting the Hadoop framework for data distribution and analysis in a distributed scenario, exploiting the whole power of this framework and of the MapReduce programming paradigm.

The system has to deal with several different aspects of data management. The main aspect is related to the on-the-fly query answering: the system must react in real-time to queries issued towards it by the user, giving the most accurate results. Moreover, the data gathered from the sensors are very useful to sense the environment and understand what is happening. By the way, storing (partially) aggregated historical data for performing further analysis, allow the framework to understand if the system behaves following some kind of sequential pattern or not, allowing to optimize query answering time (using part of previously computed results); it also permit to give the most accurate answer to each user query based both on data flows and historical data. The data flowing in the system also need to be analyzed to extract new and useful information not known before, allowing to increase the query answers accuracy.

Due to the intrinsic dinamicity of the system, a large part of this analysis task has to be executed in real-time, otherwise its results might become outdated, and reasoning could lead to wrong conclusions. On the other hand, also historical data analysis is important since it makes it possible to understand some aspects, like performance degradation in time, that real-time analysis is not able to evince. The thesis will propose both enhancements to existing systems, methods and techniques which allow to query this data stream in real-time (while at the same time preprocessing and loading data into a persistent data store for further off-line mining) and an alternative mining solution that makes it possible to mine big data obtaining results in short time.

The work aims at building a real-life system based on the above technologies, plunging more deeply into some aspects which remain to be investigated like semantic data stream processing and cloud-based data mining algorithms. Prototypes of (parts of) the system have been produced, within the application domains of car-sharing services and support for citi-
zens’ mobility in the Green Move\(^1\) (GM) project for electrical car sharing, carried out at Politecnico di Milano in the years between 2010 and 2013.

### 1.1 The SuNDroPS system: Context-ADDict Revisited

The main goal of the Context-ADDict system first, and then of SuNDroPS, is to create a middleware infrastructure to support the design and development of context-aware data-intensive applications. The focus is on mobile, possibly peer-to-peer applications, where the notion of context can be exploited to provide the user with a filtered view over the data, retrieving only the information relevant to the user in her current context. The main issues addressed by the Context-ADDict system are:

1. definition of a user and context model,
2. semantic extraction from data sources,
3. support for query distribution and synchronization of data,
4. semantic integration of different heterogeneous data sources,
5. context-aware data tailoring based on the above modules (a).

The Context-ADDict framework is based on a core Domain Ontology, whose preeminent aim is to support the data integration and querying part. A domain ontology is an RDFS/OWL ontology that describes the peculiarities of the scenario in which the users are immersed and the system is used.

SuNDroPS aims at making the notion of context transparent to the user and to the application: the user contexts can now mostly be inferred from sensor readings and historical data, relieving the user and the apps developers from the need of choosing or detecting the context on their side, at least for all those contexts that do not strictly depend on user choices or on her private interests. SunDroPS has several components; some of them inherited from the Context-ADDict system. The general architecture of the whole system will be presented in details in Chapter 3.

In addition to the Context-ADDict modules, SuNDroPS interfaces two more components that allow to handle sensor data and WSNs (Wireless Sensors Networks): TRex and PerLa. These new modules enrich the set of datasources that could be managed by the system, allowing it to manage events streams, datastreams and WSNs. They act mainly as wrappers for sensor queries, allowing to retrieve also from the sensors the information useful to the computation of the integrated answer to the user query. On the other side, these modules also enrich the proactive and reactive parts of the system, thus allowing it to improve automatic deduction of context...

\(^1\)http://www.greenmove.polimi.it
Chapter 1. Introduction

features, enhancing the user experience. Both systems provide their own client to be run on sensors as part of the middleware.

Among the already mentioned data analysis activities, a fundamental one consists in the automatic derivation of the user preferences in her current context, selecting all and only the most relevant data in the context. An initial part of this work has been carried out in [87], which present the methodologies implemented in the ADaPT system [88] that allows to collect the preferences of the users in their different contexts and, by learning the user preferences in each context, provides to the final user the most relevant information each time a given context arise, tailoring the information stored in the database according to the user needs.

For all the data mining activities SunDroPS provides a new data mining component, MR-Miner, that implements MREClat, a Map-Reduce variant of the EClat algorithm described in [139], which is a frequent-pattern mining algorithm allowing the system to reduce the processing time required to perform the frequent itemset mining task. The output of MR-Miner is then used as input by the Run-Time Preference Manager, allowing it to speed up the preferences automatic derivation step, reducing the response time necessary to further personalize the current user context.

1.2 Application Case Study: The Green Move Project

The Green Move (GM) project [10], whose aim is a zero-emission-vehicle (ZEV) sharing service for the city of Milan, has provided the opportunity to test the main components of SunDroPS. In Green Move the core services are surrounded by a social-like platform to support users in a large urban context.

A GM user cannot reserve a specific vehicle but she can ask for a vehicle of a particular class (e.g. car, scooter, van, ...) using the GM website or the GM android app.

Once the user has made a reservation, the system automatically chooses one of the vehicles of the selected class and assigns it to the reservation (using a reservation-assignment model that will be briefly presented in Sect. 7.2.1), sending a communication (via email or sms) to the user. The GM system is based on a hardware component, called Green-Ebox (GEB), which is installed on each vehicle, allowing it to interact with the system. Once the GEB is installed the vehicle can be reserved by the users of the sharing service. Once the reservation begins the user can unlock the vehicle using hers smartphone: after having downloaded a ticket from the server, she can take control of the vehicle sending – through hers smartphone – the ticket
to GEB via Bluetooth or NFC.

The ZEV-sharing service provides four different service configurations, designed to meet different user-category requirements:

**condo-sharing** for users who live in apartments and decide to share a (set of) vehicle(s) for daily usage (e.g. going to the supermarket, taking children to school, ...). This configuration is usually two-way: the user returns the vehicle to the same place where he/she got it, typically the condo parking;

**firm-sharing** for firms outsourcing their company vehicles to the GM sharing service. This configuration is usually two-way like condo-sharing;

**world-of-services** users use a GM vehicle to reach a point of interest – e.g. a museum, or a department store, which has an agreement with GM – offering dedicated services to GM customers (e.g. having the museum ticket charged on the GM monthly bill to skip the queue at the ticket office). This configuration is typically one-way: the user shall release the vehicle at a GM reserved parking nearby the aggregation point, any further usage will be independent: the user could reserve a different vehicle (or the same if it is available) for moving away after having enjoyed the service;

**generic** users whose needs are not represented in any of the previous configurations.

Each user reservation can also contain some personalization, e.g. the need of the child-seat or her usual service configuration. If this specific personalization is required often the system can learn and use it to try to predict the users needs during the reservation phase.

The GM system also aims at providing a complete user experience of core and accessory services, like integrated services offered by GM commercial partners in the city, service and traffic information and advertising based on users’ interests and GPS position. Also to fulfill such objectives a context-aware approach is necessary, in order to offer and manage situation-dependent services and support processing of data flows to extract interesting information accordingly.

The approach drives the data flows since its gathering phases, even from sensors, selectively retrieving data only in quantity and format useful according to the current context: e.g. driving downtown is different from driving in the suburbs, thus the user reasonably expects different information – like traffic density or the presence of restricted areas – and with different frequencies (as an application example will show in Chapter 7).
Chapter 1. Introduction

With a growing number of vehicles and users, the amount of collected and exchanged data will make the efficiency of advanced services a critical issue: in this perspective, the use of selective and efficient context-aware data gathering processes, which filter the information on the basis of the context(s) of its acceptor(s), can again improve the effectiveness and scalability of the system.

The context-aware system is based on methods and techniques that will be introduced in Chapter 2 and then detailed in Chapter 3.
CHAPTER 2

State of the Art

Contents

2.1 Context-Awareness ................................................. 8
   2.1.1 Communication ............................................. 9
   2.1.2 Situation awareness and behavioral variations ............ 10
   2.1.3 Context-Oriented Programming ............................ 10
   2.1.4 Managing knowledge chunks ............................... 13
2.2 Information Flow Processing ................................. 14
   2.2.1 Data Stream Management Systems and Languages ........ 15
   2.2.2 Complex Event Processing Systems ....................... 24
2.3 Map-Reduce and NoSQL Databases ....................... 27
   2.3.1 The MapReduce Programming Model .................... 27
   2.3.2 Hadoop and Map-Reduce ................................ 27
   2.3.3 NoSQL Databases ......................................... 28
2.4 Data Mining and Frequent Itemset Extraction ............ 29
   2.4.1 Implementations of FIE in Distributed Environment ..... 30
   2.4.2 Map-Reduce Implementations of FIE ................... 32

This chapter gives a brief overview of the state of the art systems, together with currently available techniques for solving the problems we have to deal with. Sect. 2.1 refers to context-aware systems
Chapter 2. State of the Art

and techniques on which the core part of the SuNDroPS system is based, Sect. 2.2 presents the state of art about Information Flow Processing, focusing on Complex Event Processing systems (CEPs) and Data Stream Management Systems (DSMS). PerLa and TReX, two different engines each reflecting one of the paradigms, will be integrated in the new system, allowing it to retrieve and automatically react to changes in the surrounding environment. Sections 2.3 and 2.4 present Map-Reduce, NoSQL and data mining state of art algorithms and systems, on which is based MR-Miner, the frequent itemset mining algorithm used in SuNDroPS as a starting point for historical data analysis.

2.1 Context-Awareness

Over time, context has evolved from the notion of a simple profile into a collection of all pieces of information that can be used to characterize the situation of an entity [57], be it a person, a place, or any other relevant object/aspect in the interaction between a user and a system. This gave rise to many applications of context-awareness, some of which discussed in the rest of this section (see also surveys in [16, 20, 24, 100, 105, 127]).

The concept of context has been developed and refined since the first approaches to ubiquitous computing [135], the research area of everywhere computing systems, which aims to provide help and information to a (possibly mobile) user querying the system. The first idea of context was limited to time and location: the location – based on cellular network/GPS positioning – is sensed and updated through time – based on the internal clock of a mobile device – (e.g., the Olivetti’s Active Badge [134] in 1992), then made available for further uses like, for instance, passing a phone call to the room where the receiver actually is. The characterization of context was then extended to other elements (like in the PARCTAB system [116]) that may change or influence the data available to a user and the idea of context-aware software was then introduced in [115].

Context, as a first-class object in a logical account of knowledge reasoning and situation-awareness, is the subject of many works also in artificial intelligence. McChartry [86] sought to bring out the contextual element in reasoning about action and planning, to manage general-circumstances related contexts and make use of such understanding in an AI system. His account is expressed in the language of Situation Calculus, a fragment of second-order logic where situation is represented as main (i.e. first-class) object of the representation. The high-order aspect is due to the application of the Closed-World assumption.
2.1. Context-Awareness

Giunchiglia and Ghidini [65,66] worked to provide foundations for context-aware reasoning with the Local Models Semantics. In particular, they define two principles: i) Locality, i.e. only part of the knowledge is useful for the reasoning process (the local part), and ii) Compatibility, i.e. different contexts have compatible kinds of reasoning. In [65], These different contexts are used to introduce the reasoning with viewpoints.

The use of context to enhance and focus the reasoning provides a powerful tool in such approaches. However, they have an “utilitarian” definition for context and non-formal modeling for its instances: they lack context modeling coupled with a formal representation mechanism.

In the following, we briefly list various approaches to context modelling and management in order to provide the reader with some insight about why context awareness is useful and how it has been exploited in software systems. As pointed out in [93, 106], applications may adopt a context aware perspective in order to manage communication, situation awareness and behavioral variations and handling knowledge chuncks.

2.1.1 Communication

One of the main perspective to be evaluated from the communication point of view is the model capability to adapt content presentation to different channels or to different devices (e.g. system communication with the users). CC/PP (Composite Capabilities/Preference Profiles) is a W3C recommendation providing a profile, which is a description of device capabilities and user preferences. Following the CC/PP recommendation, CSCP [33] is a Mobility Portal combining application-spanning media conversion and transcoding with application-specific information filtering. MAIS\(^1\) (Multi Channel Adaptive Information System) has the objective of configuring the software on board of the mobile device, based on presentation, device characteristics and available channel.

Another aspect is the agreement and shared reasoning among peers (communication among users or systems). For example, in CoBra [42] an intelligent context broker maintains and manages a shared contextual model on behalf of a community of agents, while in [94] Web information filtering on a P2P basis is provided. CoBra also provide an agent-based architecture that supports context aware computing in intelligent spaces, i.e., physical spaces (e.g., living rooms, vehicles, corporate offices and meeting rooms) populated with intelligent systems, allowing it to be used in order to build smart environments.

\(^1\)http://www.mais-project.it/
2.1.2 Situation awareness and behavioral variations

Modeling location and environment aspects (i.e. modeling the physical situation) is one of the main features to be considered from the situation awareness point of view. In CodaMos [103] every device will contain its own context specification with a full description of its provided services, plus pointers to relevant information on the device in its environment. The COntext MAgnagement oNTOlogy (COMANTO) [111] approach offers a hybrid scheme that combines a location-based context model and some context ontologies. The ontology captures the domain knowledge that is useful to support reasoning over local and distributed data, the distinction between fixed and movable resources and grants interoperability. The system known as Semantic Coordinator Over Parallel Exploration Spaces (SCOPES) [94] applies a “semantic reconciliation” to different databases, mimicking the same approach that would be used by a human to integrate data sources by using agents to access knowledge. In SCOPES “context is defined as inter-schema mappings between the schema of the local database and that of the remote database” [94] and through negotiation, even in the presence of uncertain information.

Then a crucial issue is modeling what the user is currently doing (personal situation). In [71] Activity Theory is used to represent concepts such as roles, rules and tools, which have important impacts on users activities. Activity Theory also maps the relationships amongst the elements that it identifies as having an influence on human activity.

Finally, the model can make the user interaction implicit (adapting the information to the user’s needs). For instance, QUALEG [123] proposes a unique combination of a global ontology with a dynamic context, for dynamically adapting eGovernment IT tools to a multi-lingual and multicultural setting.

Context-Oriented Programming (henceforth COP) introduces support for context-awareness in the execution flow of a program. Since the COP paradigm will be extensively used in part of the extension of the PerLa language and middleware proposed in this thesis (Sect. 5.1), some more details are proposed in the next section.

2.1.3 Context-Oriented Programming

The COP paradigm [11, 70, 113] addresses context-aware design of programs introducing several techniques, supporting the traditional programming languages adaptation and providing abstractions to manage contextual behavioral variations. A lot of libraries have been introduced to implement
COP in most of common programming languages, such as Java, Python, etc...

COP treats contextual information explicitly and provides a complete support to behavior adaptation at run-time. The behavior adaptation provided by COP techniques is achieved by using layers [70].

Context for COP is simply intended as any set of information, which is computationally accessible by any part of the system and may determine behavioral variations; when a context is active during the execution of a program, this must be able to behave in one of a determined set of alternative ways.

The main concept of COP is the behavioral variation. Each behavioral variation is related to a specific piece of context and can be dynamically activated and deactivated at run-time, enacting a behavioral change; it represents the modularization unit of such piece of behavior. Many different solutions have been proposed to create a connection in the program between the contextual information and dynamic behavioral variations; since the layer-based model [55] is the most widespread, we will refer to it in the following.

Layers are entities which group related context-dependent behavioral variations and contain partial method definitions that implement the functionality of behavioral variations. Layers can be dynamically activated and deactivated at run-time. Depending on the current context, different layers will be selected for further program executions. Several solutions have been proposed in COP for layers activation. In JCop library, layers are first-class entities, which can either be defined within the classes for which they provide behavioral variations (layer-in-class), or in a dedicated top-level layer similar to an aspect (class-in-layer) [11]. However, the first case is the most relevant, because all the possible variations are directly encapsulated in classes which need adaptations. Layer activation is dynamically scoped: it affects the behavior of the program not only for the method calls syntactically inside the code block, but also for all the calls triggered in turn.

Different layers activation/deactivation strategies can be adopted. Three of the most used strategies are: i) Dynamically scoped activation, the original mechanism introduced to manage layers’ activation [45]. Layer activation is context-based and its methods and functions are made available for use. ii) Indefinite activation provides a way to change the behavior of a program until further activation notice is received. The indefinite activation technique is very useful when program-wide behavioral changes are needed, especially in its global version, but otherwise difficult to manage.
Chapter 2. State of the Art

because of the lack of scope limitation. iii) Per-object activation, opposite to indefinite activation, provides layer activation in the scope of specific objects in the program. Such a mechanism provides a fine-grained control over layers’ activations and is very useful when only a small subset of the program needs to be context-aware. Many other activation strategies have been developed and in [113] more details can be found. Despite the extensive flexibility provided in COP, context stays in the background, with no explicit modeling. Context is implicit in the description of the program, embedded in the object-oriented nature of the supported languages [70], and it is only materialized through layers and related activation / selection mechanisms.

Layers can be dynamically composed to create more complex activation chains starting from basic layers describing simple methods and operations.

In COP, base methods describe the normal behavior; context-dependent behavioral variations [11] are represented as partial method definitions. Methods not affected by layers are called plain methods while those involved in layers composition are called layered methods. The base method is executed when no active layer provides a corresponding partial method, and at least one partial method definition exists. The dispatching mechanism is intuitive: when activated, layered method calls are dispatched to the partial method provided by the layer. So, partial methods provide a different behavior compared to base methods and they can be executed before, after, or around a base method.

Keywords with and without provide an explicit layer composition mechanism, because they specify which layers must be activated (deactivated) for the scoped block. However, an explicit layer composition, based on with statements, is not enough in many cases; for this reason a lot of different activation mechanisms have been proposed [113].

For the goal of this work we focus on dynamically scoped activation and in particular on declarative layer composition and conditional composition.

Dynamically scoped activation is a general model in which the with statements activate a sequence of layers in the code block.

In a pervasive environment, context is event-based, and events can be synchronous, especially those bounded to changes in the environment and in the system, or asynchronous, such as the ones related to the user interactions. JCop provides specific constructs for declarative and event-based composition.

The declarative layer composition model consists of a logic concatenation of predicates, which represent events, and of a composition block, which contains some with (or without) statements. Moreover, it introduces
two new constructs in order to solve the problem of scattered with statements: i) the keyword on which identifies an event in the program execution flow; ii) an optional keyword in to bind the object on which the composition declaration should be evaluated.

This model is a good solution, in particular in case of composition based on predictable events. In situations that are connected with unexpected events, the explicit specification of those events with on statements can be complex and could become really verbose.

For this reason JCop, in addition to on, introduces the when statement. This statement specifies an expression that represents an event occurred in the program control flow and, if it is evaluated to true, layered methods in the code block are activated. Composition made with when statements is called conditional. The on statement specifies the points “where” the composition takes place, whereas the when statement allows to declare “when” the layers composition must start. Furthermore, JCop provides a first-class context construct; contexts are special singleton types that cannot be instantiated. The construct can host both declarative and event-based composition statements and auxiliary methods and fields. For a more complete explanation of layers composition models see [11].

2.1.4 Managing knowledge chunks

First of all a model must determine the set of application/situation relevant information (i.e. the information management perspective). In [112] the users specify their own current context when requesting data, denoting the part that is relevant to their specific situation. The work propose a relational model extension to deal with context: e.g. an attribute may not exist under some contexts or have different facets under different contexts. The approach provides a set of basic operations which extend relational algebra. In [25, 130] context is used as a viewpoint mechanism that takes into account implicit background knowledge to semi-automatically tailor context aware views over a database. The Context-ADDCIT project [27] – to which this thesis work propose some enhancements and extensions – provides a hierarchical model of the context in terms of observable parameters that have a symbolic internal representation within a context schema, the Context Dimension Tree (CDT). A fundamental aspect of the whole context management architecture is to keep the elaborations, queries and data retrieval operations necessary to present the context-tailored information transparent to the user. The core of Context-ADDCIT architecture consists of components dedicated to: 1. Design-time context management, support-
Chapter 2. State of the Art

1. The design of the context schema of the specific scenario at hand, and 
2. Context-aware personalization, determining the currently active context, applying contextualization to queries and delivering of context-aware data. 
3. Direct data access access to data providing an early-tailed, context-aware unified answer. Further details will be provided in Sect. 3.2. Within the Context-ADDICT framework, CARVE [28] propose an approach to modeling and design context-aware systems with efforts towards a context-aware methodology based on view definition, specifically aimed at knowledge access management in context-aware applications. CARVE provides context-aware views which can be applied to the data according to the specific context it is applied within. In a similar fashion to the viewpoint abstraction mechanism proposed in [90], different situations (contexts) generate different needs for information, thus requiring the information system to adapt, providing a different set of views for each available context.

Other models determine the set of application/situation relevant services. [108] proposes efficient context aware service discovery in pervasive environments. In [67], perceptual processes provide a means to detect and track compositions of entities and to verify relations between them. The design problem is then to determine the appropriate entities (resp. relations) that must be determined (resp. verified) with respect to a task or service to be provided, in a potentially infinite set.

In the end, models can determine the set of application/situation relevant behaviours. In [90, 132] an information base is decomposed into (possibly overlapping) subsets, referred to as contexts. A context could be recursively contained in other contexts and it is assigned to one or more owners, authorised to perform any operation or transaction on it. Context-specific naming and representation of conceptual entities, relativised transaction execution, operations for context construction and manipulation, authorisation mechanisms and change propagation are all the operations supported by the system.

2.2 Information Flow Processing

There exist two main different approaches to information flow processing: one more related to the point-of-view of the database research community (Data Stream Management Systems – DSMS) and one that solve the problem from the software engineering perspective and is more related to the distributed systems community (Complex Event Processing Systems – CEP).
2.2. Information Flow Processing

The main differences among those two paradigms lays beneath their different way of considering the stream.

- In Data Stream Management Systems the stream is seen as an infinite set of timestamped tuples, ordered by their own timestamp. In some advanced implementation tuples could be also annotated with an expiration time that lets the system know when a tuple has to be discarded.

- In Complex Event Processing Systems instead stream represent an infinite sequence of events. Each event is characterized by an identifier of its type and a timestamp, while for discarding events in those systems rather than an event expiration time the paradigm provide event consumption policies, to let the system discard an event (or a set of events) after having used them for detecting a given sequence.

While DSMS propose a SQL-like query paradigm, trying to keep those systems similar to traditional relational databases, CEP are more likely to traditional Publish/Subscribe systems, usually allowing their querying by means of logical rule.

Since none of the two approach excludes the other and there exist scenarios in which one perform better than the other (CEP usually performs better in fields like surveillance while DSMS usually are better sources for data analysis), we have decided to allow both those IFP paradigm to be integrated in our final system: PerLa is an example of DSMS (and WSN management system), while TReX is a complete CEP system.

For both the state of the art on DSMS and CEP we will refer to existent and updated surveys provided in [52, 95].

2.2.1 Data Stream Management Systems and Languages

The main issue encountered while trying to apply the traditional DBMS model to data stream processing is related to the necessity to build a persistent storage where all the relevant data is kept, and whose updates are relatively infrequent (almost true for traditional relational databases but not for data streams). This approach has a negative trend on the performance if lots of processing rules are necessary (more than a given threshold) or when the data arrival rate is too high. For systems that have to process lots of information as they flows in the system those limitations are unacceptable.

Data Stream Management Systems (DSMSs), a new class of systems oriented toward processing large streams of data in a timely way have
been developed by the database community to overcome those limitations. DSMSs differ from conventional DBMSs in several ways:

- Streams are usually unbounded, while tables are usually bounded.
- The data arrival order is unpredictable and no assumption can be made on it.
- It is necessary to deal with streams in real-time, since size and time constraints make it very difficult to store and process data stream elements after their arrival.

Queries in DSMS are deployed once in the system and they keep to produce results until they are removed or the system stops (they are called Continuous Queries). In a DSMS, queries can be executed periodically or continuously as new stream items arrive. The DSMS interaction model is completely different from traditional DBMSs: According to the Database-Active Human-Passive (DAHP) paradigm [3], the system actively notifies the presence of updated information according to installed queries to the users that do not have to request them explicitly.

Several different DSMSs implementations have been proposed in different projects and contexts during past years, having different semantics associated to the queries. The answer to a query can be seen as an append only output stream, or as an entry in a storage that is continuously modified as new elements flow inside the processing stream. An answer can be either exact or approximate, depending on the system memory that has to - and can - be used to store all the required elements of input streams' history [14,131].

As stated in [52], a DSMS can be modeled as a set of standing queries \( Q \), one or more input streams and four possible outputs:

the **Stream** is composed by all the elements of the answer that are produced once and never changed;

the **Store** contains the parts of the answer that may be then changed or removed at a certain future time point. The Stream and the Store together define the current answer to queries \( Q \);

the **Scratch** represents the working memory of the system, where it is possible to store data useful to compute the answer but that is not part of the answer;

the **Throw** is a sort of recycle bin, used to throw away unneeded tuples.
2.2. Information Flow Processing

The model described above is the most complete to define the behavior of DSMSs, showing how a DSMSs only approach cannot entirely cover the needs for IFP, introducing the need of some other kind of paradigm for complex patterns detection and notification.

Several different systems and languages have been developed during different projects that need to process data flowing in stream format.

TelegraphCQ and StreaQuel  

TelegraphCQ [39] is a general-purpose continuous queries system based on a declarative, SQL-based language called StreaQuel.

TelegraphCQ is implemented as a C++ PostgreSQL extension. The compilation of rules into query plans is made through adaptive modules [13, 104, 125], which dynamically decide how to route data to operators and how to order commutative operators. As explained in [124], thanks to this modularity, TelegraphCQ can be spread on multiple machines with a clustered distribution approach, taking into account processing constraints only.

StreaQuel is a relational, SQL-derived stream query language. StreaQuel manages unbounded flows by means of its native window operator "Windows". Several WindowIs operators may be used in a query, one for each input window, embedded in a for loop, which is part of each rule and defines a time variable indicating when the rule has to be processed.

The general assumption behind this mechanism is that it must consider an absolute time model. By adopting an explicit time variable, StreaQuel enables users to define their own policy for moving windows. As a consequence, each window may contain a different number of elements, since its dimension is not bounded a priori.

Since the StreaQuel WindowIs operator can be used to capture arbitrary portions of time, TelegraphCQ naturally supports the analysis of historical data. In this case, disk becomes the bottleneck resource. A shedding technique called OSCAR (Overload-sensitive Stream Capture and Archive Reduction) is proposed in order to reduce time latency due to disk data retrieval. This technique organizes data on disk into multiple resolutions of reduced summaries and depending from the data arrival rate, the system picks the right resolution level to use in query processing [40].

NiagaraCQ and XML-QL  

NiagaraCQ [43] is an IFP system for Internet databases, providing a high-level abstraction to retrieve information from a frequently changing environment like Internet sites. NiagaraCQ is based on the XML-QL language [56].
Chapter 2. State of the Art

XML-QL is a declarative, relationally complete query language for XML; its simple structure allows to easily extend well known relational database query optimization techniques extracting data from existing XML documents and creating new ones.

The initial draft of this language is presented in a W3C note².

In [56] the authors present also an extension of XML that allows to manage ordered XML documents. This extension introduce also in the query language concepts like tag variables, regular path expressions, multiple sources integration and user-defined functions that allow advanced XML processing.

A time interval is associated to each rule, defining its validity period. Rule processing could be triggered by timers or changes in data flow; the former are evaluated periodically, while processing of the latter is driven by notifications of changes received from information sources. In both cases, a table containing the results is updated instead of producing an output stream. Once a rule has been evaluated, a new evaluation of it would not produce any additional result, in such a way that we can say that NiagaraCQ presents a selected consumption policy (information items are used only once).

As for the rule, results can be reported to the users in a push or in a pull way. For each rule, it is possible to specify a set of actions to be performed just after the evaluation.

While the NiagaraCQ engine is designed to run in a totally centralized way, information sources are actually distributed. To increase scalability NiagaraCQ uses an efficient caching algorithm, which splits operators into groups, keeping members of the same group in the same query plan, thus increasing performance.

OpenCQ Like NiagaraCQ, OpenCQ [79] is an IFP system developed to deal with Internet-scale update monitoring. In OpenCQ rules are divided into three parts:

- the definition of the operations to be performed on data (in SQL);
- the activation trigger condition;
- the stop condition

OpenCQ presents the same processing and interaction models as NiagaraCQ.

²http://www.w3.org/TR/NOTE-xml-ql/
2.2. Information Flow Processing

OpenCQ has been implemented on top of the DIOM framework [78]: a client-server communication paradigm is used to collect rules and information and to distribute results, keeping the rules processing completely centralized. Wrappers allowing the integration among different and heterogeneous sources are used in order to transform the input flow to the required results output flow. Each wrapper can behave in a pull or in a push way, depending on the source behavior: in the first scenario the wrapper periodically retrieve data from the source while in the second one it simply waits for the source to send new data to it, that only has to prepare them for the processing.

Tribeca Tribeca [129] has been developed for network traffic monitoring, and it is fit to this particular domain. Its rules takes a single information flow as input and produces one or more output flows. Rules are expressed using an imperative language, defining the sequence of operators an information flow has to pass through.

Three different operators exist in the language: selection (called qualification), projection, and aggregate; it is also possible to split (merge) flows by means of ad-hoc multiplexing (demultiplexing) operators. Tribeca supports both count-based and time-based windows, which can be sliding or tumble. Windows are used to specify the part of input flows to be processed when performing aggregates. Since items do not have an explicit timestamp, the processing is performed in arrival order. Since the windows are time-based it is impossible to count the number of information items captured at each iteration. Moreover, a rule may be satisfied by multiple and different sets of elements (multiple selection policy), while an element may be part of the computation in more than one iteration, as there is no explicit consumption policy. Rules are translated into direct acyclic graphs in order to be executed, applying graph optimization techniques during the transformation in order to improve performance.

Stream and CQL Stream computes a query plan starting from CQL rules (CQL, Continuous Query Language, is a SQL-Like declarative language developed by the DB group of the Stanford university in the same project), taking into account predefined performance policies in order to schedule the execution of the operators in the plan.

CQL [12] basic concept is the one of transforming rule, which provides a unified syntax for processing both information flows and stored relations.

CQL provides a clear semantic for the rules, together with a simple language to define them. CQL defines operators that manage the data stream:
due to the fact that a stream is a potentially infinite bag of timestamped data items, these operators extract finite bags of data. The CQL operators split into three different classes 2.1:

**relation-to-relation** operators derived directly from the SQL operators are the basic, core operators of the language and allow the definition of rules using a SQL-like standard notation (e.g., Relational algebraic expressions);

**stream-to-relation** operators transform streams in relations; the most known operator of this class is the *window* operator that support flow processing by defining *sliding*, *pane* and *tumble* windows on the stream;

**relation-to-stream** operators (*IStream, DStream and RStream*) define how to generate a new information flow starting from tuples.

CQL uses a multiple selection policy, associated with a zero consumption policy (the tuples in the stream are never consumed, they still exist after their processing was completed).

Stream provides two shedding techniques to deal with the resource overload problem, that is typical of data streams:

- the first one [14] applies load shedding on a set of sliding window aggregation queries to reduce the impact on limited computational resources

- the second one [126] discards the state of the operator for a set of windowed joins, reducing the impact on the memory occupation (in case of limited memory)
2.2. Information Flow Processing

The CQL model inspired the design of different RDF Stream Processing engines [18,34,73], and currently there are different implementations (e.g. C-SPARQL, SPARQLstream, CQELS) of the adapted model. The stream and the relation concepts are mapped to RDF streams and to set of mappings (using the SPARQL algebra terminology\(^3\)), respectively. To highlight the similarity of the RSP operators to those of CQL, similar names are used: S2R, R2R and R2S to indicate the operators analogous to stream-to-relation, relation-to-relation and relation-to-stream operators.

**Aurora/Borealis and SQuAl**  
*Aurora* [1] is a general-purpose DSMS that defines transforming rules using an imperative language called SQuAl.

*SQuAl* defines rules in a graphical way, by means of boxes and arrows, connecting different operators of the rule. The language operates on windows, processing several tuples at a time, applying a specific user-defined function, or on a single tuple at a time. This allows information tuples to be used more than once (multiple selection policy), while there exists no consumption policy.

SQuAl allows multiple input processing and multiple results outputs, allowing to constraint the output by means of ad-hoc QoS policies in order to fit users and/or application requirements. SQuAl defines the necessary load-shedding policies and customize the system behavior accordingly to QoS constraints.

Output flows may be directly linked to higher-level applications, making it possible to customize the system behavior according to application requirements.

Aurora enforces QoS constraints to automatically define shedding policies. Input and output flows do not have an associated timestamp, but are processed in their arrival order.

One of the most interesting feature of Aurora is the possibility to include intermediate storage points inside the rule plan allowing to keep historical information and to recover after operators failure.

The processing is organized by a scheduler, taking an optimized version of the user defined plan and choosing how to allocate computational resources to different operators according to their load and to the specified QoS constraints.

Some project extensions have been created in order to investigate distributed processing both inside a single administrative domain and over multiple domains [44]. The goal of these extensions, called *Aurora*\(^*\) and *Medusa*, is that of efficiently distributing load between available resources.

\(^3\)Cf. http://www.w3.org/TR/sparql11-query/
in these implementations, processors communicate using an overlay network, with dynamic bindings between operators and flows.

Recently the two projects have been merged into the Borealis stream processor [2], which includes all the two previous projects features.

**Gigascope and GSQL**  *Gigascope* [46, 47] is a DSMS specifically designed for network applications, including traffic analysis, intrusion detection, performance monitoring, etc. Its main concern is to provide high performance for the specific application field it has been designed for. Gigascope is based on a declarative, SQL-like language, called *GSQL*, which includes only filters, joins, group by, and aggregates.

The processing paradigm used by GSQL is very different from the usual one of the other data stream engines. To manage blocking operator, instead of introducing the usual concept of window, it assumes that every input tuple has at least one ordered (monotonically increasing or decreasing) attribute (e.g. the timestamp), using this attribute as a constraint in the processing. For example, the join operator, by definition, must have a constraint on an ordered attribute for each involved stream.

This paradigm has a limited set of application domains (the ones in which it is possible to make strong assumptions on the nature of the data and on their arrival order), while it makes the processing semantics easier to understand and similar to the traditional SQL semantics.

Gigascope translates GSQL rules into basic operators, composing them into a processing plan, using optimization techniques to re-arrange the plan according to the nature and the cost of each operator.

**Stream Mill and ESL**  *Stream Mill* [15] is a general-purpose DSMS based on ESL, a SQL-like language with ad-hoc constructs designed to easily support flows of information.

*ESL* is a highly expressive rule-based language which mix the SQL declarative syntax with the capability of creating custom aggregates using an imperative paradigm provided by the language.

To obtain those aggregates, the language offers the possibility to create and manage custom tables, used as variables of a programming language during the information flow processing.

The processing model is based on a loop on information items, which users control by associating behaviors to the beginning of processing, to internal iterations, and to the end of processing. Behaviors are expressed using the SQL language.
2.2. Information Flow Processing

This approach is somehow similar to the one we described for CQL: data flows are windowed and temporarily seen as relational tables and queried with traditional SQL rules. ESL extends this idea by allowing users to express how to perform the processing of those tables, setting the selection and consumption policies rule by rule allowing a very fine-grained control over the stream management policies. Since processing is performed on relational tables, the ordering of produced items may be separate from the input order.

Stream Mill keeps the traditional architecture of a database system: rules are compiled into a query plan and then dynamically schedule the operator processing flow. The implementation of Stream Mill as described in [15] shows a centralized processing, where the engine exchanges data with applications using a client-server communication paradigm.

PerLa PerLa [119] is a framework to configure and manage modern pervasive systems and, in particular, wireless sensor networks. PerLa adopts the database metaphor of the pervasive system: such approach, already adopted in the literature [82], is data-centric and relies on a SQL-like query language. PerLa queries allow to retrieve data from the pervasive system, to prescribe how the gathered data have to be processed and stored and to specify the behaviors of the devices.

The PerLa framework will become the SuNDRoPS WSN management component, since it can be deployed on different kind of sensors, allowing them to be integrated in the system. Further details about PerLa will be given in Chapter 4, Section 4.1.

C-SPARQL C-SPARQL [17] extends SPARQL (the standard ontology query language⁴) enabling the support for continuous queries on RDF data streams. The main difference between RDF streams and traditional streams is that RDF is a standard and it guarantees interoperability among different systems supporting this language, also allowing reasoning capabilities on the stream, since reasoners can easily deal with RDF data.

C-SPARQL is based on the concept of RDF streams, those are basically RDF annotated streams of data. The language provides standard operators (like selections and projectors) together with ad-hoc stream operators like windows (time-based or count-based); it also provides aggregations, that are crucial operators in a window-based, stream processing engine. It is also possible to merge data coming from different streams both in the selection and in the output phase.

⁴http://www.w3.org/TR/rdf-sparql-query/
Chapter 2. State of the Art

2.2.2 Complex Event Processing Systems

Several different complex event processing systems have been presented in [52]. We briefly review some of them leaving the details up to the original survey:

Traditional content-based publish-subscribe systems  Traditional content-based publish-subscribe middleware [58, 91] represent one of the historical basis for complex event processing systems.

The traditional publish-subscribe model considers information items as messages that flow into the system, from multiple different publishers, while the interests of the subscribers are defined through simple rules. Each rule contains a detection part, dealing with single information items and selecting them using constraints on their content, and of an action part that simply perform information delivery.

Many of those publish-subscribe systems [37, 38, 48, 59, 96, 102, 128] have been designed to work in large scale scenarios. Those systems are based on a distributed set of brokers, cooperating to realize efficient routing of the messages.

Rapide  Rapide [80, 81] is considered one of the first examples of complex event processing system. Rapide is the first system that enables users to capture the timing and causal relationships between events.

The architecture of Rapide consists of a set of different components, those communicate through events. The engine allows both to describe how to perform the detection of a given pattern of events and to specify some properties of interest for the overall system architecture.

Rapide pattern language provides different standard operators, like conjunctions, disjunctions, negations, sequences, and iterations, with timing constraints. In Rapide there exists no notion of absolute time, allowing however to timestamp events once received by the system. Other operators included in Rapide are logical implications and equivalences between events allowing the language to define more general properties rather than the ones definable with standard operators.

Padres  Padres [74] is an expressive content-based publish-subscribe middleware providing a form of complex event processing. It offers primitives to publish messages and to subscribe to specified information items, using detecting rules expressed in a pattern-based language.

The main difference between Padres and traditional publish-subscribe is that rules can involve more than one information item, allowing logic
operators, sequences, and iterations. Padres adopts a time model with an absolute time, allowing the users to write complex timing constraints in rules, providing fixed, landmark, and sliding windows.

The system adopts a multiple selection and zero consumption policies and, since the Seq operator is unbounded, it impossible to determine the maximum number of items selected by each rule a priori.

Rules are processed in a fully distributed way, exploiting a hierarchical overlay network. The processing performed inside single nodes is realized using Rete trees [60].

Cayuga Cayuga [29] is a general purpose event monitoring system based on a language called CEL (Cayuga Event Language) which structure strongly resembles the one of traditional declarative languages for databases.

All events are considered as having a duration, and much attention is paid in giving a precise semantics for operator composability.

Cayuga is explicitly designed to work on large scale scenarios, using efficient data structures; in particular, the system provides a custom heap management, it indexes operator predicates and it allows to reuse shared automata instances.

Sase and Sase+ Sase [136] is a monitoring system designed to perform complex queries over real-time flows of RFID readings. The detection rules language is based on patterns: each rule is composed of three parts: event, where and within. The event clause specifies which of the information items have to be detected exploiting the relations between them (by means of logic operators and sequences). The where clause defines constraints on the inner structure of information items included into the event clause. In the end, the within clause expresses the validity time for the rule so it becomes possible to define time-based, sliding windows. Sase pattern language does not allow to detect aggregates, but only simple events.

Sase query plans, to which each rules is translated in order to be executed, have a fixed structure, composed by six blocks:

- the first determines if the information under examination matches the logic pattern expressed in the event clause
- the next three blocks checks for selection constraints, windows and negations respectively
- the last block build the output
Chapter 2. State of the Art

The language does not allow to detect unbounded sequences, since the Seq operator is bounded to set of events to be analyzed.

Sase+ [5,68] extends the expressiveness of the Sase detection language, by including iterations and aggregates as possible parts of detecting patterns, using non-deterministic automatas for pattern detection.

One of the most interesting aspect of Sase+ is the possibility to customize selection policies using strategies that define which events are valid for an automaton transition. The language still not define any kind of event consumption policy.

SCEPter  SCEPter [143] is a semantic complex event processing engine developed at the University of Southern Carolina as part of the Los Angeles Smart Grid project.

In this system, raw events are directly annotated with a RDF-like semantics (so an event is a set of < subject, predicate, object > triples) once a raw events enter the system. Processing rules are defined using the SPARQL syntax enhanced with stream operators (e.g. the window operator).

This system splits the processing of events in two main blocks:

- first of all, the raw event enter the system and, by means of appropriate rules, it is transformed from a raw event (e.g. a set ok key-value paires) to an RDF stream
- every different RDF stream is then processed using the SPARQL language with stream extensions.

The system also allows to store the incoming events into a RDF database for further processing.

TReX and Tesla  T-Rex [51] is a general purpose complex event processing that provide a balance between expressiveness and efficiency. The system APIs allow to publish new events, subscribe to simple or complex events, and to define new complex events through specification rules, expressed using Tesla [49], a detecting language based on patterns.

The Tesla language attempts to provide a high expressiveness through a small set of operators, in order to keep the language simple and compact. The included operators are selection of single events, time bounded sequences, negations and aggregates.

Tesla and T-Rex will be introduced in the SuNDRoPS system to allow it to manage also event streams, because they provide a stable and performant
implementation; Furthermore, their semantics and implementation is not domain-specific, allowing them to be applied in a wider set of scenarios w.r.t. other systems presented in this section that provides only a domain-
pecific implementation and semantics. Further details about TReX and Tesla will be provided in Chapters 4 and 5.

Other available systems are Peex [72] (the only one based on a probabilistic engine), Amit [4], PB-CED [9], NextCEP [122], CED [83, 84], CEDR [19] and DistCED [101].

2.3 Map-Reduce and NoSQL Databases

Map-Reduce and its implementation and NoSQL databases are the basis for the new frequent itemsets mining methodology presented in Chapter 6. This section present a brief literature overview on those technologies.

2.3.1 The MapReduce Programming Model

The MapReduce programming model is a parallel, distributed programming paradigm for clusters based on the Map and Reduce functions.

Input data are split in equal size chunks and processed in parallel and independently on different nodes. A map function process each chunk handling input data and producing an intermediate set of key-value pairs (e.g. \(<\text{key, value}>\)) as output.

Each pair is then sent to one or more reduce functions, based on the key identifier. Each reduce function output a new set of key-value pairs with the results of the operation.

Data partitioning is handled directly and in a transparent way by the framework, as well as job scheduling, fault recovery and intra-nodes communication.

2.3.2 Hadoop and Map-Reduce

Apache Hadoop is a Java-based framework developed by the Apache Software Foundation allowing the distributed execution of data-intensive applications, providing a distributed and reliable storage system (HDFS) and an ad-hoc parallel programming model (MapReduce).

Hadoop supports the MapReduce implemented algorithm execution on an unlimited nodes’ number\(^5\) optimizing the scheduling in order to reduce the communication overhead among different nodes. This framework, originally developed to empower search engines, is used for data-warehousing,

\(^5\)Up to 4000 nodes with 10 MapReduce clients running.
Chapter 2. State of the Art

reccomendation systems, cryptoanalysis, fraud detection and genomic analysis. Facebook®, Yahoo!® and the New York Times® are just few examples of application scenarios.

The Hadoop framework initial version has been developed by Yahoo!; the work is based on Google File System (GFS) and Google MapReduce description reported in [54, 63]:

- **Hadoop Distributed File System** (HDFS) is equivalent to GFS: a distributed, highly-configurable, scalable and fault tolerant file system

- **Hadoop MapReduce** is the MapReduce implementation available in Hadoop, similar to the Google one, which allows to easily produce applications able to process big data in a parallel way on clusters, ensuring reliability and fault tolerance of the computation.

We have chosen Hadoop since it is a general framework, providing very interesting parallelization features that will be the ground of the MRECiaT algorithm implementation presented in Sect. 6.3.

### 2.3.3 NoSQL Databases

“Not only SQL” databases are the raising alternative to pure relational database data storage. They claim to be more performant in many fields (e.g. sensor data log storage, online clicks data storage, ...) and their usage is increasing in real-world applications.

One of the main contribution in the NoSQL database fields has been produced by Google and Yahoo! that respectively developed Google BigTables [41] and Pig and Hive [61, 133] (based on the top of the Hadoop framework).

NoSQL is just the generic category name that collects lots of very different solutions. The main difference among different solutions is in the data storage architecture: NoSQL databases belongs to six main categories:

- **Column Store/Column Families** Those architectures stores data in a basic structure called “Column”: each column store the column unique name, a value and the the insertion timestamp (the timestamp information distinguish among stale and valid content). This structure can seem similar to relational tables, but in column stores there is no constraint on the number of columns associated to a single item: two different items can have different columns, also with different names. Some examples of column-based storage solutions are Hadoop HBase, Amazon SimpleDB, Apache Cassandra and Google BigTables.
2.4. Data Mining and Frequent Itemset Extraction

**Document Store** A document store simply collects a set of documents containing information about the same object. Different object data are stored in different documents, and documents are heterogeneous (each document may contain different fields). Usually fields and values are stored in form of key-value pairs in the document. Examples of document-based storage solutions are *MongoDB* and *Apache CouchDB*.

**Key Value/Tuple Store** Key value stores information in key-value pairs format. Nested key-value structure is usually allowed to let this model gain more expressive power. Examples of this data storages are *MS Azure Table Storage*, *Redis* and *BerkeleyDB*.

**Graph Databases** Graph databases stores data in graph-like structures, like in RDF databases. Examples of graph databases are *Neo4j* and *Infinite Graph*.

**Object Databases** Object databases are based on the notion of object (usually a class in a Object Oriented Programming language like C++ or Java). The idea is the same of the document-databases: an object contains data about a real-world item. Document stores are derived from this notion of object database. Examples of this kind of storage are: *ObjectDB*, *Objectivity* and *Versant*.

**Multimodel Databases** These kind of stores use simultaneously different models, like *ArangoDB* (key-value and JSON graphs) or *OrientDB* (object-database, document-database, graph-database and key-value), trying to exploit each time which are the appropriate features to be used.

This classification is not exhaustive and some of the existent solution (e.g. *KirbyBase*\(^6\), *FileDB*\(^7\), *CodernityDB*\(^8\)) are still not classifiable in the previous categories but they are considered fully NoSQL solution.

### 2.4 Data Mining and Frequent Itemset Extraction

Data mining is the process of extracting interesting patterns from large datasets and involves approaches and algorithms born from the synthesis of methods developed in other disciplines such as artificial intelligence, automatic learning and statistics. The aim of the mining process is to extract

\(^6\)http://rubygems.org/gems/KirbyBase
\(^7\)http://www.eztools-software.com/tools/filedb/
\(^8\)http://labs.codernity.com/codernitydb/
Chapter 2. State of the Art

information from a dataset and turn it into facts not previously known or obvious, easily comprehensible for the humans and suitable for future applications.

In particular, data mining is the basic step of the process of knowledge extraction from raw data; the process includes procedures for cleaning, integrating, selecting and transforming the raw data, applied before the data mining step; then evaluation and visualization operations of the detected patterns are performed to output the results of the process to the user.

As the name suggests, frequent itemsets are sets of items which are found frequently within a dataset. The problem of Frequent Itemset Extraction (FIE) represents the most complex and expensive phase of association rules mining from the computational point of view. The algorithms used for extracting frequent itemsets can be divided into:

- algorithms that generate the candidate itemsets: these algorithms, including Apriori, generate all the potential itemsets and then check if they are frequent;
- algorithms that do not generate the candidate itemsets: these algorithms, including FP-Growth, proceed directly to the generation of frequent itemsets starting from the original dataset.

A further subdivision is based on the type of representation used for the dataset, that can be horizontal (as in FP-Growth) or vertical (as in EClat). Horizontal representation provides the transaction id (tid) followed by all the items contained in the transaction, while in vertical representation each element is followed by the list of the transaction that contains it.

In particular, EClat algorithm exploits the properties of frequent itemset to optimize the search process using the lattice theory.

The approach splits the search space in independent partitions, exploiting the recursive definition of equivalence class which allows a more easy processing in main memory. Each equivalence class, due to the properties of the construction of the lattice, is independent and contains all the information necessary to generate all the frequent itemset that contains.

Moreover, by adopting suitable techniques for exploring the sublattice, it is possible to quickly identify the frequent itemset within the sublattice defined by the class.

### 2.4.1 Implementations of FIE in Distributed Environment

Several parallel implementations of the Apriori algorithm have been proposed by the same authors of the original algorithm presented in [7], study-
2.4. Data Mining and Frequent Itemset Extraction

...


Chapter 2. State of the Art

2.4.2 Map-Reduce Implementations of FIE

The Apriori algorithm could be easily adapted to the MapReduce paradigm, while performances may not reach a significant improvement due to the need of multiple data scans.

Lin, Lee and Hsueh propose in [77] three different parallel implementations: Single Pass Counting (SPC), Fixed Passes Counting (FPC) and Dynamic Combined Passescounting (DPC). Those algorithms perform the processing in three steps: the first two steps are almost identical for each algorithm, while in the third step each algorithm adopts different counting strategies.

In the two preliminary phases the algorithms generate the lists $L_1$ and $L_2$ of 1-frequent and 2-frequent itemsets.

In the third phase the SPC algorithm iterates $k$ times (where $k$ is the length of the longest frequent itemset), reading the candidate list generated during the previous step $L_{k-1}$ and computing the new set of candidates $C_k$ with their supports.

FPC instead computes the candidates over multiple steps and counts them in a single stage. In step $k$, the map function reads the list $L_k$ and generates candidates $C_k$, and then $C_{k+1}$ and $C_{k+2}$ starting from $C_k$ and $C_{k+1}$. This technique, however, increases the amount of generated candidates, considering also non-frequent ones causing the overall performance of FPC to be worse than the one of SPC.

DPC is an intermediate solution that dynamically combines candidates among different execution steps using an estimate of the computational effort needed to perform processing.

Solutions similar to SPC were presented, without substantial variations also in [76].

Yahya, Hegazy and Ezat propose MRApriori [137], an algorithm that requires two execution phases\(^8\) operating by applying to each split the traditional algorithm, to reach a minimum local support. The first stage outputs the partial frequent itemsets, while the second computes the global support for them.

Yu Weng, Wang and Jun have recently proposed a version of Apriori [114] based on the use of a boolean matrix. The original dataset is used only once to build the boolean matrix, all subsequent operations are performed directly on the matrix. By the way, the algorithm might have problems in managing in memory the partitions of the matrix generated from large dataset; moreover it has a high communication cost caused by

\(^8\)The algorithm is based on the same idea of the Partition algorithm [114].
2.4. Data Mining and Frequent Itemset Extraction

the distribution phase of the partitions themselves.

The Mahout project includes a parallel implementation of FP-Growth [75] in which is proposed a parallel version of the algorithm that performs the mining process in a set of independent MapReduce job. The first PFP-Growth step splits the dataset in shards, then calculates the global support of all single items that belong to the dataset and computes the list of frequent 1-itemsets \( L_1 \). In the next phase, the algorithm applies standard FP-Growth on each shard and aggregates the final results. PFP-Growth computes the top-\( k \) frequent closed itemsets. A new version of PFP-Growth aimed at improving the distribution of workload across the nodes has been proposed in [142].

The EClaT algorithm is proposed in [140], moving from the analysis of the limitations of other sequential algorithms. It has four key features:

- The use of the vertical representation of the dataset, which associates each itemset to a list of the transactions in which it appears, so the frequent itemset enumeration becomes a simple intersection operation between these lists.

- The search space is partitioned applying the lattice theory. The search space represented on an lattice can be decomposed into smaller parts (called sublattices) analyzed independently.

- The decoupling of the problem of partitioning the lattice respect to the research problem. The research strategies of the itemset frequent proposals are three: bottom-up, top-down and hybrid research.

- The reduced number of readings of the dataset.

The EClaT algorithm performance analysis presented in [140], highlights the robustness of the algorithm with respect to forms of data distortion and the ability to handle frequent itemset of extended dimensions compared to Partition and Apriori algorithms. The algorithm also scale linearly with respect to the number of transactions in the dataset.

The workflow of this algorithm can exploit very well the generic parallelization features provided by the MapReduce programming paradigm, and it is the reason why we have decided to move from its traditional distributed implementation to the new paradigm.

Two different implementations of the EClaT algorithm in MapReduce are proposed in [89]. The idea presented in the BigFIM algorithm is very close to the one depicted in the MREClаТ algorithm proposed in Chapter 6 of this thesis work.
Chapter 2. State of the Art

This two algorithms have similar characteristics as they are both based on three successive phases: a first counting step, a phase of generation of the lists of transactions and a mining phase performed on independent sublattices.

Although, the basic idea is the same, important implementative details distinguish the two algorithms.

distEClaT, the other proposed algorithm, was developed using as base the idea of the Partition algorithm [114]: the algorithm performs the mining on separate nodes producing partial results, performing a final aggregation step combining all the partial results and computing the overall results.

The algorithm performs the computation in three stages:

1. 1-frequent itemset computation
2. k-frequent itemset generation
3. Lattice mining

This approach has a high communication cost due to the necessary intermediate data exchange.
CHAPTER 3

System Architecture Overview

Contents

3.1 SuNDroPS: an Overview ........................................ 36
3.2 Overview of the Original Systems ............................ 40
    3.2.1 Preliminary Notions: the CDT Context Model .......... 40
    3.2.2 The Context-ADDICT System Functionality and Modules .. 43
    3.2.3 Information Flow Processing: PerLa and Tesla/TReX .... 47
    3.2.4 PervAds: a Push Component for broadcasting Context-aware Notices ........................................ 50
3.3 Orchestrating the modules ..................................... 51

This chapter presents the new SuNDroPS system [98] architecture starting from the systems components that are already part of the Context-ADDICT system, on which the new system is based; those components need to be extended in order to implement the new features required by SuNDroPS and also new components, like PerLa, Tesla/TReX, PervAds and MR-Miner have to be integrated.

The integrated system will be composed of:

1. A core component, Context-ADDICT which is in charge of managing the interaction with the user and the contextual operation on the data.
Chapter 3. System Architecture Overview

This component will then act as the main coordinator of the system operation.

2. A module which manages the communications with sensors, i.e. a middleware that allows to work with data streams; besides providing environmental information, the data gathered from the sensors supports the system to infer automatically (part of) the user context. We have studied two different alternatives for this stream management task; they will be shortly presented in Sect. 3.2 and then described in detail in Chapter 4:

- *PerLa* (PERvasive LAnguage) which is a WSNs query language, that provides a middleware for managing WSN.
- *Tesla* and *TReX* which are respectively the rule language and the engine of a CEP system.

3. A pervasive, context-aware information distribution channel called *PervAds* (PERvasive ADvertisement).

4. A Map-Reduce based frequent itemset extraction algorithm, based on the well known *EClat* algorithm [140], *MREClat*, implemented in the *MR-Miner* module.

The architecture of the integrated system will be defined in Sect. 3.1, while the original components currently available implementation will be quickly described in Sect. 3.2; in the same section some extensions will be briefly presented and then formalized in details in the following chapters. Finally, Sect. 3.3 describes how all the SuNDroPS components interacts and cooperate to achieve the desired system behavior.

3.1 SuNDroPS: an Overview

The general – and modular – architecture of the whole system is presented in Fig. 3.1.

While the original Context-ADDICT platform deals only with static data stored in data sources, its new revisitation and extension has to manage dynamic data coming from pervasive sources (sensors and sensor networks, social networks, ...), considering also that these data carry intrinsic temporal information that might be helpful in context management.

---

1 Their respective extensions will be discussed in Chapter 5
3.1. SuNDroPS: an Overview

Figure 3.1: System Module Architecture
Chapter 3. System Architecture Overview

The core of the system is still built upon the core modules of Context-ADDICT, with the insertion of the new components handling pervasive information flows.

Considering the issue of correctly tailoring the user experience, the system needs to understand the current user context considering both environmental and user related data. As we have said, Context-ADDICT needs this context to be explicitly declared by the user or sensed by the client application and then notified to the system (e.g. by using smartphone sensor the app can identify the user position and decide which context to activate, if and only if context switching is position-based). By the way the final challenge we face in SunDRoPS is to allow the system to directly and automatically guess part of the user context. This can be achieved by inferring the user context using real-time data, a feature not yet present in the original Context-ADDICT system since it does not integrate streaming sources but only static ones. However, some data – like user personal interest – that are relevant for the context detection cannot be automatically retrieved or inferred and the user must provide them in order to allow the complete context computation by the system.

Sensor data can thus be used by the Run-Time Context Manager module to control the context switching aspect, gathering and selecting user data of interest grounded on the set of data coming from sensors; hence this allows the system to understand which is the context – among the set of designer provided context – that best fits the user needs in the current scenario of use. Sensors data may also be mixed with static Knowledge Base (KB) data already stored in the system, allowing a richer set of information to be evaluated, increasing evaluation correctness, providing a better experience to the final user.

In this lies the main difference between Context-ADDICT and SunDRoPS: while the first is based on a static and predefined sets of context and each transition must be notified by the user herself (or by an application running on the user device), SunDRoPS instead can (at least partially) automatically determine which is the running context of the user and/or of the system, on the basis of the environmental gathered data.

The problem of seamless switch context is quite on the cutting edge of the technology nowadays. Lots of different services provides context-oriented components (primarily location-based context). The SunDRoPS system empowers this contextualization allowing to use more data other than the location ones, providing a framework that can be adopted in very different scenarios in which a GPS based contextualization of the user may be impossible or at least not sufficient in order to provide a good user ex-
3.1. SuNDroPS: an Overview

perience. Such kind of scenarios could be found both in indoor applications (e.g. museums) where the GPS signal is not available (or at least too inaccurate to discriminate, for instance, from different rooms) or outdoor applications (e.g. emergency situation management) where having all the possible data available can lead to a fast solution of the problem under consideration or of the situation under analysis.

Another problem is how to manage all the historical data about user preferences and past contexts in order to extract information to be used in the future for refining the information to be sent to the user. The Run-Time Preferences Manager module aims at performing this task, aided by the ADaPT framework [88] based on the PREMINE methodologies [87]. The module operate an advanced personalization of the user current context choosing which are the information that best fit the user needs, among the ones selected applying context-aware data tailoring techniques to the original dataset. In order to achieve this result, the module needs to gather both data about the user previous contexts and the log of the activities performed by the user in each context. Analyzing the whole log could become, in complex systems, a hard task since logs can become very long requiring a lot of time to process it. It is therefore necessary to apply data mining techniques in order to shrink the number of data to be analyzed.

It is important to notice that, in order to speed up the analysis process, we can consider only the most frequent operations that a user performs in a context in order to keep track of hers habits; thus applying frequent itemset mining techniques lead to an improvement of the performance of the whole system.

This frequent itemset mining task is eased by MR-miner that, applying the MREClαT algorithm that will be presented in Chap. 6, allows to analyze in parallel (adopting the MapReduce paradigm) the large amount of data flowing to the system, once stored in order to allow its historical analysis.

Looking again at Fig. 3.1, after this brief overview of the new system, it is possible to identify the three new modules:

MR-Miner runs in parallel to the whole system, receiving in input all the data related to the previous users’ contexts and preferences, communicating its outputs directly to the Run-Time Preferences Manager module that considers also those data in dinamically ranking (or discarding) the query answer results to be sent to the user.

PerLa and TReX modules run both as standard query wrappers (receiving a query and translating it to be run on the respective system, sending the results back to the ResultIntegrator module that provides the final
integration with the static data, if necessary) and as “environmental sensing” extensions of the system.

In Sect. 3.3 more details about how the automatic personalization process is performed by the Run-Time Context Manager and the Run-Time Preferences Manager are given.

3.2 Overview of the Original Systems

As said in the chapter introduction, several components are involved in our information management process. In the following these – originally unrelated – systems are summarized, leaving the details about implementation and extensions to the next chapters.

3.2.1 Preliminary Notions: the CDT Context Model

Most of the systems that will be presented later on are based on the concept of “Context”. The model of context chosen is the Context Dimension Tree [26], a hierarchical context model based on nodes from a set \( N = N_D \cup N_V \cup N_{VP} \cup N_{DP} \). Black nodes represent envisaged dimensions \( d_i \in N_D \) for the supported application scenario, i.e., the different perspectives identified by the designer to analyze the context. White nodes represent the values \( v_i \in N_V \) taken by the dimensions; these values can in their turn be analyzed w.r.t. other dimensions, or have value parameters \( p^V_i \in N_{VP} \) attached to them. A value parameter is represented by a white squared node and allows to refer to specific data. Children of a dimension can be values or dimension parameters \( p^D_i \in N_{DP} \). A dimension parameter is represented by a white double-circle node and acts as a shorthand for representing a high number of possible values of the parent dimension; an instance of a dimension parameter is therefore equivalent to a value.

Context elements \( ce_i \) are built starting from CDT nodes; they represent statements of the form dimension=value. In particular a value can also contain a parameter, or can be a dimension parameter. From a formal point of view, a context (also called context instance) \( CI \) is defined as a conjunction of context elements \( CI = \bigwedge_{i=1}^n ce_i \).

The hierarchical nature of the CDT promotes different levels of abstraction. Different context instances express a certain level of abstraction (thus a level of granularity filtering the data) based on the different levels occupied in the tree by the nodes used to build their context elements. The root of the CDT is a value, although a special one, and represents the reference level for context elements filter granularity: a context element built from
3.2. Overview of the Original Systems

the root implies that there is no perspective intended and no filter has to be applied to the data.

Application constraints, preventing the construction of unwanted context instances, are specified by logical formulas, as defined in [28]. Examples are useless-context constraints, used to prevent conjunctions of some context elements from being present in a context instance. Binary useless-context constraints can be graphically represented by means of edges between two white nodes.

Privacy-preserving CDTs  In some scenarios [99, 106], it seems appropriate that the private profile and specific needs and tastes of a user should be unknown to the main system, resting within the user personal device. In this case, the context data have to be distributed to different locations, leading to the introduction of a combined CDT comprising a primary CDT and one or more local CDTs.

A real-world CDT (the one used in the Green Move project presented in Sect. 1.2) providing both an example of primary and local CDT is reported in Fig. 3.2. It represents all the useful dimension for identifying possible contexts in GM. A valid GM context can be defined composing the following dimensions and values (in the following a dimension name will be Typewritten while value names will be emphasized):

- each user has a Role: she can be “admin” or a simple “customer”. Both admins and users are identified by an ID; a customer in addition may provide also a Gender and an Age.

- Some Interest_topic(s) are part of the primary CDT (those not related to private user data); these are the one that are related to aggregation_points and parking_lots. An aggregation point has a given Agg_reg_type and can provide several Agg_reg_services.

- The user can reserve a particular Vehicle_type, e.g. a scooter or a car.

- The four different possible sharing-service configurations presented in Sect. 1.2 (generic, world_of_services, firm and condo) are the allowed values of the Service_type dimension.

- The two dimension Time and Location allow to state context related also to the current temporal and location values.
the Local_conf dimension represent the conjunction point between the primary and the local CDT. The local CDT is rooted in its own root and provides, in this example, two main dimension:

- The General_topic dimension (with the two values traffic and weather) that represents topics those cannot be classified in any other way.
- The Advertisement_topic dimension, containing values tightly connected to the GMAdvisor application that will be presented in Sect. 7.2.3, providing information on which are the topics of interest of the user for personalizing the ads.

A local CDT is maintained locally to user devices and is used to complete the context-based data filtering. For each user, the system will compose a specific combined CDT from the primary one, maintained by the server, and the local one, available on the personal device.

The composition of a local CDT with a primary one must comply with the CDT design constraints described in detail in [25].

A combined context CC of a combined CDT is then easily defined as the conjunction of a context CP of the primary CDT and a context CL of a local CDT, and thus it is nothing more than a conjunction of their context elements (ce):
3.2. Overview of the Original Systems

\[ CC = CP \land CL = \bigwedge_{i=1}^{n} ce_{i} \land \bigwedge_{h=1}^{m} ce_{h} = \bigwedge_{k=1}^{n+m} ce_{k} \]

3.2.2 The Context-ADDICT System Functionality and Modules

The Context-ADDICT system provides an easy, ontology-mediated way to effortlessly integrate different and heterogeneous data sources and allow the user to interact with them as if she interacts with a single, global data source [53].

This integration is achieved using  

i) a Domain Ontology which describes the application domain interesting aspects,  

ii) several ontologies that describe each source both from the point-of-view of the source structure and of the data contained in it,  

iii) some Mapping Ontologies, one for each source [27] which map concepts in the domain ontology to their representation in the sources and  

iv) ad-hoc extractors and query engines for each different kind of source (RDBMS, XML, RDFS/OWL, HTML, …); these are the components in charge of both extracting the data structure to allow the mappings and to extract the data executing the specific part of the global query on its related source.

In order to correctly perform its tasks, the system goes through an initial design and setup phase in which the designer defines the domain ontology, extracts the structural ontology from each source (if required; e.g. RDF/OWL sources do not need this step since they can be directly mapped to the domain ontology) and maps the domain ontology concepts to their corresponding ones in the sources’ ontologies.

Alternatively to the manual mapping drawing made by the designer at design time, an automatic mapping generator module is available in the Context-ADDICT system. The module, named X-SOM (eXtensible Smart Ontology Mapper), gets in input two ontologies (the domain ontology and one of the source ontology) and generates a consistent mapping between them.

The automatic ontology matching is performed in three phases:

**Matching** in which a set of different modules tries to match the ontologies comparing both their syntax and semantics. The output of this step is a set of similarity maps that is forwarded to the next step

**Mapping** Gathering the similarity maps produced in the previous step, the mapping module weight all the alternatives by means of a neural-network-based algorithm and, applying some thresholds, choose the
best matchings, producing some candidate mappings

**Inconsistency resolution** module solve semantic inconsistencies in the final model by means of reasoning and heuristics; in this resolution process the reasoner may insert new concepts in the mapping ontology in order to solve inconsistencies.

Once all the mappings have been defined, it is possible to start the querying process: the user issues a query to the system referring to the domain ontology; the system will then rewrite the original query and split it among the sources containing the data required by the query and consistent with the current context.

Once the results have been retrieved from the sources the system collects and integrates them (some of the sources may give only partial results, that may be joined with results coming from other sources) to obtain the final result of the query to be sent to the user.

Query processing time may be reduced by accepting to lower the confidence of the results: a sources catalog, containing information of partial – or total – sources overlaps (with the related percentage of overlaps) is embedded in the system and if the query optimization is active this information could be used to discard the execution of a query on some sources.

The system generates in most cases several rewrites for a source, each one with a different granularity (some rewrite retrieves more general information, e.g. every user, or more specific one, e.g. every user with surname “Franklin”). It depends on the requirements of the specific application which one of the two rewriting forms must be chosen: if the requirement is to retrieve high confidence results, the generic form must be chosen, increasing the system response time (due to the highest quantity of data retrieved and possibly integrated by the system), while in case of low response time requirements, the specific form must be chosen (despite a possible loss of valid results).

The original Context-ADDICT integration framework, shown in Fig. 3.3 is composed by five modules, performing the main tasks for which the system is built:

### 3.2.2.1 On-the-fly query wrapping

Context-ADDICT provides several different adapters and wrappers for querying different kinds of data sources (RDBMS, XML, RDF, ...). In particular a full-fledged implementation is available for RDBMS data sources in two different modules:
3.2. Overview of the Original Systems

*Figure 3.3: Interaction between the original Context-ADDICT system modules*

- **Relational to Ontology Semantic eXtractor (ROSeX)** makes it possible to introduce relational sources in the systems; it extracts three ontologies from each relational data source: *a*) one representing the structure, *b*) one representing the data (translated into concepts’ instances) and *c*) one which maps the other two. The ontology representing the structure can then be mapped with the domain ontology, allowing the processing of the queries on it.

- **SPARQLEExplorer** is the module translating SPARQL queries into SQL queries, while other modules exist to query different types of sources (XML, csv, ...); obviously on RDF/RDFS/OWL sources SPARQL queries can be executed directly without any sake of translation. After having executed the query, this module outputs the results in the datasource ontology format, making possible their integration at the framework level, by means of the mappings between each datasource ontology and the domain ontology.

The two modules are shown in Fig. 3.3 as the links between *DSO_I* and *Mapping I* (ROSeX) and between the *RDBMS* and block 6 (SPARQLEExplorer).
3.2.2.2 Data integration

Once the sources have been integrated into the system, the platform must provide a unique entry point for user requests; it is up to the system then to dispatch those requests to the appropriate sources and integrate their results in a single, final result to be sent to the user. The modules involved in this process are

- **Rewriting SPARQL (RewSPARQL)** is the framework core module since it is the one that, by means of the mappings, translates the original query (issued to the domain ontology) in the set of queries to be sent to the sources; it also uses the *query catalog* to drop useless queries, if necessary. In Fig. 3.3 is represented by blocks 2 and 4.

- **SPARQL dispatcher (SPARQLdisp)** [97] is the module that takes the output of RewSPARQL and produces a new output in a format that can be processed by each source, reported as block 5 in Fig. 3.3. As an example consider the case of relational sources, in which the output of RewSPARQL must be cleaned in order to be executed on a single source (the sources do not know each other and they cannot interoperate directly, without the help of the framework).

- **ResultIntegrator** [97] is the module (block 6 in Fig. 3.3) responsible for integrating the results coming from the sources. The process simply implies that it gathers all the results in a domain ontology compatible structure and then re-run the initial domain query on the top of it.

To gain better performance the system follows well known database practices like pushing selections and projections down to the sources, reducing the amount of data to be processed later on by the ResultIntegrator module.

The SunDRoPS system will add new modules and features to Context-ADDICT as we will see in Sect. 3.1. In particular, two new modules (*PerLa* and *TRex*) will be inserted in the system, together with all the necessary query wrappers and result collectors/integrators.

3.2.2.3 Mapping Contexts to Data

In Context-ADDICT, in order for the query result to be context-aware, at design time the CDT model is applied by the *context designer* to the *domain ontology* in order to define how it has to be “tailored” in each specific context. Once the designer defines a context on the CDT as a conjunction
of context elements (i.e. different dimension values), this has to be mapped to different concepts in the domain ontology according to the methodology defined in [93]. This simply means that, when the user is in a given context and issues a SPARQL query towards the domain ontology, only the query results belonging to the ontology portion related to the given context will be retrieved. Using a smaller ontology allows the system to produce the query rewriting in less time than the time necessary to satisfy the same request on the global ontology; for the same reason it also speeds up the integration result step. Globally the system will ensure lower response time providing a more focused, and thus better user experience.

While this context-aware process is performed at the central system level, it can happen that some context filtering has to be performed locally (e.g. by the user personal device) due to privacy reasons. When this happens, the context instance is usually split into a primary and a local context instance: the instance tailoring on the primary context can be performed at central system level, while the local context instance tailoring must be processed locally, on the subset of the data forwarded to the user device by the central system filtered by means of the primary CDT (thus reducing the number of interesting answers to be sent to the user client).

Once a client device receives the answer from the system, it can either:

- discard the answers that do not match the local context instance, or
- rank them according to the local context instance, giving priority to the ones that better match it.

In the first case some results are discarded and thrown away, while in the second one a simple ordering is performed and less interesting results are moved to the bottom of the resultset while most interesting ones stay on the top of it.

### 3.2.3 Information Flow Processing: PerLa and Tesla/TRex

The integration of PerLa and Tesla/TRex in the SuNDroPS system makes it possible to retrieve data from sensors and integrate them in the answers; it will also allow the system to gather data independently of the users, directly sensing also the environment, increasing the set of data that can be considered in order to infer the current user – and/or system – context.

In addition they could be used also by the system to sense changes in the environment; once a change is detected a context switch may be triggered (considering the context-aware PerLa, the switch could be triggered directly from the module, otherwise it should be managed from the core
system). *Context-aware PerLa* easily allow the management of the context directly on its middleware, providing that the CDT and the context instances are described also in *PerLa CL* (see Sect. 4.1.3 for further details). Otherwise each request issued toward one of this two modules is translated in *PerLa QL*\(^2\) or *Tesla*\(^3\) by an appropriate on-the-fly query wrapper and integrated in the final resultset by ResultIntegrator.

From the implementation point of view, all the system modules are implemented in Java except the TReX engine that is implemented in C++. By the way, TReX has a Java client library which makes it able to interact with Java applications reducing the efforts needed for its integration in the Sun-DRoPS system.

### 3.2.3.1 PerLa

PerLa is a middleware and query language for wireless sensors networks. It allows:

1. Inserting and configuring the sensors dynamically and “on the fly”.
2. Setting the sensors working parameters and which data must be retrieved.
3. Defining how to group and aggregate the rough data and some intermediate operations on them.
4. Managing the final information contained in data streams.
5. Declaring different contexts, how to compose and activate them, and set the actions to be performed according to their changes.

**Extending PerLa:** To be integrated into the SunDRoPS system, PerLa needs several enhancements, mainly from the middleware point-of-view.

A fully distributed version of PerLa is necessary to gain better performances in data processing: currently the platform only provides a central server and the sensor client and there is no way to exploit the computing power of any of the base stations placed on the network path between a sensor and the central server (e.g. a gateway node that may have more computing power and less power consumption problems than sensors). Computing partial data on those nodes can reduce the computing power necessary at the central server level, speeding up the processing time.

---

\(^2\)Presented in Sect. 4.1.

\(^3\)Presented in Sect. 4.2.
3.2. Overview of the Original Systems

Another needed extension of PerLa is toward a complete integration with other existent context-aware programming paradigms. Among the different available paradigms, Context Oriented Programming (COP), Aspect Oriented Programming (AOP) and Behavioral Programming (BP), the choice is to allow a total integration between PerLa and the COP paradigm, which is one of the most complete and most similar to the idea of context already present in PerLa. The extension of PerLa with COP-handling functionalities allows the programmer to obtain a complete control on the system, both on data and on functions.

Moreover, a complete reengineering of the whole system implementation is being performed since the system has been developed during years by many different peoples and many components had to be revised due to the lack of some features or to some redundancy in the implementation.

3.2.3.2 Tesla/TRex

TRex [51] is a rule-based processing engine, based on the Tesla logic whose semantic is expressed using the TRIO logic [64]. Events are represented by packets with an integer event identifier and some key-value pairs reporting values of some fields of interest (e.g. the event RoomCondition may have event\textsubscript{id} = 3 and 2 attributes, temperature and humidity).

The rules define how to process the data coming in simple packets and how to output them in useful way (e.g. the rule HeatMonitoring may state that if the temperature of a room is higher than 30 degrees a new event HeatAlarm with event\textsubscript{id} = 30 has to be created with an attribute reporting the temperature value).

Extending Tesla/TRex: Tesla and its engine, TRex, comply with the complex event processing paradigm. One of the main issues of the TRex engine is related to the unfeasibility of having a persistent storage, storing static information that may be useful to be considered during the event processing.

These kind of system need to be temporally bounded since they could not check for infinite sequences in the stream, making impossible to deal with the need of static information during processing, without the aid of any external static data repository directly handled by clients. Static information processing currently requires that some component periodically “repeat” those data sending an appropriate event to the system.

This solution, though it might work in very simple scenarios, is not elegant and rather inefficient. A connection with a static repository is needed in order to achieve the possibility to enrich the events with static data, stored
Chapter 3. System Architecture Overview

in a Knowledge Base (KB).

The choice is to use a static RDF/RDFS KB, containing all those concepts and properties useful in the domain of application in which the system is being used.

As we will see in Sect. 5.3, the use of a static, disk-stored KB rises some problems – first and foremost the increment of response time due to disk access – that have to be solved in order to keep data processing almost real-time, trying to avoid to waste time where and when possible.

The KB TRex extension will also allow a two-way interaction between TRex and the Context-ADDICT system since they can “see” each other as sources of information (i.e. the RDF part of a Tesla rule can be issued towards Context-ADDICT, retrieving the integrated results from the system). By the way, this kind of cooperation presents some restrictions necessary to avoid dangerous infinite loops in the interaction between these two modules.

3.2.4 PervAds: a Push Component for broadcasting Context-aware Notices

PervAds is a module for distributing advertisements while keeping the user personal interest undisclosed on its own device. From its original implementation proposed in [36], PervAds has evolved into a more complex system as presented in [99]. In the following all the references to PervAds are related to its last, more complete and customizable, implementation.

PervAds is based on the concept of “ad”, a simple object which is composed of a short caption, an optional image and some metadata (in XML format) that describe the category of interest of the ad. Each ad is also geolocated: its \(< latitude, longitude >\) position is stored among its attributes, so the server can perform a location-based pre-filtering step instead of pushing a very large set of ads to the client, which may not be able to handle them.

The architecture of this module is composed by a Central Server that stores the whole set of ads and a client that retrieves those ads using a REST API.

The ads distribution process comprises three steps:

1. on the central server the system performs a pre-filtering step of interesting messages for the client using the part of context obtained from the primary CDT (e.g. age, gender, time and distance among client GPS position and ad/message geolocalized descriptor);
3.3. Orchestrating the modules

2. The set of pre-filtered messages is sent to the client (e.g., user’s personal device), which performs the filtering step – the private part of the matching – using configured interest topics (local CDT context).

3. Finally, the client displays only the subset of the received messages matching the local CDT criteria: overall, the information has been filtered according to the combined CDT.

3.3 Orchestrating the modules

As it is possible to deduce from the previous system description, the system execution flows can be split into two main sub-flows:

- A **pull-based** flow in which the system answers the users’ specific requests
- A **push-based** flow in which the system provides information to the user only evaluating the context, without having received any specific request or query.

In the first case, the system simply acts as the original Context-ADDICT system does, waiting for the user to send a request and providing the answer to that request. The basic difference is that SuNDroPS can query also streams by means of the PerLa/TReX module.

In the second case instead, the system continuously uses and analyzes the information flowing through the sensors and checks if it is necessary to send new information to the user. It can happen in three ways:

- The most simple way is that the system contains a subscription to a particular information class explicitly inserted by a user (like in pure Pub/Sub systems).

- The designer may have described different operating contexts of the system, depending both on environmental information and on user-related information (both kinds of information could be automatically retrieved or pushed by the user or other actors); if the context changes, an alert may be sent to the users to notify the context change (e.g., from a safe context to a dangerous one)

- An external trigger can cause a context switch, including some alert to the users (some context switch may not be detected by the sensors, requiring an external operator to impose the switch, maybe appending some kind of information in an alert).
Chapter 3. System Architecture Overview

The two latter behaviors are supported also exploiting the power of the PervAdS framework, that provides all the necessary structures and facilities to enact and optimize the context-aware push behavior of the system.

In order to obtain a correct behavior of the system in the push-based working mode, each submodule must interact with other submodules by means of a predefined protocol.

First of all, starting from the definition of the different context, the designer must define a set of monitoring directives to be sent to the PerLa/TRex module, explicitly saying which are the parameters to be monitored for each context, how frequently and what happens once a context switch occurs. If the sensors are monitored by the PerLa system, those directives could be described in PerLa CL, otherwise an ad-hoc TRex client must be implemented by the designer to keep monitoring the desired environmental variables.

The PerLa and TRex modules can then act in two different ways: they can act as masters or as slaves. Acting as a master, once a module detects a context change it simply notifies directly to the Run Time Context Manager module to change its active context and to propagate the change to all the interested modules (and users). If it acts as a slave, the module simply gathers the information from the sensors, sending them to the Run Time Context Manager module that is then responsible for detecting context changes and of the successive propagation phase.

All these operations are controlled by standard communication protocols provided by means of common, standard interfaces shared between the involved modules of the system.

The main standard commands involved are the following:

**PushCtxChange(Context context, Conditions condition)** Pushes a context change to all those modules or users that meet the conditions described in the **cond** attribute (those conditions could both be on user ids or other elements, like position or other sensed values). This command is used both from the PerLa/TrEx module working in *master* mode and from the Run-Time Context Manager module to propagate the context switching information.

**SendData(Data sensor data)** This command sends the sensor data gathered from the sensors to the Run-Time Context Manager module, which then evaluate which are the necessary operations to be performed.

As it is possible to understand from the former descriptions, all the modules operate using a push paradigm (there is no module pulling information

---

4See Sect. 4.1.3.
3.3. Orchestrating the modules

from other ones, every module listens for new data or commands); this
requires a bit more bandwidth consumption (since the communication chan-
nels among modules need to be kept open, if they run on separate hardware
as it can happens, at least with users’ devices). By the way this ensures a
more reactive behavior of the system.

The only exception is the PerLa/TRex module, that allows a pull ap-
proach in some particular cases, to retrieve data from the sensors.

As an example of how the modules collaborate together, consider the
scenario in which a traffic jam is detected in a specific zone of Milan, de-
tected by traffic light sensors. Suppose that a traffic jam is detected when
more than ten different GM vehicles move at very low speed (less than 2
km/h) around the same point (with a given radius, e.g. 750 meters). Con-
sidering the PerLa master behavior, a context TrafficJam will be defined
directly in PerLa, identified by the aggregate count of the vehicles in a
given zone (within the radius of 750 meters) moving at a speed less than 2
km/h. Once the count becomes greater than ten, the traffic light sensor that
has detected the jam switches its context into TrafficJam, notifying it to the
upper layers that will propagate the signal to all the interested sensors (e.g.
traffic light sensors but also to the GEBs of the vehicles in the interested
zone) that will then react trying to solve the situation (the GEBs navigator
will propose alternative paths while traffic lights may change the red/green
light time accordingly to the situation).

The same problem can be managed using Perla/TRex slave behavior
but in this case the module responsible for the necessary context switch
detection will be the Run-Time Context Manager. However the result of the
context detection and switch will be the same as in the previous paragraph.

In the following chapters of this thesis the focus will be on the system
components, their implementation and their extension rather than on the
full system. A complete final theoretical evaluation of the new architecture
described in this chapter will be carried out in Chapter 8, after a complete
view of the new SuNDroPS system components (PerLa, TRex and MR-
Miner modules) and of some of their applications has been given.

---

5 Respectively in TRex some rules, equivalent to the PerLa query described above, have to be defined.
PerLa and Tesla/TRex: a Comparison Between Different Stream Processing Paradigms

Contents

4.1 The PerLa Language and Middleware ........................................ 56
  4.1.1 Data Structures ......................................................... 57
  4.1.2 Language Features ..................................................... 58
  4.1.3 Context-aware PerLa .................................................. 59
  4.1.4 A Stress-test for PerLa ............................................... 67
4.2 Tesla/TRex .............................................................................. 68
  4.2.1 Data Structures ........................................................... 68
  4.2.2 Language Features ....................................................... 69
  4.2.3 Detection Algorithms ................................................. 70
4.3 Comparison of the Two Systems .............................................. 73
  4.3.1 Examples ..................................................................... 73
  4.3.2 Common Issues of the Systems ..................................... 84

PERLA and Tesla/TRex represents the two main different paradigms of information flow processing: DSMS and CEP. This chapter dis-
Chapter 4. Original PerLa and Tesla/TRex

cusses a closer look to those system, moving from the brief descriptions given in Sect. 2.2 and Sect. 3.2. For both the systems languages, data structures and other common features are described. Sect. 4.1, after the general PerLa presentation, gives an introduction to PerLa for Context; the section ends presenting some quick result about PerLa stress tests. Sect. 4.2 presents a detailed view on Tesla/TRex, exploiting the algorithms used for rule detection. At the end of the chapter (4.3), a brief comparison between the two system is presented, trying to define some of the standard request of the Green Move system in both the languages, pointing out differences and similarities.

4.1 The PerLa Language and Middleware

As stated in Section 2.2.1, PerLa is a middleware and query language that allows to operate on WSNs.

In details the middleware provides an easy, XML-based mechanism which allows fast integration of new sensors in the system; the designer has only to provide a sensor description by means of a standard XML syntax. When the system detects the new sensor descriptor, the parser produces automatically the software implementation of the sensor and the sensor will be instantly integrated, dispatching its own sensed data as required by the queries.

PerLa Query Language is an extension of the standard SQL language, including clauses that allows to define where each query has to be run, depending on their abstraction on the physical sensors.

Being a database-like language, the queries’ results are automatically stored in a relational database, to provide a history of data. Until now, PerLa bases itself on an existing DBMS (MySQL).

As shown in Figure 4.1, there are two main runtime layers in the middleware: low level support layer, which provide standard API to handle all the different devices that compose the pervasive system; and high level support layer, that provides query execution services and data management.

PerLa uses the Channel Manger to create a channel virtualization, that allows to establish a bidirectional point-to-point channel between the FPC and the device; this channel can simulate plug & play connection, auto-recognizing new devices, when inserted. FPCs are created by FPC Factory, which receives the XML file containing the details about the device and creates a FPC adapter for the node. The FPC registry, on the other hand, has the list of all the known devices and their relative FPC.

PerLa’s interface is composed by the Query Parser, that receives the
queries directly from the user (to whom reports the execution results), and sends them to the Query Analyzer after having verified their syntax and validated their semantic. The Query Analyzer is responsible for the creation of the Low Level Query (LLQ) Executor, that manages the communication with physical layer, and the High Level Query (HLQ) Executor, that is implemented as a wrapper for external data management systems, and permits the user to perform data management operations. A LLQ Executor is created for each FPC involved, configured for the execution of the statement submitted. FPCs are then consulted for getting data from the physical devices, and results are pushed all together in an output stream.

When nodes are connected to a shared broadcast bus – as in real applications – the channel must be multiplexed: an Adapter Server is added to the middleware to perform (de)multiplexing operations on the FPC side, while on the other side of the connection, an Adapter Client manages the communication channel.

### 4.1.1 Data Structures

There are two different main data structures in PerLa, *Streams* and *Snapshots*.

*Streams* are unbounded tables, designed to be mostly output structures
while *snapshots* are sets of records that are generated during a specified time period. For both of the structures, each record has an ID, a type and a timestamp. Both streams and snapshots allow read and insertion operation.

FPC have different attributes, classified in three main types:

**static attributes** which represent constraints and known values that describe the state of the nodes;

**dynamic probing attributes** which have values that have to be refreshed, directly from devices;

**dynamic non-probing attributes** that read their values from local cache or storages: these are usually combined with notifications that communicate that their value has changed.

In order to make possible a complete knowledge of the events and attributes used, the interface must provide three kinds of functionalities:

- retrieving attributes values, assigning a unique ID to each of them;
- firing notification events;
- getting the list of all supported attributes.

Sensors produce continuous streams of data, which flow from the sensors node to the consumer node. A process of data aggregation, where information is gathered and expressed in a summary form, is needed both for performance and energy-saving reasons.

Data reduction could be based on spatial aggregation or on temporal aggregation. The sampling could be made by means of time windows, or it could be event based, starting when the logical object event is fired.

### 4.1.2 Language Features

PerLa’s data representation hides physical devices as much as possible, providing a database view of the whole pervasive system.

Language statements should allow the user to specify sampling parameters and the set of operations to manipulate raw data. On the other hand, non-functional fields are expressed in an abstract way, and then they are internally translated into concrete values.

PerLa provides three different types of queries:

- **HLQ, High Level Queries** run on the top of the system, at the central level; those queries generally describe how to aggregate information coming from different sensors.
4.1. The PerLa Language and Middleware

- LLQ, *Low Level Queries* run directly on the sensors, describing which data must each sensor retrieve and stream, how frequently and if these data has to be aggregated directly by the sensor itself.

- AQ *Actuation Queries* run on sensor that have also capabilities of performing some kind of action; the content of those query is simply a set of commands that activate some kind of actuator connected to the sensor (e.g. some kind of small electric engine).

**Low Level Queries**  
Low level statements need to have a local buffer and should retrieve data from sensors, pushing into the buffer and, then, taking values from the buffer and inserting obtained records into output data structures. To make this possible without saturating the buffer memory, the executor needs a garbage collector to clean used data.

Low level conditions include sampling statements (sampling ... where ...), that specify how to collect data. It could be time based, with a frequency set by the user, or event based, which means that operations are executed upon the occurrence of an event. In order to manage data, otherwise, there is the select clause, which retrieves data from sources that are not only tables, but also from a group of sensing devices. Select can be time- or event-based. Low level select is different from SQL's one, because it needs also a time parameter. It could be associated with a “having”, that works with both aggregates and single values, or a “count” statement.

Selecting is followed by the execute condition, “execute if”, that allows a node, with a predicate, to be part in the query, or not. Finally, PerLa allows to short the lifetime of a low level query, forcing its termination, with a “terminate after” statement.

**High Level Queries**  
High level queries take one or more streams, or snapshots, as input forcing them in an output structure, after a filtering operation (given from the user’s query).

High level language is a natural SQL's derivate, extended with some activation conditions, like the “every” clause, that specifies the execution restrictions, for example defining how often the query has to be executed; “from”, that defines a duration window, or “refresh”, to update collected data.

4.1.3 Context-aware PerLa

A context-aware extension of PerLa has been designed during last years, based on the CDT model presented in Sect. 3.2. The PerLa Context Lan-
guage (PerLa CL) allows to describe directly in PerLa QL both the CDT and the context instances of interest, leaving to the PerLa middleware itself the issues related to context switching. In this way the context is directly managed by the middleware and the programmer has the only role of defining each context enabling/disabling condition and which are the operations to be performed in each context.

CDT Declaration  This part allows the user to specify the CDT, i.e. all the application-relevant dimensions and values they can assume. As an example of its usage we report the syntax to define completely the CDT presented in Fig. 4.2. The CDT defines four dimensions: Location, that can assume 'office' and 'meeting_room' values; Smoke, Env_Temp and Env_Humidity, that represent the actual condition of the environment. Notice that, while Smoke and Env_Temp are quantified by symbolic observables, Env_Humidity values are directly used as numerical observables provided by sensor readings.

![Context Dimension Tree (CDT)](image)

Figure 4.2: Context Dimension Tree (CDT)

In Listing 4.1 labels of the fields in the WHEN clause (e.g. location, smoke, humidity, ...) refer to external input data (like data coming from the sensors or other kind of input) while the labels in the CREATE DIMENSION or CREATE CONCEPT clauses refer to the labels of the CDT nodes.

A set of CREATE DIMENSION/CONCEPT statements allows to declare the dimensions as well as their concepts nodes. When creating a concept of a dimension, the designer must specify the name and the condition for assuming the specified values by means of numeric observables that can be measured from the environment (WHEN clause). When, instead, the design requires the presence of attributes the CREATE ATTRIBUTE clause must be used, using the $ sign as a prefix before the name of the attribute, meaning that its value will be supplied by the application at runtime.
4.1. The PerLa Language and Middleware

CREATE DIMENSION Location
CREATE CONCEPT office
  WHEN location = 'office'
CREATE CONCEPT meeting_room
  WHEN location = 'meeting_room'
CREATE DIMENSION Smoke
CREATE CONCEPT none
  WHEN smoke < 0.4
CREATE CONCEPT little
  WHEN smoke >= 0.4 AND smoke <= 1
CREATE CONCEPT persistent
  WHEN smoke > 1
CREATE DIMENSION Env_Temp
CREATE CONCEPT cold
  WHEN temperature < 18
CREATE CONCEPT mild
  WHEN temperature >= 18 AND temperature < 24
CREATE CONCEPT hot
  WHEN temperature >= 24
CREATE DIMENSION Humidity
CREATE CONCEPT h_level
  CREATE ATTRIBUTE $h_value

Listing 4.1: The Context Dimension Tree

Context Creation

This part of the syntax allows the designer to declare a context on a defined CDT and control its activation by defining a **contextual block**, which is composed by four fundamental parts, called **components**:

- **ACTIVATION component**: allows the designer to declare a context, using the `CREATE CONTEXT` clause and associating a name to it. The `ACTIVE IF` statement is used to translate the Context $= \bigwedge_{i,j} (Dimension_j = Value_i)$ statement into PerLa.

- **ENABLE component**: introduced by the `ON ENABLE` clause, allows to express the actions that must be performed when a context is recognized as active;

- **DISABLE component**: introduced by the `ON DISABLE` clause is the counterpart of the previous one, allowing to choose the actions, if any, to be performed when the declared context is no more active;

- **REFRESH component**: instructs the middleware on how often the necessary controls must be performed.

In Listing 4.2 we report a declaration block for two possible contexts: the first one represents the “normal” situation, in which the environment has
Chapter 4. Original PerLa and Tesla/TRex

CREATE CONTEXT normal
ACTIVE IF Temperature = ‘mild’ AND Humidity.h_level >= 40
AND Humidity.h_level <= 65
AND Smoke = ’none’

ON ENABLE:
SELECT humidity, temperature
SAMPLING EVERY 1m
EXECUTE IF location = ’office’
ON DISABLE:
DROP normal
REFRESH EVERY 5m

CREATE CONTEXT fire
ACTIVE IF Temperature = ’hot’ AND Smoke = ’persistent’

ON ENABLE:
SET alarm = TRUE
ON DISABLE:
DROP fire
SET alarm = FALSE
REFRESH EVERY 5m

Listing 4.2: Examples of contexts in PerLa

comfortable values of temperature and humidity, while the second context represents the rise of a possible dangerous situation (a fire alarm).

Contextual Block Composition

In the previous sections we mentioned how a growing tree depth of the CDT allows the designer to capture the aspects of the environment with different granularities, since more dimensions (and thus concepts) allow to express more possible contexts. In particular the total number of contexts grows exponentially with the number of concept nodes, and, even for middle-sized CDTs, the task of declaring all the contexts using the aforementioned syntax becomes rapidly unfeasible for the designer. As an example from a CDT with 5 dimensions with 3 concepts nodes each, even with many constraints, more than 500 different meaningful contexts can be generated, charging the designer with the hard task of declaring i) every single context and ii) a set of actions for each one of them. In the following we show the possibility of relieving the designer from this arduous task, enabling the middleware to automatically build the contextual block starting from the contextual block components, called partials [118].
Partial components  The syntax of the PerLa language allows to separate the block components into one or more partials, as shown in Figure 4.3. A Partial contains a subset of the statements and clauses included in the original block, with the only constraint that this subset must be valid from the point of view of the PerLa QL syntax. This division is particularly meaningful for the ENABLE and DISABLE components, while the only block that cannot be divided is the ACTIVATION block since it deals with the definition of context itself. The only requirements for the partials is that they must be syntactically and semantically correct.

Automatic composition  With the introduction of partials, the concepts behind the automatic composition of contextual blocks can be described. The main idea, already adopted in [28] for the tailoring of data, is illustrated in Figure 4.4. The designer must only associate one or more partials with each context element of the CDT. When the system has to compose a contextual block, it starts from the partials associated with the context elements which are part of the context and combines them by means of a generic operator, represented here by the symbol $\oplus$.

The association of the partials with the CDT context elements can be performed using the following syntax, which enriches the CDT declaration section of the CL.

The WITH ENABLE COMPONENT clause may contain any query expressed using PerLa. The same holds for the WITH DISABLE COMPONENT clause. The last clause (WITH REFRESH COMPONENT) allows to specify the time period (always using PerLa’s syntax) to be used. Finally the composition can be carried out both at design and at run-time [118].

Design-time composition $\otimes_D$  When the association phase is complete and before the system is put into an operational state, it is possible to combinatorially generate all the possible contexts that are defined by the CDT

![Figure 4.3: Partial (ENABLE) components definition](image-url)
and that are not forbidden by the constraints. For each possible context the relative contextual block is then automatically generated composing the partials associated in the previous phase. In Algorithm 1 the composition algorithm is shown in pseudo-code [118].

This algorithm, as its first step, retrieves all the relative context elements (i.e.: the couples \( Dimension_i = Value_j \)) for all possible contexts. (getAttributeElements() function, Algorithm 1 line 4). With the context elements “at hand”, the CM exploits three functions\(^1\) in order to retrieve the partial components associated with every context element retrieved at the previous step (getEnableComponents(), getDisableComponents() and getRefreshComponents() functions, Algorithm 1 line 6 - 8). When all these inputs have been retrieved a composeBlock() function is invoked. This function firstly creates an empty contextual block. All the retrieved partial components are attached (attach() function) to the empty block using the dot notation (to indicate the access to a precise component of a contextual block). As far as the REFRESH component is concerned, the composeBlock() function computes (and attaches) the lowest refresh value

\(^1\)Algorithm 1 reports only one function, the other two being identical from an operational point of view.
4.1. The PerLa Language and Middleware

Input: The $\xi$ set of all possible contexts  
Output: BS set with the composed contextual blocks

1. BS=\emptyset;
2. for (context $c_i \in C$) do
   3. /*Context elements retrieval*/
   4. CE[ ] <- getContextElements ($c_i$);
   5. /*Components retrieval*/
   6. E[ ] <- getEnableComponents (CE[ ]); 
   7. D[ ] <- getDisableComponents (CE[ ]); 
   8. R[ ] <- getRefreshComponents (CE[ ]); 
   9. $B_i$ <- composeBlock (E[ ],D[ ],R[ ]); 
   10. optimizeBlock ($B_i$); 
   11. if (parseBlock ($B_i$) == 'OK') then
      12.      BS = BS $\cup$ [$B_i$];
   end
   else
      13.      return WARNING('Parse Error');
   end
end
return BS;

Procedure composeBlock (E[ ],D[ ],R[ ]): 

19. B = \emptyset;
20. for (enable comp. $e \in E[ ]$, disable comp. $d \in D[ ]$) do 
   21.      attach (B,E, $e$);
   22.      attach (B,D, $d$);
end
25. B.R = min(R[ ]); 
return B;

/*Identical for Disable and Refresh*/;

Procedure getEnableComponents (CE[ ]); 

28. EB = \emptyset;
29. i = 0;
30. for (ce $\in$ CE) do 
   31.      if (Context_Enable_Rel(ce) != \emptyset) then
      32.          EB[i] = Context_Enable_Rel(ce);
      33.          i++;
   end
end
return EB;

Algorithm 1: Composition Algorithm Pseudocode
Chapter 4. Original PerLa and Tesla/TRex

among the ones contained in the R[ ] set. It seems reasonable, in fact, that the context whose state must be controlled with a higher frequency (smallest temporal values) is the most critical one and its refresh value is to be chosen during composition. Except for the discussed 

refresh component, the enable and disable components are formed by multiple clauses expressed using PerLa syntax: simply appending, one after other, all the clauses contained in this components is not enough. An optimizeBlock() function is in charge, acting on the composed enable and disable components, of placing every single PerLa clause in the right order and position according to the specific query syntax, described in [119]. Further optimizations can be achieved, like merging components related to the same clause in a single one, also if they refer to different parameters. The last step of the algorithm instructs the CM to inject the composed contextual block into the middleware QueryParser component using the parseBlock() function. The QueryParser is, in fact, able to verify the syntactic and semantic validation of the composed block and to raise a warning message in case some inconsistencies are detected.

Run-time vs. design-time composition   In the run-time approach \( \oplus_R \), the composition of a contextual block is carried out at run-time only when its relative context is recognized as active by the middleware. The two proposed approaches show a trade-off that can be exploited in different situations. At design-time, in fact, the designer has total control over each composed block before its behavior is enacted; as a consequence, he or she can still modify the composed blocks in case the requirements ask for particular attention. On the other side, many contextual blocks will possibly be generated and controlled by the designer even if their actual activation happens very seldom, because this more static vision of the whole system considers all the contexts at the same level of plausibility. At run-time the system behaves with a more autonomic fashion, but the designer cannot modify the composed contextual blocks. In addition, performance must be kept under control: more than one context can, in fact, be active simultaneously, and also the context switches may be very frequent. The frequent on-line composition of several context elements, possibly involving complex partials, could then potentially slow down the whole system performances.
4.1.4 A Stress-test for PerLa

The PerLa system is still under development, being born in academic environment, in order to improve its performance and its offered services. Using a dataset provided for the “ACM DEBS 2013 Grand Challenge” \(^2\), some performance test have been performed on PerLa.

The challenge is about a soccer game with 7 players and of the duration of around an hour; all the players, and the ball, are equipped with sensors that continuously monitor their position, speed and acceleration.

Data are sampled with high frequency: 200Hz for players’ sensors and 2000Hz for ball’s one. Each player and the referee wear two sensors, one for each leg; the goalkeeper has also two additional sensors on the hands. The event schema format is: 
\[
< \text{sid}, ts, x, y, z, |v|, |a|, v_x, v_y, a_x, a_y, a_z >
\]
where \(\text{sid}\) is the sensor id, \(ts\) a timestamp (in picoseconds), \(x, y, z\) describe the position, \(|v|\) and \(|a|\) are the absolute value of speed and acceleration, correlated with their vector’s direction. It was requested to calculate the ball possession for each of the players, considering the ball as “obtained” if in proximity of the player and hit; and to detect shots directed to goal, defined as the union of goal hit and closely missing-goal strikes.

To perform this task, PerLa needs to be improved. The first problem is related to the codification of the data: MySQL doesn’t provide a value tolerance that allows us to manage timestamps in picoseconds. Leaving apart this issue for a while, among other things, a change in the FPC’s structure becomes necessary, in order to make it congruent to this kind of DB. On the other side, considering the state of development, some data cleaning on the original dataset is performed in a pre-processing step, clearing all those data that are useless for the final results computation, such as incorrect data (specified from the challenge description), balls and players outside the game field and data related to the referee.

Data are then processed in order to be sent to the PerLa system in a valid format.

Through a simple script, these files have been sent to PerLa’s middleware that reads them as a stream of data. Since a decay in PerLa performance has been noticed with an increasing number of active FPC, more tests are required to determine the limits in the number of FPCs that PerLa can concurrently support with acceptable performance.

\(^2\)http://www.orgs.ttu.edu/debs2013/index.php
Chapter 4. Original PerLa and Tesla/TRex

4.2 Tesla/TRex

A different paradigm of stream processing is represented by Complex Event Processing systems.

This kind of systems deals with a stream considering it a sequence of events instead of viewing the stream as an ordered flow of tuples. “Primitive” events flow into the system which processes them by means of appropriate rules emitting new “composite” events, those content is built upon the values contained in primitive events.

In CEP infrastructures [52] we distinguish between event generators (sources) and event consumers (sinks). The former observe primitive events and report about them, while the latter receive event notifications and react to them. Using the nomenclature typical of publish-subscribe systems we say that sources publish event notifications (or simply events) and sinks subscribe to events. The CEP engine sits in the middle with the task of detecting so-called composite events from primitive ones through a set of rules – expressed in an ad-hoc language – that are conceived and deployed by rule managers.

TRex [85] is a CEP engine based on the Tesla rule language proposed for combining language expressiveness and engine efficiency.

Since the CEP paradigm recalls the idea of Publish/Subscribe systems, TRex clients subscribe to events of interest by sending a particular packet (called subscription packet) to the server\(^3\), including the identifier of the event of interest (each client can subscribe to multiple events sending a subscription request for each one of them). The server will then forward to each client the events of its interest. Clients can then use the content of the event for their own further processing.

4.2.1 Data Structures

TRex is based on Events. An event has an associated type, which defines number, order, names and types of the attributes that build the notifications, and a timestamp that represents its time of occurrence. Events usually enter in the system in timestamp order. Periodic rule execution is allowed through timers that, in the functional model, are captured by the presence of the clock.

It is up to sources to set the type of the primitive events they will publish, allowing the system to correctly process rules.

---

\(^3\)The server is the core part of the system, the one that runs the TRex engine and performs all the processing.
4.2. Tesla/TRex

CEP paradigm does not provide any method to get, or set, data from any type of storage, so TRex just manages data streams.

4.2.2 Language Features

The language models the complex ordering and timing relationships that join primitive events and the actions required to aggregate them. Rules select relevant notifications according to a set of defined constraints. There could be two different kinds of constraints: one that selects element on the basis of the values they carry, either to choose single notifications or to choose a set of related notifications (we can call this “parameterization”). The other kind of constraints operates on the timing relationships among notifications and they allow capturing sequences of events.

The last event of the sequence (i.e. the ones that occurs at the end of the sequence to be detected) is called terminator. Once the terminator event is detected the whole rule processing is triggered processing the sequence in backward order⁴ (from the last event to first) checking if it satisfies one of the rule deployed in the system.

Tesla semantic is formally specified using TRIO, a first order logic-like language that allows the time “management”. TRIO formulas are correct under the assumption that a set of events can be selected by a given rule only once, providing the uniqueness of selection. So, it is impossible to generate different events sharing the same label. The concept of “label” is to introduce a unique global identifier for each event notification.

Tesla combines rules together in order to create a more complex one, exactly as in first order logic. Furthermore, Tesla is able to express selection, parameterization, negations, aggregates, sequences and iterations.

Some of the execute conditions offered by Tesla are expressed in the “₇-Within” clause, that define single or multiple selection policies as they use any available notification in a given time window. For example, “each-Within” matches every valid event in the given range of time, while “last-Within” matches only the last one in the given time window.

Tesla offers also consumption clause, which allows the user to decide to invalid an events during time.

The general rule structure follows this semantic:

- definition of the structure of the new, composite event (DEFINE);
- sequence of primitive events which need to occur to create the composite event (FROM);

⁴Tesla and TRex work using the backward approach quickly presented in Sect. 2.2.
Chapter 4. Original PerLa and Tesla/TRex

- assignment of the actual values of the attributes to the new, composite
  event ones (WHERE).

```
DEFINE ComplexEvent (Att1 : Type, Att2 : Type, ... , Attn : Type)
FROM Pattern
WHERE Att1 = f1, Att2 = f2, ... , Attn = fn
[CONSUMING e1, e2, ... , en]
```

4.2.3 Detection Algorithms

The TRex engine implements two different pattern detection algorithms:

- **Automata-based Incremental Processing** (AIP), is a detection al-
  gorithm based on automata which stores the state of the detection,
  performed “on-the-fly” and incrementally as new events arrive to the
  server.

- **Column-based Delayed Processing** (CDP), is instead based on stacks
  which stores events when they enter the server (in case constraints are
  expressed in the rule, the event must satisfy them or it is discarded
  immediately). Once a terminator event enter the system, the rule pro-
  cessing starts, checking if exists a sequence of events that satisfy the
  rule.

In the next chapters, every time we mention the TRex detection algo-
 rithm we will refer to the standard, CPU-based implementation of the CDP
 algorithm, while in this introductory section both the algorithm will be pre-
 sented.

A parallel, GPU-based implementation of the CDP algorithm is writ-
 ten in CUDA® to exploit the features of having a parallel architecture and
 gaining advantages in terms of computation time.

In order not to lose events that could happen while the previous are being
 analyzed, TRex uses advanced memory management techniques to avoid
duplicated data shared by multiple sequences.

Considering the **Automata-based Incremental Processing** (AIP) algo-
 rithm, each time a new rule is deployed into the system is processed by
the Rule Manager, who generates its corresponding automaton model that
is passed to the **Automaton Models** component. During processing, the Rule
Manager identifies static constraints, which are associated to the model de-
 fined from the rule and do not depend on the event captured, and uses them
to compile the **Static Index**, which is used to perform a preliminary type

\[\text{http://www.nvidia.com/object/cuda_home_new.html}\]
4.2. Tesla/TRex

![Image of T Rex's architecture](image)

**Figure 4.5:** Schema of T Rex's architecture

and content-based filtering of incoming events. Every new event is put in a FIFO Queue, in order not to lose any of them due to any kind of delay, and events data structures are stored. When the event exits the queue, it could be discarded, if useless, or sent to the Sequences Component. TRex makes further selection within the events collected, with the help of dynamic constraints, as timing constraints or non-valid parameters.

The analysis of the events is deployed accordingly to a set of processing rules, which describe how incoming flows of information have to be processed to timely produce new flows of *composite events* as outputs. The idea is to accumulate primitive events as they arrive, postponing the real process of analysis to when a specific event happens and is processed.

The second – and more efficient – rule processing algorithm is based on pushdown automata, and it is called *Column-based Delayed Processing* (CDP); it stores all events received, until a terminator is found. TRex translates each rule into different stacks, one for every different primitive event indicated in the rule. When a new event $e$ enters the engine, the algorithm extracts all the rules, and all the stacks in each rule, whose type and content constraints are satisfied by $e$, and adds $e$ on the top of them. If among them there is a terminator one, the processing of the events stored so far starts.

Processing is performed stack by stack, from the last to the first one, creating partial sequences of increasing size at each step. First, CDP deletes old events that cannot participate in a valid sequence since they do not satisfy the timing constraints expressed by the rule. After that, CDP starts computing valid sequences. At each step, CDP compares the partial se-
Chapter 4. Original PerLa and Tesla/TRex

sequences computed so far with the events in each column.

When an event satisfies timing and parameter constraints it is used to
create a new partial sequence. Valid sequences are then used to generate
new composite events. If a rule contains an aggregates, TRex follows the
following approach: for each rule \( R \), and for each aggregate \( a \) in \( R \), a dif-
fierent stack is created to store the events matching the type and content
constraints of \( a \).

If a valid sequence for \( R \) is detected, the stack for \( a \) (and those for the
other aggregates of \( R \)) is processed as follows:

- the timestamp of the terminator is used to determine, considering the
windows in \( R \), the events to keep according to pure timing constraints,
discard the others;

- the values and timestamps of the events participating in the sequence
are used to select events in the column according to timing and param-
eter constraints;

- the selected events are then used to compute the value of the desired
aggregate [50].

```python
for each rule in getMatchingRules(e)
    for each column in getMatchingColumn(rule)
        column.add(event)
        if (column.isLast())
            deleteOldEvents(column)
    partialSequences.insert(e)
    sequences = computeSequences(partialSequence, column)
generateCompositeEvents(sequences)

deleteOldEvents(col)
    if (col.isFirst()) return
    col.getPreviousCol().deleteOlderThan(col.getFirstTS — col.getWin())
    deleteOldEvents(col.getPreviousCol())

computeSequences(partSeqs, col)
    if (col.isFirst()) return partSeqs
    previousColumn = column.getPreviousColumn()
    for each p in partSeqs
        for each ev in col.getPreviousCol().getEvents()
            if (p.checkParameters(ev))
                newPartSeqs.insert(createPartSeq(p, ev))
            if (checkSingleSelection()) break
        computeSequences(newPartSeqs, col)
```

**Listing 4.4: The CDP Algorithm**

We can say that the rules result could be managed by piles, if useful for
more complex rules, but there is not any possibility to have any long-term
storage.

72
4.3. Comparison of the Two Systems

When sequences arrive to their accepting state, they are forwarded to the Generator Component, who manages the data, aggregating events or computing values.

Externally, Adapters communicate with the TReX engine. Currently there exist three kinds of adapters: one that allows to communicate with local clients, and other two that make possible the communication with remote clients, written in Java or C++ languages.

4.3 Comparison of the Two Systems

This section presents some examples of queries (or rules) expressed in both languages (if possible), pointing out the pros and cons of each language syntax. From the issues found in this comparison will move the extension effort on PerLa and Tesla/TRex that will be exposed in Chapter 5. All the examples derive from real-world needs highlighted during the Green Move project.

4.3.1 Examples

For each example it will be reported the request of the issue, a brief scheme of the data needed, the code in PerLa language, the code in Tesla language and some comments to describe the differences between the two languages.

Logical objects are described using a table composed of four columns: field name, data type, field type and a short description. The abbreviations reported hereafter are indicated in Table 4.1.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>Static Attribute</td>
<td>S</td>
</tr>
<tr>
<td>Non probing dynamic attribute</td>
<td>NP</td>
</tr>
<tr>
<td>Probing dynamic attribute</td>
<td>P</td>
</tr>
<tr>
<td>Event</td>
<td>E</td>
</tr>
</tbody>
</table>

Table 4.1: Abbreviation for allowed field types of logical objects.

In Figure 4.6 are indicated the definitions of the main events.

Example 1

Check if the car is moving without any valid reservation.

The following information are needed to answer this request:

• speed > 0
Figure 4.6: Events’ schema

- no valid reservation (there has been no pick-up since the last release of the vehicle)

It is also useful to know that TakenOrReleased.takenReleased = 0 means “released” (so the car is free and parked) and TakenOrReleased.takenReleased = 1 means “taken” (the car is being used).

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>P</td>
<td>Speed of the car</td>
</tr>
<tr>
<td>TakenOrReleased</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>date</td>
<td>Integer [3]</td>
<td>NP</td>
<td>Data of when the car got taken / released</td>
</tr>
<tr>
<td>takenReleased</td>
<td>Integer</td>
<td>P</td>
<td>Flag that indicates if the car is taken or released</td>
</tr>
<tr>
<td>takenInCharge_Released</td>
<td>–</td>
<td>E</td>
<td>Notices if the car had been taken or released</td>
</tr>
</tbody>
</table>

Table 4.2: PerLa’s data table for example 1
4.3. Comparison of the Two Systems

**PerLa Low Level Query**

**SNAPSHOT MostRecentUse:** for 10 samplings, every 30 minutes, it lists the most recent use, when a vehicle get taken or released.

```sql
CREATE SNAPSHOT MostRecentUse (greenBox_id String, takenReleased Integer, date [3] Integer)
WITH DURATION 10
AS LOW:
EVERY 30 m
SELECT greenBox_id, takenReleased, date [3]
HAVING date = MAX(date, 10)
UP TO 30m
SAMPLING
ON EVENT takenInCharge_Released
```

**PerLa High Level Query**

CREATE OUTPUT STREAM Theft (greenBox_id String, recentUsage date) AS HIGH:
EVERY 10 m
SELECT greenBox_id, MAX(MostRecentUse.date) as mass
FROM MostRecentUse, TakenOrReleased, VehicleData
WHERE VehicleData.greenBox_id = TakenOrReleased.greenBox_id AND
      TakenOrReleased.date = mass AND TakenOrReleased.takenReleased = 0
      AND VehicleData.speed > 0

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
<tr>
<td>speed</td>
<td>Integer</td>
<td>Speed of the car</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Release</td>
<td></td>
</tr>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
<tr>
<td></td>
<td>Taken</td>
<td></td>
</tr>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
</tbody>
</table>

**Table 4.3:** Tesla’s data table for example 1

**Tesla Rule**

```sql
DEFINE Theft (ID : String)
FROM VehicleData (greenBox_id = $id AND speed > 0) AND
LAST Release (greenBox_id = $id) WITHIN 10day FROM VehicleData AND
NOT Taken (greenBox_id = $id) BETWEEN Release AND VehicleData
WHERE ID = VehicleData.greenBox_id
```

As it is possible to notice from the code excerpts, there is not a great difference between the two languages. The only problem with Tesla language is that you are obliged to put a value in the within clause (10 days in the example) but, doing so, the list will lose all the cars stolen before 10 days.
Chapter 4. Original PerLa and Tesla/TRex

ago (or T [period of time] ago).

Example 2

In this example a check is performed if battery estimated operating time is only slightly above to the time necessary to reach the releasing established point.

The following information must be exposed:

• the state of the charge

• the gps position of the vehicle

If the state of the charge is lower than the needed battery to reach the releasing point indicated in Reservation plus five percentual points (for example), but not lower than the operating time needed to reach the releasing point, then the operating time is lacking.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greanBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>gps_data</td>
<td>GPSData</td>
<td>P</td>
<td>Get the GPS position</td>
</tr>
<tr>
<td>soc</td>
<td>Integer</td>
<td>P</td>
<td>State of charge of the battery</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>placeRelease</td>
<td>GPSData</td>
<td>NP</td>
<td>Where the car have to be released</td>
</tr>
</tbody>
</table>

Table 4.4: PerLa’s data table for example 2

PerLa Low Level Query

SNAPSHOT LastStart: get the data related to the last time the vehicle got turned on.

```
CREATE SNAPSHOT LastStart (greenBox_id String, timeMinutes Integer,
                          GPSData position) WITH DURATION 2h
AS LOW:
  SELECT greenBox_id, timeMinutes, position
  HAVING timeMinutes = MAX (timeMinutes, 2h)
  SAMPLING
    ON EVENT ChangeVehicleStatus
    WHERE ed_status = 1
```
4.3. Comparison of the Two Systems

**SNAPSHOT BatteryAtStart:** remember the state of charge when the vehicle got turned on.

CREATE SNAPSHOT BatteryAtStart (greenBox_id String, batt Integer) WITH DURATION 2h AS LOW:
SELECT greenBox_id, soc
SAMPLING
ON EVENT ChangeVehicleStatus
WHERE ed_status = 0

**SNAPSHOT BatteryConsumption:** calculates the battery consumption from start point to the time it is requested.

CREATE SNAPSHOT BatteryConsumption (greenBox_id String, battConsumed Integer) WITH DURATION 2h AS LOW:
SELECT greenBox_id, battConsumed
HAVING battConsumed = VehicleData.soc - BatteryAtStart.batt
SAMPLING
EVERY 10 m

**PerLa High Level Query**

CREATE OUTPUT SNAPSHOT LowBatteryAtReleasePoint (greenBox_id String) WITH DURATION 2h AS HIGH:
SELECT greenBox_id
FROM VehicleData, Reservation, BatteryConsumption, LastStart
WHERE VehicleData.greenBox_id = Reservation.greenBox_id AND
VehicleData.greenBox_id = LastStart.greenBox_id AND
soc < (BatteryConsumption.batt + (VehicleData.gps_data - Reservation.placeRelease)) * 5 AND
soc >= (BatteryConsumption.batt + (VehicleData.gps_data - Reservation.placeRelease)) * 5
WHERE greenBox_id IS NOT NULL

77
### Table 4.5: Tesla’s data table for example 2

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
<tr>
<td>soc</td>
<td>Integer</td>
<td>State of charge of the battery</td>
</tr>
<tr>
<td>kmCovered</td>
<td>Float</td>
<td>How many km have been covered</td>
</tr>
<tr>
<td>kmTot</td>
<td>Float</td>
<td>How many km from starting point to goal</td>
</tr>
<tr>
<td>kmRemains</td>
<td>Float</td>
<td>How many km are missing</td>
</tr>
<tr>
<td>gps_now</td>
<td>GPSData</td>
<td>Where the car is now</td>
</tr>
<tr>
<td>gps_start</td>
<td>GPSData</td>
<td>Gps of starting place</td>
</tr>
<tr>
<td>gps_goal</td>
<td>GPSData</td>
<td>Gps of goal place</td>
</tr>
<tr>
<td>tTot</td>
<td>Integer</td>
<td>Estimated time to reach the goal from the starting point</td>
</tr>
<tr>
<td>tRim</td>
<td>Integer</td>
<td>Estimated time to reach the goal from now</td>
</tr>
<tr>
<td>bat0</td>
<td>Integer</td>
<td>State of the charge at starting point</td>
</tr>
<tr>
<td>batNow</td>
<td>Integer</td>
<td>State of charge</td>
</tr>
</tbody>
</table>
4.3. Comparison of the Two Systems

**Tesla Rule**

```java
DEFINE Navigator1
(greenBox_id : String, kmRemains : Integer, kmCovered : Integer, 
kmTot : Integer, gps_now : <double, double>, 
gps_start : <double, double>, gps_goal : <double, double>, 
tTot : <Integer, Integer>, tRim : <Integer, Integer>, 
batNow : Integer, bat0 : Integer)
FROM Navigator1(greenBox_id=$id and kmCovered=0) AND
LAST VehicleData(greenBox_id=$id) WITHIN 5min FROM Navigator
WHERE Navigator1.greenBox_id = Navigator.greenBox_id AND
Navigator1.kmRemains = Navigator.kmRemains AND
Navigator1.kmCovered = Navigator.kmCovered AND
Navigator1.kmTot = Navigator.kmTot AND
Navigator1.gps_now = Navigator.gps_now AND
Navigator1.gps_start = Navigator.gps_start AND
Navigator1.gps_goal = Navigator.gps_goal AND
Navigator1.tTot = Navigator.tTot AND Navigator1.tRim = Navigator.tRim
AND bat0=VehicleData.bat AND batNow=VehicleData.bat

DEFINE Navigator1
(greenBox_id : String, kmRemains : Integer, kmCovered : Integer, 
kmTot : Integer, gps_now : <double, double>, 
gps_start : <double, double>, gps_goal : <double, double>, 
tTot : <Integer, Integer>, tRim : <Integer, Integer>, 
batNow : Integer, bat0 : Integer)
FROM Navigator1(greenBox_id=$id AND kmCovered!=0) AND
LAST VehicleData(greenBox_id=$id) WITHIN 5min FROM Navigator AND
LAST Navigator1(greenBox_id=$id) WITHIN 5min FROM Navigator
WHERE Navigator1.greenBox_id = Navigator.greenBox_id AND
Navigator1.kmRemains = Navigator.kmRemains AND
Navigator1.kmCovered = Navigator.kmCovered AND
Navigator1.kmTot = Navigator.kmTot AND
Navigator1.gps_now = Navigator.gps_now AND
Navigator1.gps_start = Navigator.gps_start AND
Navigator1.gps_goal = Navigator.gps_goal AND
Navigator1.tTot = Navigator.tTot AND Navigator1.tRim = Navigator.tRim
AND bat0=Navigator1.bat AND batNow=VehicleData.bat

DEFINE LowBatteryAtReleasePoint(ID : String )
FROM Navigator1 AND
batNow < (((batNow - bat0) x kmRemains) / kmCovered) +5)
WHERE ID = Navigator1.greenBox_id
```

**Note:** the proportion \((batteryStart - batteryNow) : coveredKm = x : remainsKm\) has been used, where “batteryStart” is the state of charge at starting point, “batteryNow” is the actual state of charge, “coveredKm” is the amount of kilometers already covered, “remainsKm” is the amount of kilometers that are missing to reach the destination and, finally, “\(x\)” is how much battery the vehicle need to reach destination.

This kind of query is easier expressed with PerLa language, using LLQs to produce short snapshots. With Tesla it is necessary to adopt a little trick to remember the state of the battery at starting point, using a recursive call
Chapter 4. Original PerLa and Tesla/TRex

of the event Navigator1 itself.

Example 3

Like the previous example, this query/rule check if battery estimated operating time is only slightly above to the time necessary to reach the nearest recharge point.

As before, the system must know:

- the state of the charge
- the gps position of the vehicle
- the distance of the nearest recharging point

Since it is necessary to calculate the amount of battery needed to reach the destination that, in this case, is not the releasing point but the nearest recharging station, the same proportion used in example 3 is used in this example.

This query is exactly the same than the one in example 3, the only parameter that have to be changed is the point of interest, from the destination point to the nearest recharging station. The PerLa HLQ is quite simple to understand, but it needs a high number of LLQ in support. In Tesla rule, on the other side, is necessary to use the same tricky query than in the previous example, which is not immediately clear.

Example 4

Test if the estimated travel time to release point is slightly below than remaining time from reservation expiration.

The information the system must have in order to compute the results are:

- travel time to release point
- reservation expiration time

80
4.3. Comparison of the Two Systems

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>placeRelease</td>
<td>GPSData</td>
<td>NP</td>
<td>Where the car have to be released</td>
</tr>
<tr>
<td>timeRelease</td>
<td>Integer[2]</td>
<td>NP</td>
<td>When the car have to be released</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(time)</td>
</tr>
</tbody>
</table>

VehicleData

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>gps_data</td>
<td>Integer</td>
<td>P</td>
<td>Flag that indicates if the car is taken or released</td>
</tr>
</tbody>
</table>

SpeedLimit

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>GPSData</td>
<td>NP</td>
<td>GPS of the street</td>
</tr>
<tr>
<td>limit</td>
<td>Integer</td>
<td>NP</td>
<td>Limit in force on that street</td>
</tr>
</tbody>
</table>

Table 4.6: PerLa’s data table for example 4

PerLa High Level Query

```sql
CREATE OUTPUT SNAPSHOT reservationExpiration (greenBox_id String, minutes Integer, orario time)
WITH DURATION 30m
AS HIGH:
SELECT greenBox_id, minutes, CURTIME() as now
FROM VehicleData, Reservation, SpeedLimit
WHERE VehicleData.greenBox_id = Reservation.greenBox_id AND
  VehicleData.gps_data = SpeedLimit.pos AND
  ((Reservation.placeRelease - VehicleData.gps_data) / SpeedLimit.limit) <
  (Reservation.timeRelease - now) + 5 AND
  ((Reservation.placeRelease - VehicleData.gps_data) / SpeedLimit.limit) >=
  (Reservation.timeRelease - now)
```

Tesla Rule

There is no simple way to write this rule with Tesla because it doesn’t provide the current time. It would be possible to write it recursively calling a timer every minute (or second), but it would add too much complexity to the rule, adding the dependency also to an external timer for correctly computing the results.

So it is possible to implement the request only as PerLa query.

Example 5

Check if the car is moving while the doors are opened.

Necessary information:
Chapter 4. Original PerLa and Tesla/TRex

- the speed is above 0
- the doors are opened (door_status = 0: closed/locked, 1: closed/unlocked, 2: opened/locked, 3: opened/unlocked)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>P</td>
<td>Speed of the car</td>
</tr>
</tbody>
</table>

Table 4.7: PerLa’s data table for example 5

PerLa High Level Query

CREATE OUTPUT SNAPSHOT OpenDoors (greenBox_id String)
WITH DURATION 2 h
AS HIGH:
SELECT greenBox_id
FROM VehicleData
WHERE speed > 0 AND door_status = 3

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>Speed of the car</td>
</tr>
</tbody>
</table>

Table 4.8: Tesla’s data table for example 5

Tesla Rule

DEFINE OpenDoors (ID : String)
FROM VehicleData (door_status == 3 AND speed > 0)
WHERE ID = VehicleData.greenBox_id

This example also demonstrates that for “basic” requests, PerLa and Tesla are equivalent.

Example 6

Check if the user driving style is out of threshold considering weather conditions.

The system must know:
- the vehicle speed
- the weather conditions
- user driving style

82
4.3. Comparison of the Two Systems

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>ID</td>
<td>Car identifier</td>
</tr>
<tr>
<td>gps_data</td>
<td>GPSData</td>
<td>P</td>
<td>Get the GPS position</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>P</td>
<td>Speed of the car</td>
</tr>
<tr>
<td>pos</td>
<td>GPSData</td>
<td>P</td>
<td>Position</td>
</tr>
<tr>
<td>climate</td>
<td>String</td>
<td>P</td>
<td>Weather in that position</td>
</tr>
<tr>
<td>limits</td>
<td>Integer</td>
<td>NP</td>
<td>Speed limit in that kind of weather</td>
</tr>
<tr>
<td>WeatherChanged</td>
<td>–</td>
<td>E</td>
<td>Change of climate</td>
</tr>
</tbody>
</table>

Table 4.9: PerLa’s data table for example 6

**PerLa Low Level Query**

STREAM WeatherChange: lists all the weather changing (only to bad weather, we consider “ice” when the temperature is less or equal than 4°C, when roads start to become slippery).

```
CREATE STREAM WeatherChange (position GPS_Data, climate String) AS LOW:
    EVERY 10 m
    SELECT position, climate
    SAMPLING
    ON EVENT WeatherChanged
    WHERE climate = "Rain" OR climate = "Ice" OR
    climate = "Snow" OR climate = "Fog"
    REFRESH EVERY 5 m
```

**PerLa High Level Query**

```
CREATE OUTPUT SNAPSHOT DangerousDriving (greenBox_id String)
WITH DURATION 2 h
AS HIGH:
    SELECT greenBox_id
    FROM VehicleData, WeatherChange, Weather
    WHERE VehicleData.gps_data = WeatherChange.position AND
    WeatherChange.climate = Weather.climate AND
    VehicleData.speed > Weather.limits
```

**Tesla Rule**

```
DEFINE DangerousDrivingRain(ID : String)
FROM VehicleData(speed >90) AND
    LAST Weather(VehicleData.pos-x<pos<VehicleData.pos+x)
    WITHIN 1h
    FROM VehicleData AND Weather.climate="rain"
WHERE DangerousDrivingRain.ID = VehicleData.greenBox_id
```

Or, in the case of ice:
Chapter 4. Original PerLa and Tesla/TRex

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greenBox_id</td>
<td>String</td>
<td>Car identifier</td>
</tr>
<tr>
<td>speed</td>
<td>Float</td>
<td>Speed of the car</td>
</tr>
<tr>
<td>gps_data</td>
<td>GPSData</td>
<td>Get the GPS position</td>
</tr>
<tr>
<td>pos</td>
<td>GPSData</td>
<td>Position</td>
</tr>
<tr>
<td>climate</td>
<td>String</td>
<td>Weather in that position</td>
</tr>
<tr>
<td>temp</td>
<td>Integer</td>
<td>Temperature</td>
</tr>
</tbody>
</table>

Table 4.10: Tesla’s data table for example 6

```
DEFINE DangerousDrivingIce(ID : String)
FROM VehicleData(speed>50) AND
  LAST 1h Weather(VehicleData.pos - pos < VehicleData.pos + x and temp<0)
WITHIN 1h FROM VehicleData
WHERE DangerousDrivingRain.ID = VehicleData.greenBox_id
```

As the example shows, PerLa allows to put all the “bad weather conditions” in a single LLQ, while in Tesla it is necessary to use different rules for each condition. By the way, both the implementations are perfectly equivalent one to each other.

### 4.3.2 Common Issues of the Systems

We present in the following two short examples that cannot be implemented in neither of the two languages.

**Example 7**

Check if there have been some traffic code infractions, basing on the knowledge in the road network.

The following information are needed:

- the position of the vehicle
- the speed of the vehicle
- the road network

**Example 8**

Check if the car entered into a restricted traffic area (ZTL) or fast tracks.

We need to know:
4.3. Comparison of the Two Systems

- the position of the vehicle
- the road network

Issues

Unfortunately, it is impossible to give a solution to both these requests, with neither of the two languages. The issue lays in the complexity of the requests.

In PerLa, it is possible to write a query that solves the problem; however this query will be very long, complex and it will probably be very inefficient from the computational point of view. Considering a Tesla rule instead, together with the complexity and length issues the main problem will be again the incapacity to deal with static data.

The requests can be simplified making some assumptions (e.g. considering the restricted traffic area to be round, instead of considering it to be a polygon), but we will obtain only approximate answers, that may lead to wrong conclusion (e.g. impose a fine when there is no reason to apply it).
CHAPTER 5

Reengineering and Extending PerLa and TReX

Contents

5.1 Reengineering the PerLa Middleware .................................. 88
  5.1.1 Middleware Refactoring .................................. 88
  5.1.2 Channel .................................................. 94
  5.1.3 Script Engine ............................................. 97
  5.1.4 Cassandra integration .................................... 97
  5.1.5 HTTP protocol integration in PerLa ......................... 98
  5.1.6 PerLa on TinyOS-based Motes .............................. 101
5.2 Extending PerLa .......................................................... 104
  5.2.1 Query Performance Enhancements .......................... 104
  5.2.2 Distributed PerLa ......................................... 105
  5.2.3 PerLa and COP ............................................. 107
5.3 Extending Tela/TReX ..................................................... 116
  5.3.1 RDF-3X SPARQL Engine .................................. 117
  5.3.2 TReX Engine Extension .................................. 118
  5.3.3 Benchmarking the New Semantic Enhanced Engine ....... 128
Chapter 5. Reengineering and Extending PerLa and TRex

This chapter presents the new extension made to PerLa and Tesla/TRex in order to fix some of the “issues” evinced during the comparison made in the previous chapter.

In particular a full reengineering of the PerLa system is presented in Sect. 5.1, Sect. 5.2 then presents also some other extension necessary to improve the PerLa system (e.g. distributed PerLa and PerLa COP integration).

Then the semantic enrichment extension to the TRex engine and Tesla language is presented in Sect. 5.3, presenting also some quick benchmark of the newly introduced semantic feature.

5.1 Reengineering the PerLa Middleware

Since the PerLa system presented in Sect. 4.1 is currently under continuous development, a complete reengineering of the system has been carried out during this thesis work.

In particular, a complete remodularization of the base system, allowing the system to be easier to maintain and develop (integrating new features) has been performed.

Moreover, in addition to the existing components, two new modules have been introduced in the system:

- A standard HTTP communication channel, adopted for allowing PerLa interoperability with standard web services, that now can be seen as standard PerLa sensors

- A communication channel among the PerLa server and motes (i.e. Mica2 motes) running TinyOS to let them to operate correctly in the PerLa system

5.1.1 Middleware Refactoring

In the follow up of this section a comprehensive description of how the PerLa Middleware has been rearranged is given, along with the rationale behind key architectural decision taken during the course of its development.

5.1.1.1 Functionality Proxy Component – FPC

The PerLa Middleware is a collection of software units designed to support the execution of PerLa queries. Its design revolves around the Functionality
Proxy Component (FPC), a proxy object which acts as a decoupling element between sensing nodes and middleware users. Essentially, the PerLa middleware constitutes the software environment needed to manage the lifecycle of a set of FPCs.

Overview Functionality Proxy Components are dynamically created by a Factory module, which assembles new FPC objects from a formal description of the node being registered into the middleware.

The most prominent trait of the FPC is its hardware-agnostic interface, a uniform API that provides a consistent method of interaction with all the heterogeneous devices found in a (wireless) sensing network. Viewed through the abstraction provided by an FPC, each device is a collection of attribute whose values can be queried or altered using two primitive operations, respectively called get() and set(). The usage of these two basic commands neither requires knowledge of the sensing network, nor of the device that will ultimately perform them. Attribute values are produced or consumed by a computing node. Inside the PerLa middleware, attributes are characterized by a name and a data type; this meta-information enables PerLa users to correctly address a single specific piece of information. PerLa currently supports the following data types:

- **INTEGER** Integer numbers
- **FLOAT** Fractional values in floating point representation
- **STRING** Sequence of characters
- **BOOLEAN** Logical value (true or false)
- **ID** PerLa device identifier
- **TIMESTAMP** Identifier of a specific moment in time

The FPC interface The concept of Device Attribute is mainly employed to correlate middleware components with the information that they provide access to.

As explained in previous paragraphs, accessing information through the primitive operations provided by a FPC do not require any knowledge of the sensing network; the user only needs to specify which attributes must be retrieved or modified.

All the operations performed by the FPCs are asynchronous and non-blocking. The immediate result of all FPC methods’ invocations is a Task, an object whose function is to allow control over the requested asynchronous
Chapter 5. Reengineering and Extending PerLa and TReX

computation (e.g., query for completion, cancel the operation, ...). The output of the operation, once ready, is delivered with an asynchronous invocation of the TaskHandler provided at request time. This operating principle is the high-level equivalent of micro-controller hardware interrupts: users define an interrupt source and an interrupt handler, then the micro-controller asynchronously invokes the callback handler whenever a new event is generated.

The non-blocking FPC interface is a feature introduced with the new PerLa middleware architecture that aims at improving reaction times and increase the overall system scalability. The reactive model fostered by this new design addresses two flaws of the previous implementation, namely the excessive number of threads required to parallelize several synchronous I/O operations, and the loss of performance due to frequent context switches imposed by the former thread-based model. It is worth noting that this non-blocking interface design is extended to all software components of the PerLa Middleware.

Operation management and scheduling The current FPC implementation supports the execution of multiple concurrent get() and set() operations. The actual scheduling of processing tasks on the remote device varies according to runtime conditions, as it depends on the number and type of requested operations, as well as the capabilities of the remote computing device itself.

The FPC may, for example, schedule a single periodic sampling operation on the sensing node, and fan-out its outcome to several consumers, potentially resampling the data stream if the requested sampling rates are different. Otherwise, in case of several non-periodic (one-off) get() operations, the execution may be performed sequentially.

The FPC is also responsible for simulating certain types of operations, when these are not provided by the remote sensing device.

Internal FPC Components The FPC is composed of several independent software units, each of which is responsible for managing a single aspect of the interaction with the remote device. This new modular design improves on the previous monolithic architecture by promoting component reusability and greater decoupling between units with separate concerns, and provides a cleaner interface better suited for expandability.

Every FPC object is composed of the following units:

CHANNEL A software component capable of performing I/O operations.
5.1. Reengineering the PerLa Middleware

Channels are responsible for managing the communication between the PerLa Middleware and the remote devices.

**MAPPER** Data marshaller/unmarshaller. *Message Mappers* allow the FPC to interpret byte streams received from a Channel and to serialise Java object prior to transmission.

**CHANNEL MANAGER** The component which manages all the *Channels* used by the FPC. The *Channel Manager* is a fundamental part of the PerLa architecture, as it is responsible for dispatching asynchronous events and data towards the appropriate consumer.

**SCRIPT ENGINE** A simple engine for running small scripts, used by the FPC to dynamically bind high level data requests to native processing tasks run on the remote device.

**OPERATION SCHEDULER** Manages the scheduling of processing task on the remote device.

Additional details on the internal software architecture of the Channel, Mapper and Script Engine components are available in the following.

![Perla reengineered components](image)

**Figure 5.1:** Perla reengineered components.
Dynamic FPC Creation  As previously mentioned, FPCs are dynamically created at runtime by an object factory, called FpcFactory (Fig. 5.2). This component is responsible for the instantiation and assembly of all the constituent part of an FPC.

![Diagram of FPC creation process]

**Figure 5.2: Dynamic FPC creation process**

The starting point for the creation of a new FPC is the Device Descriptor, an XML document which contains a machine parseable description of a single computing device of the sensing network. Device Descriptors are organized in different sections; each section defines the behaviour of a specific aspect of the FPC being created.

The first part of a Device Descriptor contains a textual description of the physical device (type attribute), and a series of XML namespaces. The XML namespaces have a twofold function in the PerLa middleware: first and foremost, namespaces are used to disambiguate element and attribute names; then namespaces are used by the FpcFactory to define which implementation of an individual component needs to be instantiated.

In the example provided below, the Device Descriptor requires the creation of Channel with type “HTTP” and Message Mappers of type “urlen-
5.1. Reengineering the PerLa Middleware

```xml
<?xml version="1.0" encoding="UTF-8"?>
<device type="test_descriptor">
    xmlns="http://perla.dei.org/device"
    xmlns:http="http://perla.dei.org/channel/ http:"
    xmlns:ser="http://perla.dei.org/channel/serial"
    xmlns:js="http://perla.dei.org/fpc/message/urlencoded"
</device>
```

Listing 5.1: Device description header

```xml
<attributes>
    <attribute id="temperature" type="float" permission="read"/>
    <attribute id="room" type="int" access="static"
        value="5" permission="read"/>
    <attribute id="pressure" type="float" permission="read"/>
    <attribute id="period" type="integer" permission="write"/>
</attributes>
```

Listing 5.2: Attribute declaration

coded” and “json”.

The first section of the Device Descriptor after the header contains a declaration of all attributes exposed by the device.

For each attribute the device designer can specify the following properties:

ID Textual label used to univocally identify the attribute

Type Data type of the attribute

Permission The allowed operations on the attribute (read, write, ...)

Access Type of access. Can be set to DYNAMIC (the attribute is generated by the device) or STATIC (the attribute value is constant). Static attributes are required to declare their value

The Channel section defines which Channels have to be used to interact with the remote device. As in the example below, the set of configuration attributes required by a Channel depends entirely on the type of the Channel itself. The following code snippet also demonstrates how namespaces are employed in the Device Descriptor to link configuration parameters with a specific component type.

The I/O Request section allows PerLa user to define a series of native, primitive functions that can be performed on the remote device. These primitives will then be used in other sections of the descriptor to specify more complex behaviours. I/O Requests are described with the same
paradigm previously seen in the Channel section; an XML namespace defines the type of request, which is then used to instantiate the right object using the given parameters.

Messages and Message Mappers are defined in the Message section. This part of the Device Descriptor is used both for defining the structure of all messages used to communicate with the device and for binding attributes to a specific message field. This section uses the familiar namespace paradigm for selecting a particular implementation of FPC component. In the example below, a Mapper for JSON objects is selected using the “json” namespace. It is important to note that, once a particular data type is selected, the user can define the structure of a message using concepts and idioms typical of that format.

The last Device Descriptor section describes how the primitive functions made available by the network node have to be combined to create a data record. Each operation declared in this section defines a Script, which is executed by the FPC whenever one of the emitted attributes is requested with a PerLa query.

5.1.2 Channel

A Channel is a software unit capable of performing I/O operations. It defines a network-independent interface that can be used by high-level com-
5.1. Reengineering the PerLa Middleware

```xml
<messages>
  <js:message id="weather_msg">
    <js:object name="msg" type="object">
      <js:object name="main" type="object">
        <js:object name="temp"
          type="float"
          qualifier="attribute"
          attribute-id="temperature"/>
        <js:object name="pressure"
          type="float"
          qualifier="attribute"
          attribute-id="pressure"/>
      </js:object>
    </js:object>
  </js:message>
</messages>

Listing 5.5: Message declaration – 1

```xml
<operations>
  <get id="get_weather">
    <i:submit request="weather" channel="http_ch_01"
      variable="result" message="weather_msg"/>
    <i:put variable="result" attribute="temperature"/>
    <i:put variable="result" attribute="temperature"/>
    <i:emit/>
  </get>
</operations>

Listing 5.6: Message declaration – 2

```xml
```
ponents of the PerLa Middleware, regardless of the particular protocol stack employed. Different concrete channel implementations are available in the PerLa Middleware; each channel is committed to the management of a single communication technology.

Channel operations are asynchronous by nature; users send requests by means of the submit method, and receive results through the specified IO-Handler. Request execution may be monitored or cancelled using the IO-Task object returned upon submitting a new IOResquest. Synchronous data transfers can be tracked despite the inherently asynchronous operating principle of the Channel; in case of synchronous communication the association between the request that triggered an I/O operation and the result of such operation is maintained by the handler callback.

The Channel component is also designed to handle asynchronous data transfer operations initiated by the remote device, a scenario that may occur whenever the node independently streams a series of data records to the PerLa system. Payload data corresponding to such communications is made available through a single IOHandler, usually associated with the FPC’s Channel Manager.

A Channel usually implements a complete communication protocol that allows the FPC to effectively and efficiently communicate with the remote device or service. The use of multiple high-level, moderately specialised Channels is preferred over a limited set of generic, multipurpose Channels, since the latter design would put too much overhead on both the FPC and the final user.

Ideally, each PerLa Channel should implement a protocol corresponding to the Application layer of the OSI model. This is in direct contrast with the previous middleware design, which fostered the creation of a small set of re-useable Channels that only managed Transport layer communications. Moving high-level protocol management from the FPC to the Channel is beneficial in that it promotes the use of readily available third-party communication libraries. As an example, a Channel implementation of the FTP protocol could use one of the immediately available FTP Java libraries. This would not be feasible if application and transport logic were to be split between the FPC and the Channel. The creation of new Channel objects at runtime is performed using a ChannelFactory. The PerLa Middleware contains a specialised ChannelFactory for each Channel implementation (i.e., the HttpChannelFactory only creates instances of the HttpChannel).
5.1. Reengineering the PerLa Middleware

5.1.3 Script Engine

The Script Engine is a newly introduced component in the PerLa middleware, specifically designed as a tool for manipulating data received or sent through the Channel. Its purpose is to act as an “impedance matcher” between the record-based world of PerLa and whichever data structure is in use on the remote sensing network. PerLa Scripts are composed of a series of simple imperative instructions, whose main functions are:

- Create and customise I/O requests with information that dynamically changes at runtime
- Submit I/O requests to a remote device
- Convert data received from the remote devices into one or more records

PerLa Scripts employ all primitive entities described in the previous sections of this document (channels, I/O requests, mappers, etc.), and connect them to implement complex behaviours.

5.1.4 Cassandra integration

The output of a PerLa query can be easily dumped in a Cassandra Database using the DatabaseWrapper class available in the perla-cassandra module. Apache Cassandra is a hybrid key-value and column-oriented open source distributed database management system designed to handle large amounts of data across many commodity servers. Unlike traditional RDBMS, rows in a Cassandra table do not necessarily share the same set of columns, and a column may be added to one or multiple rows at any time.

5.1.4.1 The DatabaseWrapper class

The DatabaseWrapper is the class responsible for saving new data records to a Cassandra database. As suggested by its name, this class is a wrapper that contains a record producing object. Its interface is composed of 3 methods, which correspond to the main data access primitives available in an FPC. Every record produced by an FPC is enclosed in a DatabaseWrapper and automatically dumped into a Cassandra database. The name of the output table can be retrieved from the DatabaseTask object returned when starting a new query.

As it can be seen in figure 5.3, the DatabaseWrapper class uses a custom Handler to save data inside a Cassandra table. The records produced by the FPC can also be relayed to an additional user-specified
Chapter 5. Reengineering and Extending PerLa and TRex

handler, thus allowing additional components to be daisy-chained for real-time data analysis.

Figure 5.3: DatabaseWrapper architecture

Records are inserted inside the database by the DatabaseHandler. This object is designed to support interruptions in the record flow, and to append data in the correct table even when a query is paused and resumed. Every record is complemented with a randomly chosen Identifier prior to its insertion in the database.

5.1.5 HTTP protocol integration in PerLa

In its first versions PerLa connects sensors through TCP/IP protocol [119] while nowadays some information could be retrieved from other system (e.g. web services) and not only from sensors. There exists plenty of Web Services offering APIs (JSON/REST or SOAP), like weather forecast web services (providing temperature, pressure and other weather-related parameters). These APIs give information that PerLa could model like a virtual sensor.

The enabling feature for this new paradigm implementation is the definition of a new kind of channel that allows the communication with Web Services. The real challenge is to create this channel as generic as possible,
5.1. Reengineering the PerLa Middleware

![Diagram](image)

**Figure 5.4: FPCs Channels**

so it will be independent from the content-type used by the HTTP call for transferring data (in the *request* or in the *response*).

5.1.5.1 PerLa Channel design

As presented in 5.1.2 the PerLa *virtual channel* (or simply channel) is the component responsible for the communication between FPC and real device. In this context the device is represented by a common Web Service with a HTTP interface (ex. SOAP or JSON/REST).

In order to define and configure a channel PerLa offers three base classes:

*AbstractChannel* is the abstraction of any communication channel between PerLa (FPC in particular) and the real device. We can consider channels based on TCP Socket technology, WebSocket, for advanced device, or HTTP protocol.

This object takes care of sending a generic *Request*, creating a *Response* and adding it to the queue passed to the FPC.

*ChannelRequestBuilder* is the object used by the FPC for creating the requests to be passed to the channel. Each request class has its *ChannelRequestBuilder*.

*Request* is the object created at run-time when a FPC needs to retrieve some data from a real device. After being created by *ChannelRequestBuilder*, it must be configured with some parameters. For example a
Chapter 5. Reengineering and Extending PerLa and TReX

JSON/REST service may need to receive the name of the city for retrieving a particular temperature.

The complete process of how the channel handles a request is shown in Figure 5.5.

![Diagram of request elaboration]

**Figure 5.5: Request elaboration**

Usually the channel provides synchronous requests; those request are added to the queue shared between the channel and the FPC. The FPC can then use immediately the response (synchronous call) or it can postpone this action and retrieve it just when it is needed (asynchronous call).

The PerLa channel structure is dynamically built at run-time through the translation of a Descriptor (Java representation of XML descriptor) into PerLa objects.

This translation is based on the following Java objects:

**ChannelFactory** is responsible of XML channel descriptor validation. So it is used by a FPC for creating Channel during the PerLa start-up phase

**ChannelRequestBuilderFactory** is responsible of the validation of the XML request descriptor. So it is used by FPC for creating a ChannelRequestBuilder during the start-up

**ChannelDescriptor** is the Java representation of channel XML tag
5.1. Reengineering the PerLa Middleware

RequestDescriptor is the Java representation of request XML tag

The proposed implementation details (which are the implemented classes, which APIs they provide, . . .) will be presented in Appendix B.1.

5.1.6 PerLa on TinyOS-based Motes

A TinyOS application has a particular structure composed by some files that are then compiled and installed on motes. There are two types of components in nesC\textsuperscript{1}: modules and configurations. Modules provide the implementations of one or more interfaces (collection of commands and events). Configurations are used to assemble other components together, connecting the different interfaces used by components. It is important to remind that TinyOS is an operating system with an event-driven architecture.

5.1.6.1 PerLa Message Structure

A TinyOS (TOS) message is mainly composed by three parts: header, data and metadata. Header contains general information about the message for the communication; data contains the values that have to be exchanged; metadata contains additional communication information. The structure of the message is the one represented in Fig. 5.6:

![Figure 5.6: Standard TinyOS message structure](image)

Because it is strongly recommended not to directly change the TOS header structure, the design choice was to create a PerLa message with its own header and payload. Then, we encapsulate it in a standard TOS message. By doing this, a total control of the PerLa message is kept (both the header and the payload part) and we lay on a TinyOS standard that will always work, even in case of new motes or of changes in TinyOS structure (Fig. 5.7).

The header part contains the following structure:

```c
typedef nx_struct perlaHeader {
    nx_uint16_t id;

1http://nescc.sourceforge.net/```
Chapter 5. Reengineering and Extending PerLa and TReX

Figure 5.7: PerLa message structure

```c
nx_uint16_t timestamp;
x_uint16_t type;
x_uint16_t numPckt;
x_uint16_t numPcktToReceive = 0;
} perlaHeader_t;
```

Where id is the identifier of the mote that sends the message, timestamp contains date and hour in which the message was composed and send, type specifies the kind of the message (something like “0 descriptor, 1 periodic sampling (+ one-shot), 2 event based sampling, 3 end of periodic sampling, 4 end of event based sampling, 5 periodic response, 6 event based response”).

numPckt and numPcktToReceive are two fields used when the amount of information to be sent exceeds the maximum size of message_t and data must be splitted over more packets. Every time that the PerLa payload is splitted and PerLa header is replicated with different values of this field: the first one is used to keep the received packet ordered allowing to rebuild the original data, the second one tells to the receiver how many packets it must receive before the communication of the information ends.

The PerLa payload, instead, has the following structure:

```c
typedef nx_struct perlaPayload {
    nx_uint8_t data[TOSH_DATA_LENGTH-sizeof(perlaHeader)];
} perlaPayload_t;
```

The payload is a simple array of bytes. There is no need to specify a more complex structure because the underlying idea is to keep it as more generic as possible. The sequence of bytes is then composed and interpreted both from motes and channel by a predefined XML descriptor file. This file is pre-loaded on motes at production time and, when a mote is integrated in a wireless sensor network, it sends its descriptor to the channel at boot. At this point both channel and motes know how to write and read the sequence of bytes, then they can put in or get out information.

102
5.1.6.2 Implemented Features

At the state of the art, a mote with installed the PerLa TinyOS application
is able to perform several operations. In particular it can:

- Periodically sense temperature and/or humidity with a specified time
  period (in seconds); each time the timer fires it reads the values,
  prepares the message and sends it to the channel
- Sense temperature and/or humidity one-shot. This means that, when a
  mote receive the request, it sense the chosen parameter once and sends
  the response
- Sense temperature and/or humidity based on an event trigger. Un-
  til now the events supported are “temperature greater than” and “hu-
  midity greater than”, even if it will be pretty simple to extend those
  boundaries to different ranges. An event reading is, in other words, a
  periodic reading with a control on the sensed data: if the sensed data
  satisfies the conditions, then they are sent to the channel middleware
  component, otherwise they are discarded

It is possible to have more sampling operations in parallel; the only con-
straint is that it can be only one operation - per type and per sensor - run-
ning.

Further details on the provided implementation will be presented in Ap-
pendix B.1.

5.1.6.3 Open Issues

In the end, there exists many problems that have not been solved yet.

At first, it is necessary to extend the event-based sampling condition:
motes can currently control only one type of condition for event-based sam-
pling that is “measure greater than”. It could be useful to have condition
like “lower than” or “between .. and ..”, being able to choose among them
in a dynamic way.

Secondly, in wide wireless sensor network, it can be very useful to im-
plement some routing protocol in order to cover bigger distances and to
save battery on motes (it is not strange that a hop-to-hop communication
be cheaper than a direct communication), since battery is one of the
weakness of motes and wireless sensor network. If some tricks are run
based on battery state of motes, they probably can perform better and for a
longer time.
Moreover a more complete set of network operations must be implemented in order to ensure the maximum reliability of the network.

5.2 Extending PerLa

In order to let PerLa fully integrate within our framework, some features have to be added: first of all some performance enhancement has to be achieved in query processing since the system must gain quasi-real-time performances, allowing it to process huge quantity of data in reduced time.

A full distribution of the PerLa system is necessary, to correctly split computation loads among the whole system components, to gain better processing performances.

5.2.1 Query Performance Enhancements

PerLa is not yet comparable with high performances stream handling commercial system, due to some lacks in the original implementation idea, as it has been pointed out in Sect. 4.3.

The system limit is due to the architecture provided for the HLQ processing: while LLQ runs directly on the sensors, describing how them should gather and dispatch their data, HLQ at the moment could be executed only on the PerLa central server.

Unfortunately the current PerLa implementation lays upon a traditional DBMS server (MySQL). This means that the system acts as a proxy for the sensors, gathering all their data and inserting them into the database. Once data are stored in the database, HLQs (that syntax highly recall standard SQL syntax) can be run directly on the DBMS, retrieving the final results.

The key issue with this architecture is that MySQL (or other similar DBMS) cannot deal with such an high arrival rate combined with thousand of queries possibly simultaneously running on the system.

The only way to handle such a big amount of data and queries is to implement an intermediate structure on which perform real-time querying.

The most valuable solution is to use a main memory NoSQL database (e.g. Redis\(^2\) or other NoSQL solution like Cassandra\(^3\)) in order to speed up the processing storing recent, useful data in main memory for processing before moving old, no more necessary data out of main memory (in a traditional RDBMS) together with queries execution results if necessary (a query may tell the system to store its output for future processing).

\(^2\)http://redis.io/
\(^3\)http://cassandra.apache.org/
5.2. Extending PerLa

Such a modification requires a complete revision of the query executor module, since now queries must be translated in order to allow them to be processed on a NoSQL data storage and management system.

In particular, the query execution can be split into two main sections:

- on-the-fly queries that need to process only recent data are executed using the main memory data processing structure
- on the other side, queries that rely on historical data (e.g. because they need to compute some kind of trend in data) run on a RDBMS structure

This happens because traditional DBMS already provides useful and smart mechanisms for processing a big quantity of data calculating aggregates (those are needed for calculating historical trends in a smart way), while main memory structures provides fast computation of simple queries.

The main memory structures must also take into account that although several queries may use the same set of data, each query may have a different window. So the system must keep in memory the data necessary to compute the answer to the query which has the widest window, to avoid the need to retrieve data from the RDBMS structure, lowering the on-the-fly query answering performances.

These considerations has already been taken into account in the ongoing reengineering process described in the previous section and constitutes the starting point for the new QueryExecutor module to be rearranged.

The QueryExecutor module is now composed of:

- the OnTheFlyQueryExecutor module, which runs queries in main memory
- the StandardQueryExecutor module, which runs all those queries that do not need to be run in main memory

Both this two modules are implemented in Java, allowing them to perfectly integrate in the existent system.

5.2.2 Distributed PerLa

PerLa currently acts as centralized server architecture. In case of small sensors’ networks this will not be a problem, since the central server have the capability to handle the whole network traffic and react to the appropriate queries.
Chapter 5. Reengineering and Extending PerLa and TReX

However since in complex and wide sensors’ networks problems may arise due to the heavy network load, distribution may result as a crucial aspect for the PerLa system.

Distribution is possible since in these networks some nodes with higher capabilities, both for computation and routing – usually defined as gateways – are located somewhere in the network path.

In addition gateways are the ideal candidates for the computation load distribution since they usually \( i \) gather data from several nodes and \( ii \) have much more computing power than needed for routing purposes.

On gateways a “light” version of the PerLa server can be installed and executed. The central server can then decide, depending on the query, how to use gateways. In some queries all the data coming from the underlying sensors may be necessary at central level, while in some cases an aggregate value (e.g. the average temperature detected by sensors located in a given zone, connected to the same gateway) may be enough for the central server needs.

In the second case the PerLa central server just need to know that the “complex sensor” (the gateway) outputs a stream in a given format (and the format is described as it were a standard sensor using the FPC XML descriptor), aggregating data coming from a part of the sensor network, no matter which are these sensors (that on the other hand must be well known to the PerLa gateway level) that become absolutely and totally transparent to the central system.

However, before the reengineering process, it was really hard to understand if such “light” version of PerLa can be obtained extracting only the necessary modules from the original implementation.

Now it is possible to produce many different “light” version of the PerLa server. The main modules involved are the Channel module and the two query executor (OnTheFlyQueryExecutor and StandardQueryExecutor) modules.

The output of the subsystem can be seen from the central server as a sensor (e.g. like a Web Service sensor) that provides as fields the ones of the query that runs on it.

In such a way the subsystem can be described by an XML descriptor in the central server, and its data could be used for computing other HLQs. From a database point-of-view, the subsystem output results are simply a view on data coming through a specific gateway.
5.2. Extending PerLa

5.2.3 PerLa and COP

At the current development stage, PerLa contextual features mainly focus on data, and the actions that can be performed in response to contextual changes are limited to the execution of PerLa statements [118]. Context has the role of a data “tailor”, i.e., it allows the user to define which data, retrieved by sensors, must be selected in a specific situation. The actions permitted are essentially limited to the activation or deactivation of devices, to the setting of some sampling parameters or to a limited range of tasks performed by some actuators, as presented in the office example in the previous section.

**Context management** PerLa with its queries allows to monitor the entire life cycle of the information.

The PerLa middleware has been designed to work with several data streams, produced by the sensing devices; the Low Level Queries allow to set their working mode: the sampling intervals, which data must be selected and the computation to perform on the sampled data. With PerLa CL (see Sect. 4.1.3), the designer can define a sort of **contextual dynamic view** on a data stream. On the contrary, COP is not directly aware of how information is provided; in fact it is not directly responsible of sensors, but it uses the information provided by them, to perform behavioral variations. Adopting COP, the developers must implement a mechanism to monitor continuous data sources, in order to provide contextual information.

Moreover, PerLa is very suitable for context distribution; the CDT model deals naturally with contexts belonging to different groups of sensors and to distributed instances of the application. The nodes at the lower level of the tree could be used to abstract several instances of a dimension, creating a local CDT for each instance. In this way, context data can be distributed to different locations, leading to the introduction of a combined CDT comprising a primary CDT and one or more local CDTs. An example of CDT distribution is provided by the Green Move application, in which – for privacy – a portion of the CDT is maintained locally to the user’s devices and is used to complete the context-based data filtering [99].

Even if, several threads may exist for each local instance and each thread adapts its own behavior w.r.t. the instance in a different way, with COP it is only possible to implement the behavioral variations of the instances; mechanisms to compose information coming from different local contexts, in order to infer a higher level context, and for data sources monitoring are not supported. For this reason, it may become necessary to introduce ded-
Chapter 5. Reengineering and Extending PerLa and TReX

icated components to generate significant contextual information, starting from rough data provided by sensors, and to decide which layers must be activated on the application.

The PerLa middleware has the components already designed for this purpose: the CM for what concerns the context and the application actions management, and other low level components for what regards the data sources monitoring, their operational impact being hidden to the user. Designers, through the PerLa CL, are relieved of the responsibility to develop dedicated components for context management which could result a rather complex task. With the possibility to define new contexts by composing other contexts at run-time, it becomes easier to specify the desired number of contextualized actions.

Enacting behavioral variations

The semantics of the ON ENABLE and ON DISABLE clauses of PerLa CL could apparently look similar to that of the with and without statements of COP, enabling (or disabling) a given procedure, retrieving some data from the sensors and thus operating on them.

If the context is active and therefore the condition in the WHEN clause of the corresponding concept is satisfied, an established action is performed. In the office example shown in Listing 5.9, the context OverheatMonitoring checks every thirty minutes if, in the set of data provided by a group of devices, there is a value in the temperature field higher than the settled threshold, and if it finds it, the air conditioner switches on the cooling function.

The following PerLa code (Listing 5.7) shows that, by installing a new air outlet in the offices, its activation becomes possible in case of persistent smoke. This new function is available at the same time as we plug the air outlet into the system, without the necessity of further changes to it. In addition, also a new configurable air conditioner has been installed in the offices. It can work in different modalities, depending on different properties of the room environment (it can cool down or heat up the temperature and it can dry off the environment). The corresponding CDT is shown in Figure 5.8:

The context SmokeMonitoring presented in Listing 5.7, is actually an extension of the fire context \(^4\) since it performs additional actions (e.g. it stops the ventilation not to spread the smoke and it opens the outlet to let it flow away) in order to be more effective in case of fire. PerLa CL creates a direct connection between monitoring and adaptations: the context

\(^4\) The fire context was presented in Chapter 4, Sect. 4.1.3, Listing 4.2.
5.2. Extending PerLa

![Context Dimension Tree (CDT) – Addition](image)

**Listing 5.7:** The SmokeMonitoring example in PerLa

```
CREATE CONTEXT SmokeMonitoring
ACTIVE IF Location = 'office' AND Smoke = 'persistent'
REFRESH EVERY 1h
ON ENABLE (SmokeMonitoring) :
SELECT smoke
SAMPLING EVERY 10 m
EXECUTE IF EXISTS (smoke)
SET PARAMETER air_outlet = TRUE
SET PARAMETER alarm = TRUE
SET PARAMETER speed = 0
ON DISABLE (SmokeMonitoring) :
DROP SmokeMonitoring
SET PARAMETER air_outlet = FALSE
SET PARAMETER alarm = FALSE
```

SmokeMonitoring is automatically activated and it checks immediately for the presence of smoke, in order to determine if the air outlet must be opened or not.

To implement the office example with COP, an external mechanism to monitor changes in context must be specified in addition to the definition of layers and when they have to be activated. An intuitive solution could be the introduction of a thread to monitor the temperature and the risk of fire, and another thread to monitor the smoke level in the room. With COP, we can define a **context** structure to encapsulate changes related to the smoke detector. In fact, we can assume that the activation of the fire alarm is the “normal” behavior, whereas the activation of both the fire alarm and of the air outlet represents a variation.

In order to provide a comparison between PerLa and COP, we adopt a conditional composition approach for the next examples, due to the presence of the structure context and the **when** clause.

The proposed COP code (Listing 5.8) translates the PerLa example of Listing 5.7: the main thread SmokeMonitoring sleeps for ten minutes (cf. the PerLa **SAMPLING EVERY 10m** clause) and, when resumed, it controls the smoke level in the room; if it detects persistent smoke (cf. the PerLa
Chapter 5. Reengineering and Extending PerLa and TRex

```java
context SmokeRisk {
  in(SmokeMonitoring sm) && when(SmokeMonitoring.smokeRisk()) {
    with(SmokeLayer);
  }
}

class ActiveActuators {
  public activeAlarm() {
    // When this method is called by thread FireMonitoring,
    // the fire alarm is activated
  }

  layer SmokeLayer {
    activeAlarm() {
      // If the layer is activated, the air outlet will be opened
      Outlet.sendOpenCmd();
    }
  }
}
```

Listing 5.8: The SmokeMonitoring example in COP

 ACTIVE IF clause), it activates the context smokeRisk() in which the with statement will activate the SmokeLayer in the class ActiveActuators, so executing the same actions that the PerLa code invokes in the ON ENABLE clause.

When SmokeLayer has been activated and if a high amount of smoke is detected, the partial method activeAlarm() switches to "open" the state of the air outlet. When the layer is deactivated, the state of the actuator is switched to "close", going back to the previous situation.

This example shows that context in COP is more driven by events than by rough data; it also introduces other similarities between PerLa and COP, at least from a conceptual point of view. The concepts of partial components and partial methods could be considered based on the same idea of composing different entities to provide a new behavior; the PerLa context query and the COP context structure contain dedicated statements to declare when context changes and which actions must be executed in response.

PerLa partial components and COP context entities refer to context composition and adaptations as a whole, i.e. from the composition of different basic data in order to obtain a new higher concept, to the execution of combined actions, covering both data management and behavioral variations.

To reinforce this idea, consider the extension of the office example presented in Fig. 5.8 (Listing 5.7) to a more complex case shown in Listing 5.9. The dimension AC_Temp tells if the system is working and if it is cooling
CREATE DIMENSION AC_Temp
CREATE CONCEPT off
    WHEN ac = 'off'
CREATE CONCEPT cool
    WHEN ac = 'cool'
CREATE CONCEPT heat
    WHEN ac = 'heat'
CREATE DIMENSION AC_Humidity
CREATE CONCEPT dry
    WHEN ac_dry = 'on'
CREATE DIMENSION AC_Ventilation_Speed
CREATE CONCEPT stop
    WHEN speed = 0
CREATE CONCEPT slow
    WHEN speed > 0 AND speed < 0.3
CREATE CONCEPT medium
    WHEN speed >= 0.3 AND speed < 0.7
CREATE CONCEPT fast
    WHEN speed >= 0.7

CREATE CONTEXT OverheatMonitoring
ACTIVE IF Location = 'office'
    AND Env_Temp = hot AND Env_Humidity.h_level > 0.65
REFRESH EVERY 30m
ON ENABLE (OverheatMonitoring) :
    SELECT temperature, humidity
SAMPLE EVERY 10 m
EXECUTE IF EXISTS (temperature, humidity)
SET PARAMETER ac = 'cool'
SET PARAMETER ac_dry = 'on'
SET PARAMETER speed = 0.65
ON DISABLE (OverheatMonitoring) :
    SET PARAMETER ac = 'off'
DROP OverheatMonitoring

Listing 5.9: The OverheatMonitoring example in PerLa

or heating the office; the concept AC_Humidity sets the condition for
the activation of the dry mode, while the dimension AC_Ventilation-
_speed indicates the current speed of the fan, and is used later on to adjust
the speed, on the base of the values of humidity and temperature.

As an example of how the coordination of several actions related to dif-
ferent contexts can be achieved by the CL in a simple way, suppose that a
context VentilationMonitoring (Listing 5.10) is declared in order to monitor
when the office temperature lowers too much: in this case there would be a
partial overlap between the data required by contexts VentilationMonitoring
and OverheatMonitoring (Listing 5.9). Both the composed contexts need
the current temperature value as a partial context while only OverheatMon-
toring requires humidity (see Fig. 4.3); the PerLa middleware deals with
both the composed contexts to provide data and to perform actions in or-
der to set the air conditioner variables: the fan speed, the humidity and the
temperature control.

Furthermore, PerLa can be adopted for applications performance mon-
Chapter 5. Reengineering and Extending PerLa and TRex

| CREATE CONTEXT VentilationMonitoring |
| ACTIVE IF Location = 'office' AND Env_Temp = cold |
| REFRESH EVERY 30m |
| ON ENABLE (VentilationMonitoring) : |
| SELECT Temperature |
| SAMPLING EVERY 10 m |
| EXECUTE IF EXISTS [temperature] |
| SET PARAMETER ac = 'heat' |
| SET PARAMETER speed = 0.65 |
| ON DISABLE (VentilationMonitoring) : |
| SET PARAMETER speed = 0.3 |
| DROP VentilationMonitoring |

Listing 5.10: The VentilationMonitoring example in PerLa

...itoring. Sensors can be configured for this purpose and the collected data can be used to change the working parameters of the monitored application, in order to increase its efficiency. For example, if the application response time must be decreased in case of heavy workload, at the risk of increasing the energy consumption, the proper dimensions and concepts must be introduced in the CDT and a simple CL query to the PerLa middleware will satisfy the request.

Therefore, the application developers must declare the significant context changes and which actions must be performed in different situations, with no need to implement anything at the operational level.

Is an integration possible? In the previous sections, we saw how COP introduces the problem of choosing the most appropriate way to compose the application, also including the choice of the layers composition mechanism. For example, dynamically scoped activation approaches are convenient when all the entities in the control flow are context-dependent, while per-object activation is a suitable solution when behavioral variations partially cross-cut the application structure, and it could happen that the choice made at design stage may prove wrong during development [113]. However COP cannot be considered a good solution for monitoring the entities from which contextual data can be retrieved not only in direct comparison with PerLa, but even with dynamic AOP; dynamic AOP, in fact, allows to separate cross-cutting concerns, such as monitoring tasks, with minimum impact on the code.

On the other hand, as we showed in Chapter 4 (Section 4.1) and in the previous section of this chapter, PerLa is a complete support to manage all the entities involved in the pervasive environment and it also very easily
5.2. Extending PerLa

allows to add new functionalities, even while the applications runs, without
having to manually edit the operational software [119] so "covering" all
the most important aspects of the application and not only those related to
context.

Then, the most intuitive solution to the implementation of a Context-
Aware self-adapting system should be to delegate everything related to the
configuration of sensors, the intermediate computations on data, and con-
texts declaration and management to PerLa, and to use COP to perform
behavioral variations, using the contextual information provided by PerLa
(see also [120, 121] for further details).

Considering that PerLa creates several data streams which include the
values retrieved by sensors, a context can be viewed as particular context
data stream, i.e. a normal PerLa stream, properly adjusted to provide only
data related to the defined context. This operation is performed by the CM:
it manages the CDT and the actual contexts, and creates contextual streams
accordingly. Some simple actions, such as those of the office examples, can
be still performed by the CM.

COP, in turn, can handily use the information inserted in context data
streams to implement the desired behavioral variations. The programmer
has only to care about the definition of layers, with statements, partial meth-
ods, etc., for the entities whose normal behaviors must be dynamically
adapted.

The resulting architecture is shown in Figure 5.9.

What is still to be designed is how these streams of data can be handled
by the application.

Considering the office example, the OverheatMonitoring context (List-
ing 5.9) creates a sort of view, which checks every thirty minutes the tem-
perature value, sampled by the proper group of sensors. In this case a real
stream is not created, but the context provides a simple value each thirty
minutes. However, if it is necessary to declare complex contexts, which are
always active and provide continuously data, it is unavoidable to work with
data streams.

Now, suppose that instead of setting the state of the air conditioner, the
OverheatMonitoring context is used by the adaptive part of the application.
For this purpose, it is not significant considering the temperature sample,
but the fact that the current temperature value, obtained at the moment of
the activation of the context, has reached the established threshold value.
In this way, contextual information is essentially passed to COP in form of
"events" (or sequence of events), so it becomes easier to handle.

The adaptive part can "see" the occurrence of events by the introduc-
Chapter 5. Reengineering and Extending PerLa and TReX

![Diagram of system architecture]

Figure 5.9: The system architecture

...tion of dedicated threads, whose task is to monitor the contexts. So when a context is declared with PerLa CL, a corresponding component must be implemented to monitor it and to activate the layers in the proper application modules.

In Fig. 5.9, which shows the system architecture, a box devoted to the **monitoring threads** is shown. In the example shown in Listing 5.9, a monitor thread checks only the part of the global context useful for the *Overheat-Monitoring* context activation; so the *OverheatMonitoring* context stream contains all and only the values related to temperature, humidity and air conditioner fan speed of the sensors located in the offices. If a sensor is not located in an office, its data are completely discarded in this context stream (Since the **EXECUTE IF** clause tells the system to check data gathered from offices’ sensors). If a sensor gathers more data than those required, the context unrelated data are discarded from the *OverheatMonitoring* context stream (another context stream must be declared if this data are still needed by another context).

This solution can be adopted in large and complex applications, where a considerable number of modules, belonging to different parts, need adaptations; however, it does not provide a real integration between the two treated actors. In fact, it is necessary to design dedicated components in order to provide the proper contextual data provided by PerLa to COP which, in turn, performs the behavioral variations for the application. These com-
5.2. Extending PerLa

<table>
<thead>
<tr>
<th>CREATE CONTEXT SmokeMonitoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE IF Location = 'office' AND Smoke = 'persistent'</td>
</tr>
<tr>
<td>REFRESH EVERY 1h</td>
</tr>
<tr>
<td>ON ENABLE (SmokeMonitoring) :</td>
</tr>
<tr>
<td>SELECT smoke</td>
</tr>
<tr>
<td>SAMPLING EVERY 10 m</td>
</tr>
<tr>
<td>EXECUTE IF EXISTS (smoke)</td>
</tr>
<tr>
<td>LOAD COP CONTEXT SmokeRisk</td>
</tr>
<tr>
<td>SET PARAMETER speed = 0</td>
</tr>
<tr>
<td>ON DISABLE (SmokeMonitoring) :</td>
</tr>
<tr>
<td>DROP SmokeMonitoring</td>
</tr>
<tr>
<td>DROP COP CONTEXT SmokeRisk</td>
</tr>
</tbody>
</table>

Listing 5.11: The integrated COP-PerLa approach

Components form a kind of intermediate layer that separates the information source and the adaptive part of the application.

An alternative solution may be to extend the PerLa CL with the option of directly launching scripts, programs, functions or even applications from the context queries, with a mechanism analogous to a Remote Procedure Call (RPC). In this way, a direct connection between context information and behavioral variations is created. The application designers can define a different independent behavior for each context, not needing to implement intermediate components for data and context management. The PerLa middleware takes the responsibility of every aspect related to context management. The CM acquires a decisive role: it must supervise the context activation state in order to avoid any possible inconsistent situation, which, in this scenario, could be impossible to handle.

In Listing 5.11, the new clauses LOAD COP CONTEXT/DROP COP CONTEXT have been introduced, in order to tell the system to activate/deactivate the related COP context (in this example, the COP code refers to Listing 5.8). The two clauses allow PerLa to execute external code (e.g., Java code) implemented following the COP paradigm, incrementing the whole PerLa CL and PerLa QL expressive power, introducing in them the possibility to perform operations that a SQL-like only approach cannot execute (and often also cannot express).

The activated COP context is then responsible of the relevant layer activation (e.g. SmokeLayer) and of managing all the related threads and procedures, while PerLa only manages the COP context activation/deactivation and handles the switch among different contexts. Notice that the COP context activation is not the only action performed by the system, since it firstly sends the air conditioner the “stop” (setting its speed to “0”) command using the default PerLa SET PARAMETER clause.
Chapter 5. Reengineering and Extending PerLa and TRex

5.3 Extending Tesla/TRex

As stated in Sect. 4.3, one of the main features that lacks in Tesla/TRex is the possibility to merge the data coming from streams with static data stored in a database or, more generally speaking, in a knowledge base.

In this section a solution to this problem will be given, allowing the TRex engine to semantically enrich events by gathering (part of) the necessary data from an RDF knowledge base, using them in the complex event generation process.

Extending the Tesla semantic becomes necessary in order to allow the system to enrich events with data stored in a standard \textit{RDFS} knowledge base (KB), producing more detailed complex events or avoiding to generate events based on the information contained in the KB that may invalidate some of the sequences computed by the event processor. The enrichment process must be defined at Tesla rule level, allowing the rule designers to define which complex events they are interested in and how the KB has to be queried in order to produce the static, enriching part.

Differently from some of the systems presented in Chapter 2, Sect. 2.2.2, the new rule syntax outlined in this section keeps the definition of how the event data and their semantic enrichment should be processed in a single place (i.e. each rule will define both how to handle the events coming from the stream and the data extracted from the KB). The processing model will thus integrate the KB data processing in the event stream processing, keeping the two different system components strictly related to each other.

A new operator called \textit{IN} has been introduced in Tesla, allowing the language to describe both the queries to be sent to the KB and the way in which the retrieved results must be used in order to enrich the results, or how to filter the event-only computed results ground on the KB retrieved data \footnote{While in this section the examples will focus on the filtering part, we will provide some examples of event enrichment in Chapter 7, Sect. 7.2.2}.

The operator also allows to specify parameters in the KB queries, so that it becomes possible to use data coming from events to reduce the amount of information to be retrieved from the KB. This is a very crucial feature, since a KB could be very large (many Terabytes) and thus not fit in system’s main memory but need to be stored on disk; in this scenario, the disk becomes an access bottleneck, which increases the response time, downgrading the whole system performance.

In Listing 5.12 the new semantics of the rules is shown: the first rule \textit{(TemperatureDecrease)} generates a new event each time a temperature in a
5.3. Extending Tesla/TRex

room of a building of type skyscraper decrease of 33% in 6 (or less) hours, while the second one (TemperatureDecreaseWithParam) shows how to add the parameter \textit{&floor} to the query adding to the KB query a filtering attribute of the Temp simple event.

```
define TemperatureDecrease(room: string)
from Temp(room = $r and value = $t) as T1 and last Temp(room = $r and value >= 1.33 * $t) as T2 within 6hrs
and $s in (SELECT ?room WHERE ?room isIn ?building. ?building hasType "skyscraper").
where room=T1.room
```

```
define TemperatureDecreaseWithParam(room: string)
from Temp(room = $r and value = $t) as T1 and last Temp(room = $r and value >= 1.33 * $t) as T2 within 6hrs
and $s in (SELECT ?room WHERE ?room isIn ?building. ?building hasType "skyscraper". ?floor is $floor)
where room=T1.room
```

Listing 5.12: Example of rules with KB operators

The new semantics require a revision of the TRex engine implementation since the disk access problem requires new structures to be implemented in the engine in order to properly handle the RDF processing. The proposed implementation provides some extensions to both the chosen SPARQL engine (RDF-3X) and to the TRex engine, in order to allow them to correctly interoperate and give consistent processing results.

5.3.1 RDF-3X SPARQL Engine

A SPARQL query engine has to be integrated in the TRex engine, in order to provide support to the execution of SPARQL queries declared in the IN statement of the Tesla rule. The chosen SPARQL engine is RDF-3X, a C++ SPARQL 1.1 \textit{RISC-style} engine implementation developed at the Max Plank Institute [92].

RDF-3X is open source and its code is freely downloadable from the developer website. However, the code present in the codebase provides only the implementation of a standalone application, while for a seamless integration with the TRex engine a dynamic library would be more appropriate. Several changes are required and have been implemented in order to obtain a library which provides a method that allows the user to submit a SPARQL query string and get back a set of valid results.

First of all, the engine does not provide a structure to store the result (the original implementation simply displays them on the standard output device). A \textit{Resultset} class has been added to the implementation in order to store the query results to be returned to the method invoker. The Resultset class is simply a wrapper of a vector of \textit{Result}, which is another class that
Chapter 5. Reengineering and Extending PerLa and TRex

has been added to the engine, providing a vector of Field that is the class that currently stores the result value; a Field has two attributes:

1. a value (that is a union which can store an integer or floating point or boolean or string value) and
2. a value type, an enum that identifies the type of the value that is stored.

The library then exposes the static method execQuery – through the interface class RDFQuery – that, given two strings (one including the path to the file containing the KB and one containing the SPARQL query) runs them and pack the results in a ResultSet, returning them to the invoking function.

The code is then compiled in a shared library, allowing it to be linked by the TRex engine adding to it the capability to run SPARQL queries. Some excerpts of the code of the Field, Result and ResultSet classes are shown in Listings 5.13, 5.14 and 5.15.

5.3.2 TRex Engine Extension

The TRex engine needs two important additions:

- first of all, TRex needs new structures to handle the SPARQL rule semantics
- then these new structures must be optimized in order to reduce the disk access time.

The structure proposed to handle the new KB related stuff is named KBPredicate (Listing 5.16) and simply stores db and query strings to which the predicate refers and a set of external parameters to the query (the ones coming directly from the event part of the system that have to be inserted in the query), if any.

Together with the new KBPredicate structure, many of the existent methods have been reimplemented in order to provide all the needed features to correctly handle the new predicates (e.g., allowing to declare the parameter - or the parameters - of the event that must be evaluated against the KB). It is now possible to declare a parameter between two states (as it is possible to do before) but also between a state and a KB query using the same method invocation, thus not requiring to revise the implementation of older rules, seamlessly integrate within the existing TRex system.

The next step is to provide a standard interface for the engine to handle the KB queries.
5.3. Extending Tesla/TRex

```cpp
/**
 * A field of the result
 */

typedef union fielddata {
    int i;
    float f;
    bool b;
    char s[STRING_LEN];
} fielddata_t;

enum ValT {
    INTV=0,
    FLOATV=1,
    BOOLV=2,
    STRINGV=3
};

class Field {
    fielddata_t value;
    ValT type;
    public:
        Field();
        virtual ~Field();
        // getters
        int getIntValue();
        float getFValue();
        bool getBValue();
        char* getSVValue();
        ValT getType();
        // setters
        void setIVValue(int v);
        void setFVValue(float v);
        void setBVValue(bool v);
        void setSVValue(char* v);
        // helpers
        int getSValueLen();
};
```

Listing 5.13: The KBPredicate structure

```cpp
/**
 * A result fo the query
 */

class Result {
    private:
        std::vector<Field> result;
    public:
        Result();
        virtual ~Result();
        std::vector<Field> getResult();
        void addElement(std::string e);
};
```

Listing 5.14: The KBPredicate structure
Chapter 5. Reengineering and Extending PerLa and TRex

```cpp
/**
 * The set of results
 */
class ResultSet {
private:
  std::vector<Result> results;
public:
  typedef std::vector<Result>::iterator iterator;
  ResultSet();
  virtual ~ResultSet();
  void addResult(Result r);
  std::vector<Result> getAllRes();
  void clearRes();
  iterator first() { return results.begin(); }
  iterator last() { return results.end(); }
};
```

**Listing 5.15: The KBPredicate structure**

```cpp
/**
 * A basic KB predicate
 */
typedef struct KnowledgeBasePredicate {
  Constraint *constraints; // Predicate constraints
  int constraintsNum; // Number of constraints in the predicate
  CompKind kind; // The kind of constraint
  std::string db; // The KB db reference for querying
  std::string query; // The KB query
  std::vector<ExtParameter> param; // External Parameters
} KBPredicate;
```

**Listing 5.16: The KBPredicate structure**
The class *QueryItem* (Listing 5.17) provides attributes and methods to handle the query processing at runtime, inside the whole rule processing performed by the *StackRule* class.

Notice that *QueryItem* provides an attribute *offset*, which helps the engine to deal with queries with large results. If a query resultset size exceeds the maximum number of allowed results for a query answer, the system simply scans the resultset using a sort of window-based procedure: first of all a LIMIT clause is appended to the query, using as limit the value provided by the offset *QueryItem* attribute, starting from offset 0. Hence, it process the query “limited” version an evaluates the results, incrementing the offset value adding the offset attribute value at each step, until a match has been found or the end of the resultset has been reached.

This allows to process queries that provide large resultsets also when those resultsets cannot be stored completely in main memory.

During the object instantiation phase, a check on the presence of external parameters in the query is performed (the parameter has to be retrieved from an event in the stream during the processing). If an external parameter is found, it is appended to the *params* vector. At processing time the rule processor that finds a KB-related rule first of all checks for external parameters in the *params* vector. If any, the processor tries to substitute it (or them) using the attribute(s) of the associated event. If such a substitution cannot be performed (i.e., the attribute cannot be found between the event parameters), the whole Tesla rule evaluates to *false* since it cannot satisfy the KB-related part. Otherwise, once all the external parameters have been substituted by the appropriate event-retrieved value, the system can send the query to the SPARQL engine and evaluate the results.

Once the processing of the rule has been performed, the processor resets the query parameters to their original values, avoiding inconsistent processing in further rule evaluations.

The processing of the SPARQL queries can be performed in two different points of the computation of the whole rule:

- If the query does not contain any parameter to be gathered from the event, the query can be processed before the whole Tesla rule is evaluated (and its results can be stored into the main memory); the query can also be processed at the end of the Tesla rule event part computation if it evaluates *true* (since it will be useless to perform the KB checks if we already know that the event part cannot be satisfied)

- If the query contains any parameter to be retrieved from an event, the query must be processed after the event has been received. In this
Chapter 5. Reengineering and Extending PerLa and TRex

```cpp
class QueryItem {

private:
    std::string db;   // The KB db reference for querying
    std::string query; // The KB query
    std::string originalQuery;  // The original query, in case substitution occurs
    std::vector<ExtParameter> params; // The KB query optional external parameters
    std::vector<std::string> fields; // The list of fields that the query returns
    CompKind kind;  // The kind of constraint
    ResultSet rs;  // The KB query retrieved results set
    int limit;  // The maximum number of results those can be stored in memory
    int offset; // Offset for scanning query results
    std::map<std::string, bool> replaceParams;  // Tell if a parameter has been replaced or not

    bool hasCachedResults(ResultsCache *qCache);
    ResultSet getCacheResults(ResultsCache *qCache);
    void storeResults(ResultsCache *qCache);

public:
    QueryItem(std::string & kb, std::string & q, std::vector<ExtParameter> p, CompKind k);
    ~QueryItem();
    bool runQuery(ResultsCache *qCache);
    ResultSet getResult();
    int getField(char * name);
    bool replaceExtParam(const std::string & pName, const std::string & pValue);
    bool needsReplace();
    bool hasMoreResults();
    std::vector<ExtParam> & getExtParams () { return params; }
    void resetExtParam();
};
```

[Listing 5.17: The QueryItem class]
5.3. Extending Tesla/TRex

case, to avoid useless disk access, the SPARQL query is executed if and only if the Tesla rule event part evaluates true.

In the first case, the results of the query can be used in the selection part of the Tesla rule, removing from the stacks the packets that do not satisfy the KB predicate, reducing the number of Tesla rules evaluation to be performed, improving the overall evaluation process performance.

Since the engine must deal with thousands of rule hits each second and must provide near real-time results, it is unfeasible to consider that, each time the query needs to access the KB, it must access the local disk, since this will increase the global response time too much.

To keep response time in a reasonable range, it is necessary to implement a cache structure that allows the system to store at least a part of the results (the most frequently used ones) of the queries in the main memory, solving the disk access problem.

5.3.2.1 Caching RDF Queries

The cache has been implemented in three steps: first of all we provide a simple cache, containing all the results of a single query. The second step provides a parametric cache, useful to avoid memory waste and disk access every time queries contain one or more parameters; the third, more complex cache will provide caching of some frequently accessed values of the KB. A combination of the three caches might be the best choice, in order to obtain the best overall performance.

Moreover, some specific sequences of queries can be predicted using instruments like statistical analysis and sequences mining (temporal pattern detection); those techniques allow us to understand if any specific queries activation patterns exist, and eventually preload their results in the cache in order to speed up the execution.

In order to remove old, no more used, results from the cache, we will simply consider the cache hit rate metric, removing the least used results from the cache.

**Caching Basic Frequently Used Resultset** The first basic cache implementation stores the result sets of frequently used queries in a main memory structure identified by the pair of strings describing the KB towards the query is issued and the query itself. Once the query is executed the first time, the results are retrieved from the disk and stored in the cache. The next time the query is executed, results are retrieved directly from the cache, avoiding the loss of time due to disk access. The cache is refreshed
Chapter 5. Reengineering and Extending PerLa and TRex

```cpp
class ResultEntry {
    public:
        ResultEntry();
        ResultEntry(std::string dbId, std::string qId, Resultset rs);
        virtual ~ResultEntry();
        std::string getDb();
        std::stringgetQuery();
        int getHitCount();
        bool isRemovable();
        bool isPartial();
        Resultset getRes();
    
    private:
        std::string db;
        std::string query;
        int hitCount;
        bool dontRemove;
        bool partialResult;
        Resultset results;
};

class ResultsCache {
    public:
        ResultsCache();
        virtual ~ResultsCache();
        void addEntry(std::string dbId, std::string qId, Resultset rs);
        void getEntry(std::string dbId, std::string qId);
        bool hasEntry(std::string dbId, std::string qId);
    
    private:
        std::vector<ResultEntry> cache;

#if MP_MODE == MP_LOCK
    pthread_mutex_t *mutex; // Mutex for synchronized methods
#endif
};
```

Listing 5.18: The cache structure

periodically verifying which are the most frequently used queries, dropping data related to the one which have lower hit frequency freeing up main memory space.

It is also possible to assign to each cache entry a flag, not allowing its deletion also if it is not frequent. This will allow to keep in main memory results related to queries whose answers may be requested in strict real-time, if they are related to crucial system tasks.

Looking at the code in Listing 5.18, we notice that the cache is a vector of entries, each one of those identified by the pair < kb, query > characterized also by the other parameters presented above. Once an entry is added to the vector, the hitCount is initialized to 0, and each time an entry is retrieved its related hit counter is incremented by 1.

124
5.3. Extending Tesla/TRex

Each cache entry provides also a flag stating if the cache contains all the results related to a specific query or not (i.e., the complete resultset is too wide to be stored in main memory), allowing the system to search for the result in the cache first and then maybe search on the disk storage (obviously skipping already considered cached results, appending a LIMIT clause to the KB query). In this situation, the caching algorithm tries to keep in main memory the subset of the original dataset containing most frequently used results. This subset however must be a contiguous subset of results within the complete KB query answer set, allowing to skip it if necessary, every time data have to be retrieved from disk.

The presence of partial results in the cache will reduce the overall cache performance, so it may be a better strategy to try to keep less, but complete, query answers in main memory.

Also notice that, in order to support the parallelism of the original TRex architecture, the cache must be thread-safe, so we must introduce a mutex to provide the synchronization mechanism necessary for the system to correctly behave when more than a detection thread is running.

While the described simple caching approach speeds up query processing reducing disk access request, it also presents several flaws:

- It may require very large main memory in order to be effective, increasing the system hardware requests
- If there are no frequent queries, the cache may be refreshed frequently, requesting lots of disk accesses and thus reducing the advantages of having a cache
- The cache is empty at system startup, and maybe very often bootstrapping will be needed to fill it
- The cache may contain duplicate entries if the same attributes are used by different queries
- This approach has very low performance if the query contains external parameters those values depend on the execution of the Tesla, event-oriented, part of the rule

Caching Parametric Queries  The basic version of the cache could be extended in a way that allows a smarter cache for those queries that contains external parameters.

During a preprocessing phase, each query that contains at least one parameter is normalized by means of the following rules:
Each parameter name “&x” is transformed into a new variable “?x”

Each new variable is appended to the SELECT clause

The cache will then contain unfiltered results for each parametric query, containing both the original required fields together with the auxiliary ones introduced for managing the filtering step. Each time one of those queries is used, a simple filtering is performed on the cache retrieving all and only the results belonging to the appropriate resultset.

Furthermore, the system keeps track of the hit rates of each specific query (i.e., logs the hit rates of queries after having substituted to parameters specific parameters values). The most frequently used parameter values related to query results are then directly stored in the cache, speeding up the processing since the filtering step has become useless.

Caching Frequently Accessed Data  Opposite to the resultset cache, the other possible approach is to store only part of the data the query engine use to compute a query result. These data are the ones most frequently used in different queries, so it is not possible to compute a single query result using only the cache data, but they can be used to reduce the size of the query resultset to be retrieved from disk, reducing the necessary time to collect them.

In order to produce consistent results, some properties have to be enforced by the elements in this particular cache. First of all consider the two SPARQL queries in Listing 5.19: during the queries static analysis step, the systems identifies ?b and ?c as two of the most used properties for the individuals ?x.

So, if it becomes necessary to store the pair <?b, ?c > in cache, either ?b or ?c or both of them has to be used as unique identifiers, in join with ?a or ?d, in order to produce the original resultset (it is a kind of referential integrity constraint). Otherwise it will not be possible to reconstruct the original resultset (if the previously stated integrity requirement is not respected, instead of join we will obtain a sort of cartesian product, generating a lot of false results).

This approach requires some new structures, also a bit more complex than the one presented in the previous paragraph, to help the computation.

The new structures are required in two phases of the query life:

1. Query loading
2. Query execution
5.3. Extending Tesla/TRex

```
SELECT ?a ?b ?c
FROM $foo
WHERE {
    ?x foo:prop2 ?b.
    ?x foo:prop3 ?c.
}

SELECT ?b ?c ?d
FROM $foo
WHERE {
    ?x foo:prop2 ?b.
    ?x foo:prop3 ?c.
    ?x foo:prop4 ?d.
}
```

Listing 5.19: Two queries examples

In the first phase, the query is loaded in the system and it must be statically analyzed in order to understand which are the classes, object properties and data properties involved in the query process. Once those elements have been clearly identified, each of them is stored in a catalog, which stores the class (or property) name, a counter that will be incremented each time a query requires the same element and the list of all the necessary items to reconstruct the original queries’ results in which the class (or property) appears. If this last fields lists all the classes (or properties) related to the queries containing the properties object of the analysis, this property should not be considered as a good cache candidate, since the engine must use the original queries to retrieve the queries results and no optimization will be achieved, without any advantages with respect to the original engine implementation without cache.

Considering the counter it is possible to understand which are the most common elements and store them in main memory first, after having verified that they allows an effective enhancement as described above. If there is empty space in main memory, also some of the least accessed elements may be stored.

Once part of the result is cached, the queries need to be rewritten in order to allow the system to retrieve cached data from the main memory and access to disk for the other ones. This rewriting step is aided by the catalog compiled in the previous steps, that tells which are the elements to be kept in the query header and which can be removed, and how to join the query and cache results to obtain the final result.

**Combined Caching Strategy** An hybrid approach is the right trade-off among the approaches presented above.
Chapter 5. Reengineering and Extending PerLa and TRex

The system starts using only caching frequently accessed data, retrieving and storing in the cache frequent itemset and rewriting queries.

Once the necessary data were computed (query hit rate statistics), the system can then retrieve the resultsets of the most frequent queries and store them in main memory allowing a faster processing of those queries, also considering parametric versions of them, if necessary.

So the system can easily and fast retrieves all the results to frequent issued queries directly from the main memory cache, while for other queries the cache may include only part of the result or no result at all.

Prefetching On the other side, applying statistical and sequence mining techniques to the system could lead to an improvement of the performances.

Consider the case in which the system keeps a log, in execution order, of all the rules triggered by the system.

Looking at the log it is possible to analyze the sequences of rules and understand if some frequent sequence exists. If any of them exists, you can prefetch in the cache all those resultsets related to the sequence, once you recognize that the sequence is being respected (e.g. a given number of the initial steps of the sequence has been already detected).

This techniques gives huge performances improvements when some of the queries involved in the sequences have large resultsets; in those cases the prefetching makes the results available in the cache just before they have to be used, avoiding the system to wait their retrieval from disk.

Cache Management The prefetching and the frequent query related management tasks requires a separate thread (or set of) to be in charge of handling those operations. Handling the cache management job during the processing phase would make totally ineffective the advantages of having a cache. The cache management thread will instead run independently and periodically, keeping the cache always up-to-date, increasing the whole processing performances.

5.3.3 Benchmarking the New Semantic Enhanced Engine

In order to understand if our decision to implement a cache is right, we perform some tests in order to evaluate if a speed increment corresponds to the introduction of the cache.

The test were performed on a AMD Phenom(tm) II X6 1055T six-core processor with 8 GB of RAM running Ubuntu GNU/Linux 14.04.
The first basic test is performed by having a four-processing-thread TReX instance running on the VM and having to access to a small subset of the Yago KB\(^6\) (around 3 GB KB).

The test rules set is generated before starting each run and it detects a standard event packet, checking if it respects some parameter stored in the KB. In addition, some of the queries require to extract some values from the packet attributes in order to use them in a filtering clause.

In each run we send from 200 up to 3000 packets and we register the execution times (mean, max, min and percentile time); the run is then repeated ten times to acquire different results.

The results of the test (considering the mean execution times) show that we have, on average, a speedup of the query execution. The speedup is very volatile because the queries are randomly generated at the beginning of the run while packets are randomly generated at the beginning of each step. So, depending on the kind of packets that have been generated, having a lot of packets with similar parameters value will show an higher performance improvement using the cache rather than when packets contain a lot of different parameters values, reducing the cache hit rate and, consequently, its performances.

Furthermore, since the cache management thread is not yet implemented in the prototype, most cache related operations are performed directly during rule processing, decreasing the overall cache performances.

The results obtained during the tests show an average speedup factor around 10x due to the RDF results caching, reducing the overall engine response time and making it closer to the pure event-processing engine response time.

On the other side the tests reveal that the simple cache implementation is not performant and stable enough to support a real-world application and needs to be improved and extended as described in the previous sections of this chapter.

\(^6\)http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/
CHAPTER 6

A parallel, Map-Reduce approach to association rule mining

Contents

6.1 Preliminary Concepts .................................................. 132
6.1.1 MapReduce and Hadoop ............................................ 132
6.1.2 NoSQL Databases .................................................. 133
6.1.3 The Frequent Itemset Mining Problem ......................... 133
6.2 The Idea – Problem Statement ....................................... 136
6.2.1 Lattice Aided Itemset Enumeration .............................. 137
6.2.2 Map-Reduce EClαT Design ........................................ 140
6.3 The framework .......................................................... 141
6.3.1 System Design ..................................................... 141
6.4 Results and benchmarks .............................................. 147
6.4.1 Experimental Results ............................................. 148
6.4.2 Final Remarks ..................................................... 152

This chapter presents MREClαT, a MapReduce implementation of the EClαT algorithm for frequent itemset mining. After having presented some preliminary concepts in Sect. 6.1, Sect. 6.2 describes the idea behind the algorithm while in Sect. 6.3 its implementation is shown.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

Sect. 6.4 exposes some benchmarks and results obtained during system tests.

As already noticed in the State of the Art (Chapter 2) while this work is under development other similar frameworks has been developed in other research groups (e.g. the ones developed by the Bart Goethals research groups described in [89]); it is a clear indication that we are moving in the right direction.

A small comparison between our approach and the one described in [89] will be shown in Sect. 6.4.

6.1 Preliminary Concepts

Before moving to the algorithm details, it is necessary to define some preliminary concepts about the systems involved (Hadoop, NoSQL databases), the programming paradigm adopted (MapReduce) and about the data mining operations, with a closer look on frequent itemsets extraction.

6.1.1 MapReduce and Hadoop

In order to take advantages from the Hadoop parallel execution, programs must be coded according to the MapReduce style. The framework can automatically decompose the computation in two phases (MAP and REDUCE) assigning each task to a different node (or set of nodes).

Several MapReduce phases can be chained, to obtain complex program behavior.

Each MapReduce phase begins processing some key-value pairs\(^1\); the programmer must specify both the type of the key and the type of the value field in the pair for every pair and must write the map and reduce functions.

The intermediate results of the map computation are then ordered and grouped by their key identifier (shuffle-sort is the name of this intermediate step) and then each reduce receives as input a vector containing some keys with all their values. After the reduce computation, we obtain another set of key-value pairs, containing the computation results.

Whenever it is necessary, it is possible to specify also a combine function. This function is basically an optimization function that allows to reduce the communication overhead between map and reduce functions. If a combine function is available, it is processed just after the map function, receiving in input the output of a map function and providing as output the aggregated results to the reduce function.

\(^1\) Except from the first map that could be used to parse the initial data set, like extracting data from a text file.
The framework however does not ensure the execution of the combine function which can be executed once, multiple times or even not at all.

The combine function is not suitable to every application scenario, since this particular function must ensure the associative and commutative properties.

6.1.2 NoSQL Databases

NoSQL databases allow to store non-critical data that do not have a complex structure (e.g., log data) ensuring low latencies in append and search operations.

The performance of these databases directly depends on the CAP Theorem [30–32], relaxing one of the requisite of a distribute database system:

**Theorem 6.1.1.** In a distributed system it is impossible to guarantee simultaneously that: i) all the nodes contain the same data at the same time instant (Consistency), ii) all the requests receive at least a notification of success or not (Availability) and iii) the system will operate also if a part of it fails or the data contained in a part of the system are lost (Partition tolerance).

It reasonably follow that each different NoSQL solution choose which of the three properties to discard, allowing each specific solution to be adopted in a different and particular context of use. Obviously it does not exist a perfect and standard solution, but is necessary to evaluate which solution best applies to each scenario: maybe NoSQL is not the right solution at all (e.g. bank accounts and their related transactions), a specific NoSQL solution may fit or multiple and different NoSQL solution may fit (and then other parameters have to evaluated to choose the correct alternative).

Critical data must be stored in a traditional RDBMS database since NoSQL system may not ensure Consistency or Availability properties, with the risk of unpredictable, unexpected and unrecoverable data loss.

6.1.3 The Frequent Itemset Mining Problem

After having defined $I = 1, 2, \ldots, m$ as a set of literals called items, $D$ as a set of transactions, where each transaction is a set $T$ of items such that $T \subseteq I$ and given that each transaction is associated with an unique identifier $tid$, then a transaction $T$ contains $X$, where $X$ is a subset of $I$, if $T \subseteq I$.

The number of items that compose the itemset defines its size: an itemset containing $k$ elements is said $k-\text{itemset}$. The support of an itemset $X$, denoted as $\sigma(X)$ is the number of transactions in which the itemset appears.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

An itemset is said to be frequent if its support exceeds a minimum support threshold (called minsup) set by the user and an itemset $X$ is closed in $D$ if no superset $Y \subseteq D$ of $X$ has the same support of $X$. If $X$ is closed in $D$ and is frequent in $D$, then $X$ is a closed frequent itemset.

As we have partially seen in 2.4, there exist three main algorithms for computing frequent itemset: Apriori, Partition and FP-Growth; all those algorithms are well-known alternatives for frequent itemsets extraction. In addition, we will consider EClaT, one of the state-of-the-art algorithm developed for this purpose.

Apriori Algorithm

This frequent itemset extraction algorithm requires multiple steps – and multiple dataset readings – in order to extract frequent itemsets.

In the first step the algorithm computes the support of individual elements and determine which of them are frequent. In the next steps, using the set of frequent itemsets computed in the previous steps, it generates a set of potential itemsets, called candidate itemsets. During the data scan, the support of each candidate itemset is computed to determine which ones are frequent; these frequent itemsets become the seed set for the next step. The process continues until there are no more frequent itemsets.

The Apriori algorithm [8] generates the candidates itemset in step $k$ only using the results of step $k-1$, without considering the transactions in the dataset. This approach reduces the number of candidates generated and tested during each execution step by improving the overall performance of the algorithm.

The Apriori property states that all the subsets of a frequent itemset are frequent. Consequently, the candidate itemset of $k$ elements can be generated from the union of frequent itemset of $k-1$ elements, removing those which contain at least a not frequent subset.

A critical analysis of the algorithm proposed in [69] highlights its limitations. The use of the Apriori property allows the algorithm to perform significantly better than all those algorithms (including AIS and SETM) those enumerate and compare all candidates. However, in situations where the frequent patterns are particularly long, or in the case of low minimum supports, the benefit obtained from the application of the Apriori heuristics is less relevant.
Partition Algorithm

The goal of the Partition algorithm [114] is to generate candidates and calculate their support in a single dataset reading. To achieve this goal it is necessary to choose a sufficiently small set of potentially frequent itemset.

The original dataset is then segmented into disjoint partitions which are analyzed separately during each iteration of the algorithm.

A partition \( p \subseteq D \) of the dataset \( D \) is any subset of transactions contained in \( D \); moreover, the partitions are disjoint: \( p_i \cap p_j = \emptyset, i \neq j \).

The local support of an itemset is the number of the transactions of the partition that contain the itemset on the the total number of transactions contained in the partition \( p \). Frequent itemset is defined as a local itemset where the local support exceeds the minimum support threshold set in the partition.

Using the properties of the frequent itemset support it is possible to conclude that an itemset is frequent in the dataset if it is frequent in at least one of the partitions. From this property it follows that the local frequent itemset generated from the first phase algorithm is a superset of the global frequent itemset, i.e. it is a superset of the original dataset frequent itemset.

The Partition algorithm performs its job in four stages:

1. Stage 1: all the processors independently generate the local frequent itemset

2. Stage 2: processors exchange the set of local frequent itemsets, at the end of this phase all the processors have all local frequent itemset

3. Stage 3: all processors calculate the support of each local frequent itemset in its own partition

4. Stage 4: The global support of each frequent itemset as sum of the local media is calculated by a processor

This processing pattern is very close to the MapReduce paradigm that proceeds by performing a set of independent tasks that read every split of the dataset (map function), followed by a task that collects intermediate results and produce total results (reduce function).

The Partition algorithm performs worst than the Apriori algorithm in the management of high support, while in other cases it shows significant performance improvements. One of the critical issues of the approach consists in the choice of how many partitions to use, a choice that also depends on the available hardware infrastructures.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

FP-Growth Algorithm

The FP-Growth algorithm [69] adopts a “divide et impera” strategy to avoid generating unnecessary candidates. Before mining, the algorithm compresses the transactions dataset in a prefixes’ tree called FP-tree (Frequent Pattern tree), a particular type of tree that incorporates all the necessary information for the mining process.

The FP-Tree is based on the prefix divided into a set of conditional FP-trees (representing a particular projection of the dataset) and mining is performed separately on each tree. The algorithm completes the search of frequent itemset performing only two scans on the data. The performance analysis undertaken by the authors emphasizes the good scalability of the algorithm, with respect to the decrease of the support.

The algorithm also scale linearly with respect to the number of transactions in the dataset and presents a better scalability than the Apriori algorithm on large datasets. It has some limitations, such as the time overhead introduced by FP-tree creation and the difficulty of managing the FP-tree in main memory as the size of the dataset under consideration increases.

Introduction to the EClaT Algorithm

The EClaT algorithm – of which we propose a MapReduce implementation – is based on a vertical representation of the dataset.

The itemsets are generated intersecting the lists of items that compose every transaction. Moreover, thanks to the vertical representation, the support of each itemset coincides with the cardinality of the list of its transactions. The execution of the algorithm can be decomposed into three stages:

1. 1-frequent itemset generation;
2. 2-frequent itemset generation;
3. k-frequent itemset generation;

The second phase can be expanded up to an appropriate $k - 1$ prefix, approach that allows to limit the size of the lists and then improve the overall performance of the algorithm. The algorithm authors themselves suggest to adopt the Apriori algorithm to generate the k-frequent itemset in the first $k$ stages.

6.2 The Idea – Problem Statement

In addition to the components presented in the previous chapters, a new data mining component, MR-Miner, has been added to the SuNDroPS system
architecture. This component implements \textit{MREClaT}, a Map-Reduce variant of the \textit{EClaT} algorithm described in [139], which is a frequent itemsets mining algorithm based on the notion of \textit{equivalence class}.

This algorithm will allow the system to reduce the processing time required to perform the user preferences extraction from the user usage logs gathered by the system. The output of this component is one of the main input of the \textit{Run-Time Context Manager} module, that aims to predict and select which is the actual user context, also by means of sensor-generated data and analyzing the user context log.

\subsection{Lattice Aided Itemset Enumeration}

In this paragraph we will give some definitions about the lattice theory useful for the logical definition of the problem, to allow its complete understanding; for further details please refer to [140].

Given a set \( P \), an \textit{order relation} on \( P \) is a binary relation \( \leq \) such that for each \( X, Y, Z \in P \), the relation has the following properties:

1. Reflexivity: \( X \leq X \)
2. Antisymmetry: if \( X \leq Y \) and \( Y \leq X \) then \( X = Y \)
3. Transitivity: if \( X \leq Y \) and \( Y \leq Z \) then \( X \leq Z \)

A set \( P \) on which the relationship \( \leq \) is valid it is said to be an \textit{ordered set}.

Given an ordered set \( P \) with \( S \subseteq P \), an element \( X \in P \) is said \textit{upper bound} (respectively \textit{lower bound}) of \( S \) if \( s \leq X \) (\( s \geq X \)) for each \( s \in S \). The lowest upper bound of \( S \) is denoted by \( \forall S \) or with \textit{sup}, while the greatest lower bound of \( S \) is denoted by \( \wedge S \) or with \textit{inf}. The greatest element in \( P \), indicated with \( \top \), is said \textit{maximum} while the lowest element of \( P \), indicated with \( \bot \), is said \textit{minimum}.

Given an ordered set \( P \) and \( X, Y, Z \in P \), we say that \( X \) is \textit{covered by} \( Y \), denoted by \( X \sqsubseteq Y \), if \( X \leq Y \) and \( X \leq Z \leq Y \) imply \( Z = X \), i.e., there is not an element of \( Z \in P \) such that \( X \leq Z \leq Y \).

Given an ordered set \( L \), \( L \) is said \textit{sup semilattice} (respectively \textit{inf}) if the transaction is defined as \( X \lor Y \) (\( X \land Y \)) for each \( X, Y \in L \). \( L \) is a \textit{lattice} if both a semilattice \textit{sup} and a semilattice \textit{inf} are defined in it, i.e., if the operations \( X \lor Y \) and \( X \land Y \) are defined for all the pairs \( X, Y \in L \). The lattice \( L \) is said to be \textit{complete} if \( \forall S \) and \( \wedge S \) are defined for all the subsets \( S \subseteq L \).

An ordered set \( M \subset L \) is a \textit{sublattice} of \( L \) if \( X, Y \in M \) implies \( X \lor Y \in M \) and \( X \land Y \in M \). Given the ordered set \( S \), the power set of \( S \) denoted
by \( \mathcal{P}(S) \), is a complete lattice where conjunction and meet operations are respectively the union \( \bigvee \{ A_i \mid i \in I \} = \bigcup_{i \in I} A_i \) and intersection \( \bigwedge \{ A_i \mid i \in I \} = \bigcap_{i \in I} A_i \). The maximum of \( \mathcal{P}(S) \) is \( T = S \), while the minimum is \( \bot = \emptyset \). For each \( L \subseteq \mathcal{P}(S) \), \( L \) is said lattice of sets if it is closed respect to finite union and intersection operations, i.e., \((L; \subseteq)\) is a network where the order relationship is specified by the subset operation \( \subseteq \), specified as \( X \lor Y = X \cup Y \) and \( X \land Y = X \cap Y \).

Fig. 6.1 shows the lattice defined by the powerset for the set \( I = \{ A, C, D, T, W \} \); the set of items in the dataset is represented on the left part and the frequent itemset are circled in gray on the right side. It must be observed that the set of all frequent itemsets form a semilattice w.r.t. the operation of meeting, i.e. for any pair of frequent itemset \( X \) and \( Y \), \( X \land Y \) is frequent.

Moreover, we note that the frequent itemset does not form a semilattice compared to the operation of conjunction: stating that \( X \) and \( Y \) are frequent itemset does not imply that \( X \lor Y \) is frequent.

### 6.2.1.1 Supports computation

A lattice \( L \) is said to be **distributive** if for each \( X, Y, Z \in L \) the notation \( X \land (Y \lor Z) = (X \land Y) \lor (X \land Z) \) is valid.

Let \( L \) be a lattice with minimum \( \bot \). \( X \in L \) is an **atom** if \( \bot \sqsubseteq X \); i.e., \( X \) covers \( \bot \). The set of atoms of \( L \) is denoted by \( \mathcal{A}(S) \).

A lattice \( L \) is said to be a **boolean lattice** if:

1. it is distributive
2. it contains the \( \bot \) element
3. every element $X$ of the lattice has a complement

Consequently, the lattice formed by all the powersets $\mathcal{A}(T)$ on the set of items $I$ is a boolean lattice.

Each atom in the lattice is associated with the list of identifiers of the transactions (tid) in which it appears, denoted as $\mathcal{L}(X)$.

If $L$ is a finite boolean lattice and $X \subseteq L$ then $X = \bigvee \{ Y \in \mathcal{A}(L) | Y \leq X \}$. This statement is valid in every boolean lattice.

In other words, each element of the boolean lattice can be computed applying the operation of conjunction to a subset of the atoms. Since the lattice of the powerset $\mathcal{P}(I)$ is a boolean lattice where the operation of conjunction corresponds to the set union operation it is possible to state that for each $X \in \mathcal{P}(I)$, if $J = \{ Y \in \mathcal{A}(P) | Y \leq X \}$ then $X = \bigcup_{Y \in J} Y$ and $\mathcal{L}(X) = \bigcap_{Y \in J} \mathcal{L}(Y)$.

It means that any itemsset can be obtained by applying the operation of conjunction to the atoms of the lattice and its support is calculated as the intersection of the tid-list of atoms. The previous statement could be generalized as follows: for each $X \in \mathcal{P}(I)$ if $X = Y \cap_{Y \in J} Y$ then $\mathcal{L}(X) = \bigcap_{Y \in J} \mathcal{L}(Y)$.

It is then possible to formulate each itemsset applying the merge operation to a set of itemssets belonging to $J$, then the support of the itemsset is computed as the intersection of the tid-list of elements of $J$. In addition, it is possible to determine the support of any $k$-itemsset by intersecting the tid-list of all pairs of its subsets of size $k - 1$.

In the end, given two itemssets $X$ and $Y$, if $X \subseteq Y$ then $\mathcal{L}(X) \supseteq \mathcal{L}(Y)$.

The last statement means that if $X$ is a subset of $Y$, the cardinality of the tid-list of $Y$ (i.e., the support) is less than or equal to the cardinality of $X$. Consequently, the cardinality of intermediate tid-lists decreases during the construction of the lattice.

### 6.2.1.2  Prefix-based Lattice Decomposition

The enumeration of all frequent itemsset through a search procedure in the complete lattice is a very expensive process in terms of memory. In addition, it is possible that, during the computation of the frequent itemssets, the tid-list reaches a size that exceeds the maximum size that can be handled in main memory.

To solve the problem of the limited resources, a partition of the original lattice small enough to be processed in memory independently is considered. The partitioning is possible since the lattice hold the following properties.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

Given a set \( P \), an equivalence relation on \( P \) is a binary relation \( \equiv \) such that for each \( X, Y, Z \in P \) the relationship has the following properties:

1. Reflexivity: \( X \equiv X \)
2. Symmetry: if \( X \equiv Y \) then \( Y \equiv X \),
3. Transitivity: if \( X \equiv Y \) and \( Y \equiv Z \) then \( X \equiv Z \).

Through the equivalence relation it is possible to partition the set \( P \) in disjoint subset said equivalence classes. The equivalence class of a element \( X \in P \) is denoted by \([X] = \{X \in P | X \equiv Y\}\).

A function \( p \) representing the prefix of an equivalence class, is defined as \( p : \mathcal{P}(X) \times \mathbb{N} \rightarrow \mathcal{P}(I) \) with \( P(X, k) = X[1:k] \) and \( k \) is the prefix of \( X \).

Considering all the previous definitions, an equivalence relation \( \theta_k \) on the lattice \( \mathcal{P}(T) \) is defined as:
\[
\forall X, Y \in \mathcal{P}(I), X_{\theta_k} Y \iff p(X, k) = p(Y, k).
\]
It means that two itemsets belong to the same class if they share a common prefix of length \( k \). For this reason, the relation \( \theta_k \) is said equivalence relation based on the prefix.

It is also possible to state that every equivalence class \([X]_{\theta_k}\) introduced by the equivalence relation \( \theta_k \) is a sublattice of \( \mathcal{P}(I) \); as a consequence, each equivalence class defined by \([X]_{\alpha}\) is itself a boolean lattice.

6.2.2 Map-Reduce EClaT Design

The first step of the algorithm involves the computation of the 1-frequent and 2-frequent itemsets. Thanks to the adoption of the vertical representation, \( \mathcal{F}_1 \) can be computed in a single dataset scan and the support count is achieved by simply sliding the tid-list of an item and incrementing a counter. On the other side, the process of 2-frequent itemset generation is computationally very expensive and it requires at least \( N/2 \) dataset readings.

Moving from this basic consideration, it is suggested to adopt a step of pre-processing for the calculation of \( \mathcal{F}_2 \) to compute the pairs of frequent items, arguing that the cost of computation can scale over the number of times that the dataset is used for mining. Alternatively, it is possible to adopt the horizontal representation for the computation of \( \mathcal{F}_2 \).

In the next step the sublattices are processed individually in memory using a bottom-up search strategy.
Finally, a recursive call to the procedure is made, providing as parameters the frequent itemset results at the current level. The procedure is repeated until all the frequent itemset have been listed.

**Preliminary Observations**

The performance analysis shows that the algorithm is more efficient than the Apriori [6] and Partition [114] algorithms in all the test and evaluations performed.

The algorithm takes advantages of the reduced number of dataset readings needed to perform the whole computation and of the simplicity of the structures and operations used during the process.

Another version of EClat, dEClat [141], is based on a new technique of vertical representation, named *diffsets*, which drastically reduces the memory required to manage intermediate data. The new solution avoids to save the entire tid-list of every member of the equivalence class, building a list of difference sets keeping only track of the differences between the lists of each member of the class and the prefix list.

The differences are then propagated from the root node to all levels of the lattice; the root node can be represented by a diffset compared to the empty prefix or a tid-list.

As it will be described later in Sect. 6.3, the MapReduce implementation proposal complies with the execution of three sequential phases as in the traditional algorithm. The proposed algorithm implementation adopts a horizontal representation of the dataset for the early stages and switches to the vertical representation only after having generated the equivalence classes.

### 6.3 The framework

#### 6.3.1 System Design

The proposed implementation refers to the original MapReduce paradigm aiming at fully exploiting its power; other MapReduce implementation of the algorithm adopt some “tricks” in order to try to achieve better performances, not fully complying with the original paradigm.

#### 6.3.1.1 Representation of the dataset adopted

The MapReduce framework splits the input files automatically and transparently to the user and to the developer, in such a way that the exact location of the record on the available nodes it is not known. The default
behavior is to split the input in equally sized chunks and to distribute them on the available nodes.

For this reason, the algorithm uses a horizontal representation of the data for the early stages and switches to the vertical representation only after having computed a prefix of an appropriate length. This horizontal representation is necessary for the first steps of the execution since a vertical representation of the original dataset would have required a transition to the horizontal representation for the computation of $F_2$, and for all subsequent stages where the prefix $\theta_2$ will not allow in memory storage due to the limited size of the memory respect to the number of data to be stored.

The transition from one representation to another in MapReduce is easy, since it is enough to start some MapReduce tasks on the dataset to manipulate the representation.

In particular, to move from a vertical representation to a horizontal one, the map function emits output pairs $<tid, item>$ for each transaction contained in the tid-list. Then a reduce function aggregates the intermediate results allowing to obtain transactions $<tid, \{list of items\}>$ pairs.

Similarly, to move from a horizontal representation to a vertical one a map function runs on the original dataset outputting pairs $<item, tid>$, corresponding to all the items that appear in the transaction. Then a reduce function aggregates the results allowing to obtain $<item, \{list of tid\}>$.

### 6.3.1.2 Outline of the algorithm

Figure 6.2 presents a description of the MREClaT algorithm execution steps, considering a prefix of length 2 as suggested by the original authors of the algorithm in a distributed environment [139].

The algorithm performs the frequent itemset extraction task in four phases:

1. **Initialization**: Computation of the list of frequent itemset $F_1$, performed separately on each node, with a final synchronization step.

2. **Equivalence Classes Generation**: Computation of the frequent itemset list $F_2$, performed separately on each node, with a final synchronization step. All the nodes generate a global list of candidates $C_2$, starting from $F_1$.

3. **Processing Classes**: Distributed computing of frequent itemset of size $k > 2$. Each node processes independently one (or more) equivalence class assigned to it.

4. **Result Aggregation**: Aggregation of the results into a single output file.
Figure 6.2: MREClaT algorithm outline
Once the distribution of the classes to the nodes has been completed, the traditional EClat algorithm is then applied. To use the dEClat algorithm version (the one based on diffsets), it is enough to simply replace the function that creates the tid-list with the function that generates the diffsets and to adopt the corresponding method for computing the itemsets’ supports; these are minor changes that allow to switch from one version of the algorithm to the other one with a very small effort.

The algorithm MapReduce implementation shows many similarities with the distributed version mentioned before, except for the first initialization phase.

### 6.3.1.3 Detailed analysis of the algorithm

In the next paragraphs a more detailed analysis of the algorithm implementation will be presented, referring to how each of the previously introduced four stages is implemented in term of map and reduce jobs.

**Initialization** The first stage map function takes in input a subset of the transactions contained in the original dataset as \(< tid, \{list of items\} >\) pairs and it analyzes every transaction producing in output a \(< item, 1 >\) pair for each item contained in the transaction. It is essential to point out that transactions are processed independently.

The combine function (not shown in Fig. 6.2) aggregates the results of each map; i.e. receiving the vector \(< item, 1, 1, 1 >\) it computes a pre-aggregate node result \(< item, 3 >\). The combine function is applied to the results of each map, aggregating pairs on the basis of their key. It is possible to adopt this optimization since the sum operation ensures commutative and associative properties.

The reduce function computes the sum of the values for each input pair vector, checking if the support of the item exceeds the minimum support threshold required and produces as output the pair \(< item, support >\). All the itemsets those exceed the minimum support threshold are then saved in \(\mathcal{F}_1\), while all the others are discarded. The list is then forwarded to the next step, where it is used to generate the list \(\mathcal{C}_2\) of candidate frequent itemsets of size 2.

**Equivalence Classes Generation** Each map computes the complete list \(\mathcal{C}_2\) of the candidates itemset of size 2 starting from \(\mathcal{F}_1\). The map function receives in input a set of transactions \(< tid, \{list of items\} >\) from the same input dataset of the previous phase, which already resides in the distributed filesystem.
6.3. The framework

The function analyzes each transaction producing in output the pair \( < 2 - \text{itemset}, tid > \); the analysis is performed by comparing all the pairs of items in the transaction with the couples belonging to the list \( C_2 \). The transactions are processed independently, without constructing any additional global structure.

The reduce function performs an aggregation of the partial results, verifying that the cardinality of the final result exceeds the minimum support threshold. The items that exceed the threshold are part of the output as pairs \( < 2 - \text{itemset}, \{tid - list\} > \). From this point on the algorithm switches to the vertical representation.

Processing of classes In this phase the map function takes as input the output pairs produced by the previous step reduce function.

This step map function simply reads individual records and produces \( < \text{prefix}, \{2 - \text{itemset}, \{tid - list\}\} > \) pairs. This allows the shuffle-sort phase to randomly assign available prefixes to the next reduce. The procedure for the assignment of the prefix is equivalent to assign each node an equivalence class with prefix \( \theta_k \) of length 2.

Moreover, each node receives all the data necessary for the mining phase of each equivalence class assigned to it, or all 2-itemsets which share the prefix together with their corresponding tid-list.

The reduce function receives the output pairs from the previous map and begins the traditional EClaT algorithm on the equivalence class. The algorithm proceeds using the intersection operation between the lists and computes recursively the frequent itemset of increasing size, adopting a bottom-up strategy.

The reduce function produces as output \( < k - \text{itemset}, \text{support} > \) pairs.

Aggregation This last stage is necessary because each reduce produces an output file with each node results and it is necessary to merge them in a single result file. In this step a simple concatenation of all the results is performed, packing all the nodes’ output in a single file.

Further optimization adoptable

A pre-filtering step could lead to an optimization of the execution of MRE- ClaT. A typical solution will be to use the bloom filter pattern [23].

This pattern uses an additional map function at the beginning of the execution, which performs the filtering by allowing all the next MapReduce jobs to work only on the pre-filtered data.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

In this case, it is necessary to introduce a map function on the set $F_1$ which analyzes the transaction input and discards transactions that do not contain any frequent element$^2$. The adoption of this technique allows to decrease the number of comparisons performed by the map to generate the frequent itemset in the next steps.

The effectiveness of the pre-filtering step is strongly bound to the context and to the dataset that has to be processed. On a generic dataset, the advantages obtained during the second phase of implementation is limited and they do not justify the overhead of running an additional step.

The implemented version of MRECluT proposed in this work and used for the tests does not include the pre-filtering step, but is possible to insert it – if needed – in some contexts with a small implementation effort.

Extension of the initial prefix  The algorithm performance analysis that will be shown in Sect. 6.4 highlighted some issues in managing low supports on small, dense datasets.

It has been experimentally verified, that for a prefix of length 2 the size of the tid-list of the greatest equivalence class is too large to be handled in memory.

To improve the performances, an extension of the initial prefix has been introduced allowing to obtain smaller equivalence classes, applying $k$ steps of the Apriori algorithm for computing the set $F_k$ of frequent itemset of size $k$, during the classes generation phase. The other steps of the algorithm remain unchanged.

The transition to the vertical representation is then deferred until the start of the elaboration phase of the classes, using the horizontal representation during the first steps.

The equivalence classes generation phase is performed using Apriori as follows: the map function $I$ takes as input a set of transactions $<tid, \{list of items\}>$. The input is considered to be the same for all the steps and it is stored in the distributed filesystem.

The map function then scans each transaction individually producing in output some pairs $<n+1-itemset, 1>$ comparing the items of the transaction with the elements of the list $C_{n+1}$.

The reduce function computes the sum of the values of each frequent itemset pair in input and it produces $<n+1-itemset, support>$ pairs in output, if the support count exceeds the required minimum support threshold, discarding, the itemsets that do not exceed the required threshold, while saving the others in $F_{n+1}$.

$^2$Items that have a support below the required minimum support threshold.
6.4 Results and benchmarks

The list is then forwarded to the next step, which is used to generate $C_{n+2}$, the list of candidate itemset of size $n + 2$.

In the last step, the Apriori algorithm computes the frequent itemsets together with their tid-list, as it does in the original algorithm implementation presented before, then it switches to the vertical representation.

The main objective of this extension is to adapt the idea already proposed in the SPC algorithm [77] described in Chapter 2, Sect. 2.4.

6.4 Results and benchmarks

In the following pages some results about the comparison between the MREClAT algorithm and BigFIM algorithm (presented in Chapter 2, Sect. 2.4.2) are presented. The evaluation was performed on a small cluster of virtual machines.

Comparison between MREClAT and BigFIM The following benchmarks compares the two algorithms MREClAT and BigFIM [89], described in Sect. 2.4.2. A comparison between MREClAT and distEClAT is not proposed since the second algorithm is more easily classifiable as a MapReduce extension of the Partition algorithm than as an alternative MapReduce implementation of the EClAT algorithm. Moreover, there were no significant differences between BigFIM and distEClAT considering the implementation level, except from the first computation phase of the k-frequent itemset.

In addition, distEClAT has some critical issues:

- The second step can be infeasible, since a mapper may need the entire dataset to construct the 2-frequent itemset, or the tid-list of an itemset could frequently exceed the size of main memory available on the node.

- The third step of execution could have a high communication cost since it is possible that the mapper requires almost all of the dataset to make the mining of the sublattices.

Differences between MREClAT and BigFIM are summarized in Tables 6.1 and 6.2 pointing out the main characteristics of the two algorithms.
Chapter 6. A parallel, Map-Reduce approach to association rule mining

<table>
<thead>
<tr>
<th>MREClAT</th>
<th>BigFIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key elaborations performed in the reduce function</td>
<td>Key elaborations performed in the map function</td>
</tr>
<tr>
<td>Horizontal data representation</td>
<td>Horizontal data representation</td>
</tr>
<tr>
<td>Standard EClAT algorithm implementation</td>
<td>Diffset based EClAT algorithm implementation</td>
</tr>
<tr>
<td>Predetermined prefix length (k=2)</td>
<td>No predetermined prefix length set</td>
</tr>
<tr>
<td>Frequent Itemset extraction</td>
<td>Extraction of a superset of the frequent itemset</td>
</tr>
<tr>
<td>Binary-offset-based tid</td>
<td>Progressive tid</td>
</tr>
</tbody>
</table>

Table 6.1: MR-EClAT and BigFIM comparison

<table>
<thead>
<tr>
<th>MREClAT</th>
<th>BigFIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-frequent itemset generation</td>
<td>k-frequent itemset generation and global prefix tree computation</td>
</tr>
<tr>
<td>Prefix generation and computation of the ( k+1 ) \textbf{itemsetrelatedtid} \textbf{list}</td>
<td>Prefix-free-based ( k+1 )-itemset generation</td>
</tr>
<tr>
<td>EClAT execution start (in the reduce function)</td>
<td>EClAT execution start (in the map function)</td>
</tr>
</tbody>
</table>

Table 6.2: MR-EClAT and BigFIM execution steps comparison

6.4.1 Experimental Results

For the benchmarks a small virtual cluster of two virtual machines (VM) equipped with the CentOS 6.4 operating system and 5 GB of RAM for the master machine and 4 GB RAM for the slave machine has been used.

A virtual machine acts as the master node, while the other acts as slave node.

The physical host is a 3.30 GHz Intel Xeon quad-core processor with 16 GB of RAM. Hadoop version 1.2.1 has been installed on each VM, configured using the default options.

The infrastructure available during the tests did not allow the use of large dataset; however, the experimental results allow to suppose that the algorithm should have good performances in real applications.
6.4. Results and benchmarks

6.4.1.1 Analysis of performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#items</th>
<th>#transactions</th>
<th>Transaction size</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>accidents</td>
<td>572</td>
<td>340184</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>mushroom</td>
<td>119</td>
<td>8124</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>pumsb</td>
<td>2113</td>
<td>49046</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>T10I4D100K1</td>
<td>870</td>
<td>100000</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.3: Datasets characterization

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MinSup %</th>
<th>1-FI</th>
<th>2-FI</th>
<th>tot FI</th>
<th>$t_{exec}^1$</th>
<th>$t_{exec}^2$</th>
<th>$t_{exec}^3$</th>
<th>$t_{exec}^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>accidents</td>
<td>60</td>
<td>21</td>
<td>124</td>
<td>2181</td>
<td>33</td>
<td>248</td>
<td>60</td>
<td>342</td>
</tr>
<tr>
<td>mushroom</td>
<td>10</td>
<td>56</td>
<td>762</td>
<td>575898</td>
<td>22</td>
<td>34</td>
<td>60</td>
<td>112</td>
</tr>
<tr>
<td>pumsb</td>
<td>90</td>
<td>20</td>
<td>140</td>
<td>2862</td>
<td>26</td>
<td>65</td>
<td>34</td>
<td>126</td>
</tr>
<tr>
<td>T10I4D100K10.0005</td>
<td>869</td>
<td>1575422204337</td>
<td>22</td>
<td>45</td>
<td>143</td>
<td>232</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: MREClaT performances (times reported in seconds)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#transactions</th>
<th>Transaction avg size</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset1</td>
<td>92911</td>
<td>15</td>
</tr>
<tr>
<td>dataset2</td>
<td>928744</td>
<td>15</td>
</tr>
<tr>
<td>dataset3</td>
<td>908751</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 6.5: Syntetic datasets characterization

The analysis refer to four synthetic dataset (accidents, mushroom, pumsb and T10I4D100K1) whose characteristics are specified in Table 6.3. Execution results are summarized in Table 6.4 and in Fig. 6.3. The results reported show the strengths of the algorithm that is particularly well suited to datasets whose transactions are composed of a small number of items.

The algorithm, in fact, does not perform well on datasets where the average size of transactions is big. Therefore, there is the necessity to improve
the research phase of the 2-frequent itemset that is critical in the case of long transactions, as evidenced by the results obtained on the accidents dataset.

The management issue raised for the transaction size is due to the small size of the main memory assignable to each task available in the small test cluster.

Some scalability tests for the algorithm MRECIaT have been carried out using a synthetic dataset generator. Increasing size datasets – in terms of number of tuples – have been generated; those datasets use transactions of reduced dimensions. The experiment shows that the algorithm, using limited resources such as the ones provided by the test cluster, can handle datasets up to a million transactions keeping more than reasonable execu-
tion times.

Datasets composed of long transactions do not benefit from distributed execution since they need to generate larger sublattices requiring, even in the distributed environment, the allocation of adequate resources.

6.4.1.2 Initial Prefix Related Scalability

To verify the scalability of the algorithm with respect to the prefix extension, *accidents* and *pumsb* datasets were used. The experiment was carried out by extending the prefix from a two elements length to a three elements length, using three Apriori algorithm steps before starting the EClAT algorithm. The results confirm that extending the initial prefix allows to reach lower supports on datasets consisting of transactions with a large number of items.

The graphs in Fig. 6.4 show the results in terms of execution time and in the number of frequent itemset generated. The prefix length increment, and the benefits that derives from it, allowed to reach a minimum support threshold of 90% on the *pumsb* dataset, and a minimum support threshold around 60% on the *accidents* dataset.

![Graphs showing execution time and itemset generation](image)

**Figure 6.4:** Prefix extension benchmarks

The experimental approach emphasizes that the extension of the prefix is effective and allows to manage transactions consisting of a large num-
Chapter 6. A parallel, Map-Reduce approach to association rule mining

ber of elements, reaching smaller supports in a context where resources are limited, such as the test cluster configuration. The execution times are high due to the overhead introduced by the execution of a large number of jobs and to the operations carried out on the prefixes tree, highlighting the need to implement a prefixes tree structure that allows insertion and search operations to be extremely efficient.

6.4.2 Final Remarks

The experiments allowed to state the actual applicability of the algorithm on datasets with heterogeneous characteristics.

Progressively increasing the size of the synthetic datasets makes also possible to estimate the scalability of the algorithm with respect to number of transactions in the processed dataset.

The results obtained are more than satisfactory when compared with the results experienced by BigFIM and distEClA{T algorithms applied to the same dataset (Fig. 6.3): in almost the same execution time, the MREClA{T algorithm generates a significantly higher number of patterns.

By the way, to correctly analyze and benchmark the algorithm, it is necessary to perform a more extensive experimentation on very large data sets with the support of an ad-hoc and more performant infrastructure.
CHAPTER 7

The Green Move Project: an Application Example

Contents

7.1 The Green Move Project ........................................ 153
7.2 SuNDroPS in Green Move ...................................... 155
  7.2.1 Context-aware Reservations ............................... 155
  7.2.2 Green Move Dynamic Apps ............................... 158
  7.2.3 GMAvisor .................................................... 160
  7.2.4 Context-aware Sensors .................................... 163

This final chapter presents some applications involving components of the SuNDroPS framework. The chosen application scenario is the Green Move project, in which Context-Aware PerLa, PervAds and TRex have been used in different system modules to construct the Green Move system.

7.1 The Green Move Project

As presented in [21], the main elements of the Green Move platform, shown in Fig. 7.1, are the Green Move Center, which coordinates the system, the
Green e-Boxes, which constitute the interface between the vehicles and the rest of the system, and the users’ smartphones, on which the Green Move client app is installed. The Green Move Center (GMC) coordinates the activities of the Green Move system and offers services such as user and vehicle registration, vehicle reservation/acquisition/release/monitoring, and so on. To retrieve and distribute information among the managed vehicles, the GMC includes the T-Rex Complex Event Processing (CEP) engine [51] [50].

The Green Move platform uses CEP technology to both monitor the vehicles and send them configuration commands. The PerLa language and middleware [119], part of the SuNDroPS system like the SemTRex CEP engine can be used together with it since it offers native context management capabilities [118].

Each Green Move vehicle is equipped with a device, the Green e-Box (GEB), through which it interacts with the overall system. From a technical point of view, the only requirement for a vehicle to be introduced in the Green Move system is to have a GEB; this ensures that the interaction between vehicle and system occurs according to standardized protocols. This allows for the possibility of adding to the system vehicles that are heterogeneous not only in their types, but also in their ownership [10]. The GMC communicates with GEBs through 3G channels to manage the fleet. The same channel is used by GEBs to send to the GMC vehicle data – which are distributed at regular intervals as T-Rex events – such as diagnostic in-

![Figure 7.1: Schematic view of the overall architecture of the Green Move system.](image-url)
formation, usage statistics, and trip data (e.g., current GPS position, speed, state of charge). GEBs run the Android operating system, which executes the Green Move vehicle app. This app is the software interface between the vehicle and the rest of the system; in addition to executing core functions such as data retrieval and actuation of commands such as open/close doors, it provides hosting facilities for dynamic applications.

The interaction of Green Move users with the system occurs through their smartphones, on which the Green Move client app must be installed. The app communicates with the GMC through a WiFi or 3G channel to reserve vehicles and to retrieve the electronic key that is necessary to access the vehicle. This key is exchanged between the Green Move client app and the GEB through a Bluetooth or NFC channel; it is used to send commands from the user’s smartphone to the vehicle, such as open/close the doors (if present), and enable/disable the drive. By using a direct (Bluetooth or NFC) link between the user’s smartphone and the GEB, this communication can occur at any time, even when there is no 3G/WiFi connection available between the GEB and the GMC. The same connection can be used by the GEB to send data to the users’ smartphones, such as customized advice of various kinds (e.g., commercial).

The architecture outlined above is the basic layer on the top of which all the functionalities of the GM system are developed, using SuNDroPS modules.

### 7.2 SuNDroPS in Green Move

There exist many different applications of SuNDroPS components in Green Move; the most important ones are: the Context-aware Reservations handling subsystem, the Green Move Dynamic Apps infrastructure, the GMAdvisor application and Context-aware sensors.

#### 7.2.1 Context-aware Reservations

This component supervises all the sharing services provided through the system. For instance, the reservation process provides functionalities like checking available vehicles, foreseeing to vehicles’ scheduling, proposing additional services to the users, all taking into account their actual context and historical contextual data.

This component implementation has been divided into six ASP modules, each dedicated to a specific task:

- **CDT definition** represents the CDT in ASP (see an excerpt in Listing 7.1)
Chapter 7. The Green Move Project: an Application Example

Application constraint definition through which the designer limits possible context instances excluding context-element combinations

Database access: using Open Data Base Connectivity (ODBC) [62], the DLV program accesses the GM database to build contextual views of the data

Partial-view definition allows, at design time, to associate each context element with the related portion of data

Context-instance recognition is based on context data captured from the user, sensors and mining historical data

Run time contextual-view generation combines the partial views previously associated with the context elements which compose the current context.

At run time, the system detects sensor values (e.g. the position, from GPS coordinates) and collects other contextual information from the user (e.g. current interest topic); this information gives values to CDT dimensions, and thus generates the corresponding context elements. The logic program that encodes the definition, properties and constraints of the CDT is run against the collected context elements, generating one or more models which are in fact the current context instance(s), in the typical ASP style.

```asp
%dimension(Dimension).
dimension(role). dimension(interest_topic).
dimension(vehicle_type). [...]
%value(Value).
value(greenmove_root).
value(customer).
value(car). [...]
%dim2val(Dimension, Value).
dim2val(role, customer).
dim2val(vehicle_type, scooter).
dim2val(vehicle_type, car). [...]
%val2dim(Value, Dimension).
val2dim(root, role).
val2dim(root, interest_topic).
val2dim(root, vehicle_type). [...]
```

Listing 7.1: Excerpt from the ASP representation of the GM CDT

Once the current context(s) have been generated, the ASP program computes the contextual views [107]. Each contextual view provides a version of the database appropriately tailored according to the current context. At design time, together with the CDT, the designer has defined as many partial views [107] as the context elements from the CDT; each partial view
represents the fragment of the original dataset which has been recognized by the designer as interesting for that context element. The contextual view associated with a generic context is then generated at run time by intersecting the partial views defined for its context elements, and presented to the user for further querying, as required.

As an example, context data together with data about vehicles, users and reservations becomes the input of the DLV program which manages the reservation process (henceforth the reservation system). This reservation system evaluates all information at its disposal and generates the vehicle reservation schedule; at the end, reservations are fixed and should not be modified without user intervention. In order to do this, the reservation system evaluates all the vehicles’ possible states (a vehicle could be available, reserved, out of service, in charge or under maintenance) and tries to satisfy all reservation requests (part of this prototype code is shown in Listing 7.2), although sometimes it might not be possible. The system generates all the possible assignments of vehicles to reservations, ordering such alternatives from the best one to the worst one, according to their ability to satisfy all the constraints (like the last constraint reported in Listing 7.2). At the end of the process, the best alternative is stored in the database, and vehicles are assigned according to this. Note that the final GM system envisages the optimization of vehicle reservations by means of sophisticated operations research algorithms.

% Assign a vehicle to a reservation
assignment(VId,ResrvId) :- reservation(ResrvId, _),
vehicle(VId, _).

% If two reservations overlaps,
two different vehicles must be assigned to them
:- overlaps_resrv(ResrvId1, ResrvId2),
assignment(VId1, ResrvId1),
assignment(VId2, ResrvId2), VId1 != VId2.

% A vehicle can be assigned to a reservation iff it is not locked by a constraint
:- assignment(VId1, ResrvId),
not_avail_assignable(VId2, ResrvId),
VId1 != VId2.

% Only a vehicle must be assigned to a reservation
:- assignment(VId1, ResrvId),
assignment(VId2, ResrvId),
VId1 != VId2.

% Try to satisfy all reservations
:- reservation(ResrvId, User, StartDay,
StartHour, EndDay, EndHour, _, _),
not assignment(VId, ResrvId),
vehicle(VId, _). [1:1]

Listing 7.2: Excerpt from the ASP code for reservations handling
Chapter 7. The Green Move Project: an Application Example

In addition, the reservation system infers from the user’s context data a set of possibly useful services to suggest him/her during reservation confirmation. To perform this operation, that requires a systematic historical data analysis, the system relies on MR-Miner, which allows to extract the history of context and preferences related to each user analyzing all the user previous contexts data (and all the other data stored in the system log), using the instruments and methodologies provided by the PREMINE framework [87]. The outcome of this operation can then be used both from the reservation management system and from the run-time context and preference managers to optimize their outcome.

Further description about this component and of its related theory see [106].

7.2.2 Green Move Dynamic Apps

![Image of GMB dynamic apps](image)

Figure 7.2: Installation of a dynamic application.

In the Green Move platform, dynamic applications are bundles of Java code that, packaged into JAR files, can be sent to GEBs, where they are executed within the Green Move Android application running on the GEB. As explained in some detail in [22], a Green Move dynamic application (GMA for short) must implement suitable interfaces, through which it receives access to the resources of the GEB, such as the data collected from the vehicle. Fig. 7.2 shows the steps through which a GMA is uploaded on a GEB. First (7.2(a)) a T-Rex event is sent to the GEB, which notifies the latter that a GMA is to be installed: the event contains the address on the GMC where the GMA is to be retrieved. Then, the app running on the GEB accesses the GMC (7.2(b)) and retrieves the Java code of the GMA (7.2(c)). Finally, the interfaces implemented by the GMA allow the GEB to instantiate the application and start it (7.2(d)). Similar mechanisms are used to uninstall a GMA from a GEB, or to reload a running GMA. The GMC gives administrators different options to select the vehicles on which a GMA is to be uploaded: single vehicles (unicast), groups of vehicles (multicast) or the whole fleet (broadcast).
7.2. SuNDroPS in Green Move

On the GEB side, the component that is in charge of managing GMAs is called GMcontainer, and is shown in Fig. 7.3. In particular, the GMcontainer listens to the events sent by the GMC; it reacts to them by downloading the GMA code, instantiating it, and removing it when it is no longer needed. In addition, the GMcontainer keeps a persistent registry of the GMAs installed.

![Diagram of Green Move Dynamic App Container](image)

**Figure 7.3:** Schematic view of the Green Move Dynamic App Container.

The interface implemented by a GMA allows it to retrieve from the GMcontainer references to other components of the Green Move environment running on the GEB. In particular, a GMA can use primitives that allow it to: (i) send messages through a variety of communications channels; (ii) acquire the vehicle data (speed, acceleration, state of charge, etc.) from the sensors; (iii) display information on the Graphical User Interface (GUI) of the GEB if one is available.

A GMA can communicate with remote components in several different ways. It can use an HTTP channel that is made available and mediated by the GEB. Such a channel can be used to interact with external services to send/retrieve information (e.g., weather, traffic, advertisements) through standard protocols. For security reasons, the HTTP connection is managed on the GEB by a software proxy which has the possibility to check the traffic and, for example, block access to blacklisted servers. A standard HTTP channel, however, is ill-suited for transmitting (or receiving) streams of data, such as telemetry or composite events. For this reason the GMcontainer offers GMAs access to a T-Rex channel, through which they can publish or subscribe to events. The T-Rex channel allows a GMA to distribute information to other vehicles, if the latter subscribe to the events published by the former. A GMA can also use the Bluetooth channel mediated by the GMcontainer (see Fig. 7.3) to send messages to the smartphone of the user during the rental. This allows a GMA to communicate with the Green Move client application, or another companion application on the user’s smartphone.
Chapter 7. The Green Move Project: an Application Example

The GMcontainer offers also GMAs the possibility to interact with the vehicle through a high-level interface. The interface is very general, and it can be used to read the status of the vehicle, such as the state of charge of the battery or the speed of the vehicle.

If the GEB has a screen (not all do, such as those of scooters) a GMA can use it to provide information to the user.

Using the new, semantic-enabled implementation of TRex SemTRex, described in Sect. 5.3, it is possible to keep a database of applications. Formerly it is up to the developer using the GMC webGUI interface or to the client running on the GEB to set up some parameters of the event (e.g. the url from which to retrieve the JAR of the app).

Instead, using SemTRex, supposing to have a KB containing data about GMA applications, it is possible to retrieve those parameters from the KB and automatically set them during the installation phase by means of the appropriate installation complex event.

The system can retrieve data about application from that KB which contains data about them and their related properties (e.g. the downloadFromURL and mainClass properties) providing there exists a constraint stating that each GMA must have a different name; as an example, the Tesla rule of Listing 7.3 generates the complex event for installing the app “foo” on a specific vehicle: specified by its GEB.

```
define SendApp (String: greenBox_id, String: appUrl, String: class)
  from VehicleStatus[greenBox_id=$a] and ($url, $class) in
  (SELECT ?url ?class FROM appKB.rdf WHERE (?a prop:hasName "foo". ?a prop:
downloadFromURL ?url.
  ?a prop:mainClass $class))
Where SendApp.greenBox_id=VehicleStatus.greenBox_id and SendApp.appUrl = $url and
SendApp.class = $class
```

Listing 7.3: Rule that install the foo app.

7.2.3 GMAadvisor

A service that aims to be tailored to the needs of its users must be able to provide them with customized advice without violating their privacy; otherwise, users might be inclined to give inaccurate information about their preferences, which is of little use for personalization purposes. The GMAadvisor GMA exploits the mechanisms available to dynamic applications to provide users with customized advice through context-aware procedures, without requiring that user preferences be shared with the rest of the Green Move system.

The GMAadvisor GMA is based on the notion of CDT presented in Sect.
3.2.1\footnote{The CDT shown in Fig. 3.2 is the one related to the GM system and will be used in the application examples described in this chapter.} for describing contexts [25], and on an adaptation of the PervAds framework [36], which supports privacy-enforced ads and the distribution of service-related information from sellers to customers. In the rest of this section we briefly describe the features of CDTs and of the PervAds framework, then present the GMadvisor GMA.

The mechanisms described in Sect. 3.2.4 have been adapted to fit the dynamic applications paradigm of Green Move Dynamic Apps (GMA) and to build a GMA, called GMadvisor, for providing Green Move users with customized advice.

The GMadvisor GMA relies on the following features of the dynamic applications framework, as shown in Fig. 7.4:

- access to the GEB GUI, to show suggestions on the GEB display (if this is available);
- communication with the user smartphone through the Bluetooth channel, to exchange messages between the user device (which selects the advice to show, according to the PervAds framework) and the GEB;
- communication with external servers through the HTTP channel, to interact with the central server containing the advice to be distributed;
- retrieval of vehicle data, to determine, for example, the current position of the user.

![Diagram of GMadvisor GMA]

\textit{Figure 7.4: Features offered by the GMcontainer used by the GMadvisor GMA.}

The choice to adapt the PervAds framework into a GMA is driven by two factors. First, not all users may be willing to receive additional information during their trip; this might also depend on the purpose of the trip, as a user might be willing to receive advice -- e.g., of commercial nature -- during a
leisure trip, but not during a business one. Second, by exploiting the HTTP channel provided by the GEB the user avoids consuming her own internet traffic, with its related costs.

The framework providing customized advice to users is based on three components (as depicted in Fig. 7.5):

- A server storing the set of advice that can be sent to users (e.g., position of charging stations, points of interest).
- The GMadvisor GMA installed on the GEB; the application sends to the advice server the user id and position, receives the advices and displays those that match the user preferences as determined by the client app.
- The GMadvisor client app installed on the user device; the app is responsible for storing the private user preferences that should not reside on a remote server; depending on them, it filters the advices received by the GMA; it can also display advices to the user if the GEB does not have a GUI.

The set of possible advice stored on the server can be large. Hence, the system should balance the computational load between the server, the instances of the GMadvisor GMA installed on GEBs and the GMadvisor client apps, and also avoid sending too many suggestions to GEBs, thus exhausting the available bandwidth. To achieve this, the information to be sent to GEBs is pre-filtered at the server level based on the user position, the time and the user’s data that is associated with her registration with the Green Move system (non-sensitive information such as age, gender, etc.). The pre-filtered advice received from the server is further filtered on the client app installed on the user’s smartphone using the sensitive profile information stored there.

Fig. 7.5 shows the steps performed and the interactions among the different components in a usage scenario. First, the user installs the client app on her device and selects her preferences according to a CDT retrieved from the central server. During the trip, the client app exploits the Bluetooth channel to interact with the GMadvisor GMA installed on the GEB. In response to the GMadvisor GMA sending the id and position of the user, the advice server sends back a set of pre-filtered advice, whose metadata is relayed by the GMA to the client app for further filtering. Finally, the selected advice is displayed on the GEB if a GUI is available there (otherwise the advice is shown on the user’s smartphone).
7.2. SuNDroPS in Green Move

Figure 7.5: Schematic description of the GMadvisor GMA and service.

7.2.4 Context-aware Sensors

Another application of the techniques and components that are part of the SuNDroPS framework is the use of context-awareness with sensors, as described in [99].

Available bandwidth in mobile systems is considered a scarcely-relevant issue due to the advent of powerful mobile communication networks. Unfortunately, mobile networks can have quality and/or overloading issues, resulting in decreased net throughput. In a complex car-sharing system with social interactions like GM many information flows have to be managed and the issue needs attention.

Since frequent data transmission is the most energy-consuming operation and can bring to network congestion, operations on the sensed data (e.g. data aggregation) can be performed locally on the sensing nodes, which can send larger packets at lower frequency, instead of small sets of possibly redundant values [35]. However timeliness constraints might be strong and, in this case, the transmission protocol should ensure a good compromise for a proper real-time behavior of the system (e.g. some key data about road events should always be transmitted as soon as available).

To manage the data produced by sensors, we use the PerLa (Pervasive Language) framework [119]: PerLa supports locally managed operations on data in a finely controllable and tunable fashion, hiding the complexity of the possibly high heterogeneity of the underlying devices, which can span from RFID(s) to ad-hoc sensor boards or even portable computers and smartphones. Moreover, as we have seen in Chap. 4, PerLa supports context-awareness for sensors [117] and can be integrated in a general
context-aware system based on the CDT framework.

In our scenario, the data gathering process starts from the moment Mr. Verde unlocks the doors of the assigned vehicle and continues until he gets out of the vehicle releasing it and making it available for the next reservation (the data gathering process restarts for the next user). The whole process is context-mediated by means of PerLa, collecting only data useful for the current user and vehicle context. The data gathered locally from sensors on the vehicle (GPS position, speed, actual power consumption, ...) are pre-processed by the Green e-Box (on which runs a PerLa module) and part of the computation (possibly aggregation) is done by this component. From Green e-Boxes data are pushed to the GM server using the PerLa middleware infrastructure for further processing and storage.

The PerLa context language helps us by allowing different settings in different contexts (e.g. different sampling frequencies).

After declaring the contexts as described in [117], PerLa allows the user to declare the activities that the system must perform at run-time when these contexts become active.

For instance, Mr. Verde, a typical GM user, can drive in two different zones of the city: downtown, where battery charging stations are close to each other, or in the suburbs, where they are located farther away. Whether Mr. Verde is driving downtown or not is detected by his GPS position: if the city center (identified by its GPS coordinates) is farther than a predefined distance \textit{max\_distance} from the user actual GPS position then he is driving in the suburbs, otherwise he is driving downtown. To give him the needed information, we define in Listing 7.4 and 7.5 two different contexts:

**Driving in the suburbs** (Listing 7.4) this context will be enabled only if the \texttt{ACTIVE IF} clause precondition is true; in this case, the system will sample GPS position and battery charge every 60 seconds if the battery charge is \textless{} 50\% (\texttt{SAMPLING EVERY} \ldots{} \texttt{WHERE} clause), only if the vehicle is moving and the sensor provides GPS, speed and battery charge data (\texttt{EXECUTE IF} clause); if the battery charge is \textless{} 35\% an alarm is set (\texttt{SET PARAMETER} \ldots{} \texttt{WHERE} clause) and thus the system will display the nearest charging station.

**Driving downtown** (Listing 7.5) if this context is enabled, the system will sample GPS position and battery charge every 120 seconds if the battery charge is \textless{} 50\%, only if the vehicle is moving and the sensor provides GPS, speed and battery charge data; if the battery charge is \textless{} 35\% an alarm is set and thus the system will display the nearest charging station.
Furthermore, in both contexts the system checks every 5 minutes if a context
switch is necessary (REFRESH EVERY clause), modifying the sampling
frequency as a consequence.

```
CREATE CONTEXT Suburbs_Driving
ACTIVE IF lat > center_lat + max_dist
AND long > center_long + max_dist
ON_ENABLE:
SELECT lat, long, batt_charge
SAMPLING EVERY 60 s WHERE batt_charge
<= 0.5
EXECUTE IF EXIST lat, long, speed,
batt_charge
AND speed > 0
SET PARAMETER ‘alarm’ = TRUE
WHERE batt_charge <= 0.35;
ON_DISABLE:
DROP Suburbs_Driving;
SET PARAMETER ‘alarm’ = FALSE;
REFRESH EVERY 5 m;
```

Listing 7.4: Suburb context

```
CREATE CONTEXT Downtown_Driving
ACTIVE IF lat <= center_lat + max_dist
AND long <= center_long + max_dist
ON_ENABLE:
SELECT lat, long, batt_charge
SAMPLING EVERY 120 s WHERE batt_charge
<= 0.5
EXECUTE IF EXIST lat, long, speed,
batt_charge
AND speed > 0s
SET PARAMETER ‘alarm’ = TRUE
WHERE batt_charge <= 0.35;
ON_DISABLE:
DROP Downtown_Driving;
SET PARAMETER ‘alarm’ = FALSE;
REFRESH EVERY 5 m;
```

Listing 7.5: Downtown context

Do note how the computation can be distributed among the system
components, as described in Chapter 4, Sect. 4.1: all the processing involving
battery charge is executed locally to the Green E-box, sending data to the
GM server if and only if all the required conditions are satisfied (speed > 0
AND batt_charge <= 0.35) preserving battery charge that might be wasted
in frequent, unnecessary transmissions.

More complex operations can also be implemented using the PerLa COP
extension described in Chapter 5, Sect. 5.2.3: for instance it will be possible
to perform complex interaction operation (like activating control sequences
on the vehicle, supposing it exposes some kind of standard Java API) de-
pending from the context in which the user (or the vehicle) is. For instance,
if there exists several context of use of the vehicle depending from the user
driving style (standard, sport, eco, ...), PerLaCOP can trigger modification
(through the Java vehicle APIs) of the vehicle configuration to keep
the driving safe and also trying to reduce the battery consumption in every
case.
The design and implementation of the SuNDroPS system presented in Chapter 3 and of its components presented in Chapters 4, 5, 6 aims to give a full-fledged solution to the big data management and analysis problem presented in Chapter 1.

Although there is still a lot of work to be done, the proposed and partially implemented system represents a first step in supporting users that must deal with big data in different contexts.

Moving from the original Context-ADDICT implementation, that already allow to achieve a complete integration of data coming from different and heterogeneous data sources, SuNDroPS implements some new and interesting features that allows to manage also data streams and semi-automatic context detection.

This detection process is based both on the analysis of the current environment through gathered sensor data (collected by PerLa and TRex modules) and from the historical data analysis (performed by MR-Miner module), operations performed by the three newly introduced modules. Several issues have been faced during the modules integration process, as it has been described in previous chapters.
Chapter 8. Conclusions and Future Work

By the way, considering what the new system can do w.r.t. what ContextADDICT does, this integration must become tighter, requiring increasing cooperation among them.

In the previous pages some evaluations of the proposed implementation have been shown and commented, pointing out the strengths and weaknesses of the new SuNDroPS components. Part of those issues, the ones that have been discovered and analyzed in Chapter 4, Sect. 4.3 have partially already been dealt with in the proposed PerLa and TReX extension in Chapter 5, Sections 5.1 and 5.3 respectively while many other ones have been only partially analyzed and fixed and they will be discussed deeply in Sect. 8.1.

However, as the different examples proposed in this work suggest, there exist many application fields of the techniques and of the instruments described in the thesis. Those application scenarios comprehend very different fields like automotive, home & office automation, eCare & eHealth, emergency situation & disaster recovery, …

The common element that connects all those different scenarios is the need of treating and analyzing large flows of data in almost real-time, partially storing them (e.g. storing some aggregate result computed on the basis of the gathered data) for further analysis, integrating them with data coming from heterogeneous datasources in order to give the most accurate and complete answer to the users’ request in a given context.

Considering those needs, systems like SuNDroPS will represent part of the core of the frameworks that will be used in order to automate the interaction among users and very complex systems, like the ones mentioned above, providing an optimized user experience and reducing all the useless waste of time due to the processing of lots of unnecessary, out-of-context data.

8.1 Future work

Several different open issues still exist, related to the three main research areas object of this thesis. Those issues with information flow processing, context-awareness and data mining must be considered and (possibly) solved in order to achieve the final and complete implementation of the full-fledged SuNDroPS system and to further improve our approach.

8.1.1 Information Flow Processing

As it has been already said before, it is not possible to tell which is the best IFP paradigm among the two paradigms presented (DSMS, represented by
8.1. Future work

PerLa, and CEP, represented by TRex), since they have functions and peculiarities that best fit them in different scenarios. Both the two proposed extensions need to be finalized in order to prove the theories exposed in this thesis.

From the implementation point of view, PerLa requires the biggest effort: the system is being completely reengineered, not allowing to introduce many of the new proposed features. This reengineering step is the base step for all the other features, and is still ongoing. Once it has been finished, all the proposed extensions can be easily integrated in the whole system (since for many of them at least a partial implementation is already available).

SemTRex needs the caching algorithms described in Chapter 5, Sect. 5.3 to be implemented (only the first two have been currently implemented and integrated in the system). An effort is also required since the RDF-3X SPARQL engine, and its related library, do not currently support the offset expression in the LIMIT clause, not allowing to use the partial resultset caching feature described in Chapter 5, Sect. 5.3\(^1\).

SemTRex requires also efforts in defining even more complex caching algorithms trying to speed up and stabilize the processing phase when it involves the semantic enrichment of the events.

Also an integration between SemTRex and GTRex (the GPU-based TRex engine implementation described in [110]) has to be considered to obtain overall better performances.

8.1.2 Context Awareness

The theoretical framework that underlies all the context-aware operations described in this thesis is almost complete in all its features.

However the methodologies presented in this thesis require more of real-world field test results to prove that they allow to acquire actually (at least) some of the benefits depicted on the works presented in Sect. 2.1.

Once the SuNDroPS system has reached its complete implementation, it can be used on-field to acquire some experimental results and to perform all the necessary evaluation.

To do so, the structure and APIs presented in the architecture section of this thesis (Chapter 3, Sect. 3.1) must be fully implemented, allowing all the existent modules to cooperate, allowing SuNDroPS to run in its full-fledged version.

\(^1\)Please notice that the benchmarkings proposed in the same section have been performed using a subset of the KB small enough to fit in the main memory of the test systems.

169
8.1.3 Data Mining

Since the tests performed in Sect. 6.4 show that the algorithm performs well with the benchmarking datasets, and our evaluations allow to state that a slightly modified version of MR-Miner algorithm should scale well, MREClarT and MR-Miner require more extensive test phase, with bigger datasets in order to achieve more information about the real applicability of the algorithm to real-world scenarios.

The implementation of the algorithm should be also “cleaned” and revised to obtain the best performance, since it will be quite a critical – and time consuming – task that it has to perform.

More evaluation on the proposed extensions should also be carried out, to understand if the underlying assumptions are correct.

8.2 Final Thoughts

After these three years of work and study as a Ph. D. candidate and as a research and teaching assistant I personally feel more confident of my own resources. These three years of study and research allowed me to understand, and sometimes to devise, new methods, techniques and tools in Computer Engineering, as well as to improve my personal know-how and reinforce my abilities in the perspective of using them during my future professional life.

At the same time, I had the opportunity to challenge the acquired competence with several different research projects, understanding how my knowledge can be applied to real-world problems and scenarios: a step that may not be as simple as imagined while you are just studying a method, and applying it to some sample, artificial scenario. Also, I have gained experience in relating with other professionals, acting in team and cooperating in order to reduce the efforts to reach our final common goals.

Furthermore, during the three years, I have also had the opportunity to act as a teaching assistant in many different academic courses (from basic programming ones to more specific ones, mainly related to information systems and pervasive systems). This experience helped me to improve my relational abilities, which are very important for people whose aim is to work in team, understanding how to relate to different kinds of people (e.g., you cannot interact with a student in the same way as you interact with an engineer). During a teaching experience you also understand how important is to share your knowledge as much as possible, in order to reduce the information gap between academia, professional and ordinary people.
8.2. Final Thoughts

All this things have been possible thanks to the excellent people, professors and researchers I have met during my activities.

On the whole, considering my last three years’ experience and where I started from, I think I grew up a lot both as a professional and as a man. I would really recommend to perform such a kind of an experience to my current and future students (and in fact I have done it already: one of my master students recently became a Ph. D. student!), since I believe that the experience made in this journey can become a really important added value for the professional life of engineers who want to obtain important roles in their future working experience.
## List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Schema of Perla’s architecture</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>System Module Architecture</td>
<td>37</td>
</tr>
<tr>
<td>3.2</td>
<td>Green Move project CDT</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Interaction between the original Context-ADDICT system modules</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Schema of Perla’s architecture</td>
<td>57</td>
</tr>
<tr>
<td>4.2</td>
<td>Context Dimension Tree (CDT)</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>Partial (ENABLE) components definition</td>
<td>63</td>
</tr>
<tr>
<td>4.4</td>
<td>Contextual block composition $\oplus_D$</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Schema of TReX’s architecture</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Events’ schema</td>
<td>74</td>
</tr>
<tr>
<td>5.1</td>
<td>Perla reengineered components</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>Dynamic FPC creation process</td>
<td>92</td>
</tr>
<tr>
<td>5.3</td>
<td>DatabaseWrapper architecture</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>FPCs Channels</td>
<td>99</td>
</tr>
<tr>
<td>5.5</td>
<td>Request elaboration</td>
<td>100</td>
</tr>
<tr>
<td>5.6</td>
<td>Standard TinyOS message structure</td>
<td>101</td>
</tr>
<tr>
<td>5.7</td>
<td>PerLa message structure</td>
<td>102</td>
</tr>
<tr>
<td>5.8</td>
<td>Context Dimension Tree (CDT) – Addition</td>
<td>109</td>
</tr>
<tr>
<td>5.9</td>
<td>The system architecture</td>
<td>114</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of a powerset lattice ( [138])</td>
<td>138</td>
</tr>
<tr>
<td>Chapter</td>
<td>Figure/Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2</td>
<td>MREClAT algorithm outline</td>
<td>143</td>
</tr>
<tr>
<td>6.3</td>
<td>Benchmarks results</td>
<td>150</td>
</tr>
<tr>
<td>6.4</td>
<td>Prefix extension benchmarks</td>
<td>151</td>
</tr>
<tr>
<td>7.1</td>
<td>Schematic view of the overall architecture of the Green Move system</td>
<td>154</td>
</tr>
<tr>
<td>7.2</td>
<td>Installation of a dynamic application</td>
<td>158</td>
</tr>
<tr>
<td>7.3</td>
<td>Schematic view of the Green Move Dynamic App Container</td>
<td>159</td>
</tr>
<tr>
<td>7.4</td>
<td>Features offered by the GMcontainer used by the GMadvisor GMA</td>
<td>161</td>
</tr>
<tr>
<td>7.5</td>
<td>Schematic description of the GMadvisor GMA and service</td>
<td>163</td>
</tr>
<tr>
<td>B.1</td>
<td>Class diagrams of HTTP PerLa components</td>
<td>198</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Abbreviation for allowed field types of logical objects</td>
<td>73</td>
</tr>
<tr>
<td>4.2 PerLa’s data table for example 1</td>
<td>74</td>
</tr>
<tr>
<td>4.3 Tesla’s data table for example 1</td>
<td>75</td>
</tr>
<tr>
<td>4.4 PerLa’s data table for example 2</td>
<td>76</td>
</tr>
<tr>
<td>4.5 Tesla’s data table for example 2</td>
<td>78</td>
</tr>
<tr>
<td>4.6 PerLa’s data table for example 4</td>
<td>81</td>
</tr>
<tr>
<td>4.7 PerLa’s data table for example 5</td>
<td>82</td>
</tr>
<tr>
<td>4.8 Tesla’s data table for example 5</td>
<td>82</td>
</tr>
<tr>
<td>4.9 PerLa’s data table for example 6</td>
<td>83</td>
</tr>
<tr>
<td>4.10 Tesla’s data table for example 6</td>
<td>84</td>
</tr>
<tr>
<td>6.1 MR-EClat and BigFIM comparison</td>
<td>148</td>
</tr>
<tr>
<td>6.2 MR-EClat and BigFIM execution steps comparison</td>
<td>148</td>
</tr>
<tr>
<td>6.3 Datasets characterization</td>
<td>149</td>
</tr>
<tr>
<td>6.4 MREClat performances (times reported in seconds)</td>
<td>149</td>
</tr>
<tr>
<td>6.5 Synthetic datasets characterization</td>
<td>149</td>
</tr>
<tr>
<td>B.1 HttpRequest Attributes consistency</td>
<td>199</td>
</tr>
</tbody>
</table>
List of Algorithms

1  Composition Algorithm Pseudocode ........................ 65
Listings

4.1 The Context Dimension Tree .......................... 61
4.2 Examples of contexts in PerLa ........................ 62
4.3 Modified CL. syntax to support partial components ... 64
4.4 The CDP Algorithm ........................................ 72
5.1 Device description header ................................ 93
5.2 Attribute declaration ..................................... 93
5.3 Channel declaration and configuration ................. 94
5.4 I0request configuration ..................................... 94
5.5 Message declaration – 1 .................................. 95
5.6 Message declaration – 2 .................................. 95
5.7 The SmokeMonitoring example in PerLa ................ 109
5.8 The SmokeMonitoring example in COP ................ 110
5.9 The OverheatMonitoring example in PerLa ............ 111
5.10 The VentilationMonitoring example in PerLa .......... 112
5.11 The integrated COP-PerLa approach .................... 115
5.12 Example of rules with KB operators .................... 117
5.13 The KBPredicate structure .............................. 119
5.14 The KBPredicate structure .............................. 119
5.15 The KBPredicate structure .............................. 120
5.16 The KBPredicate structure .............................. 120
5.17 The QueryItem class .................................... 122
5.18 The cache structure ...................................... 124
5.19 Two queries examples .................................... 127
7.1 Excerpt from the ASP representation of the GM CDT ... 156
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Excerpt from the ASP code for reservations handling</td>
<td>157</td>
</tr>
<tr>
<td>7.3</td>
<td>Rule that install the foo app.</td>
<td>160</td>
</tr>
<tr>
<td>7.4</td>
<td>Suburb context</td>
<td>165</td>
</tr>
<tr>
<td>B.1</td>
<td>HTTP Channel XML descriptor – 1</td>
<td>199</td>
</tr>
<tr>
<td>B.2</td>
<td>HTTP Channel XML descriptor – 2</td>
<td>200</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


APPENDIX

Relevant Publications

A.1 Publications on Journals and Magazines

   “Green move: A platform for highly configurable, heterogeneous electric vehicle sharing”
   Intelligent Transportation Systems Magazine, IEEE, 6(3):96–108, Fall 2014

A.2 Publications on Congress Proceedings

1. E. Panigati
   “Personalized management of semantic, dynamic data in pervasive systems: Context-addict revisited”

2. F. A. Schreiber, E. Panigati
Appendix A. Relevant Publications

“Context-aware software approaches: a comparison and an integration proposal”


“Vehicle-sharing: Technological infrastructure, vehicles, and userside devices - technological review”


4. E. PANIGATI, A. Rauseo, F. A Schreiber, L. Tanca

“Pervasive data management in the green move system: a progress report*”


5. E. PANIGATI, A. Rauseo, F. A Schreiber, L. Tanca

“Aspects of pervasive information management: an account of the Green Move system”

In Proc. of the 10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, Paphos, Cyprus, Dec 2012


“Green Move: towards next generation sustainable smartphone-based vehicle sharing”

In Proc. of SustainIT2012, Oct 2012

7. E. PANIGATI, A. Rauseo, F. A Schreiber, L. Tanca

“Context-aware information management in the Green Move system – extended abstract”

A.3 Submitted Book Chapters

1. E. Panigati, F. A. Schreiber, C. Zaniolo
   “Data Streams and Data Stream Management Systems”
   Submitted for publication in “Data Management in Pervasive Systems
   - The Shapes and Dynamics of Information in a Pervasive World”
   Book, Springer

A.4 Technical Reports

1. F. A. Schreiber, E. Panigati
   “Context aware data management and context oriented programming:
   is convergence possible?”
   Technical Report 2014.7 (DEIB)

2. E. Panigati
   “Methods for Supporting Critical Systems’ Failure Diagnosis in the
   Railway Scenario”
   Technical Report 2013.12 (DEIB)
APPENDIX

PerLa Reengineering: Implementation Details

This appendix presents some details about the implementation of the reengineered PerLa version presented in Chapter 5, Sect. 5.1.

B.1 Middleware Reengineering

The two components involved in a deep reengineering process are the FPC and the Channel components. We will briefly describe which are the main APIs exposed by them. Considering the channel massive reimplementation, we will also detail it in the following, considering the example of the HttpChannel.

B.1.1 Functionality Proxy Component (FPC)

The new implementation of the FPC interface is entirely defined in terms of operations on attributes by the following APIs:

Task get(Attributes, TaskHandler) Retrieves the value of the specified attributes
Appendix B. PerLa Reengineering: Implementation Details

Task `get(Attributes, Period, TaskHandler)` Periodically retrieves the value of the specified attributes.

Task `set(Attributes, Values, TaskHandler)` Sets the value of the specified attributes.

These new APIs allow the previously impossible asynchronous management of the interaction among the various system components and become the ground on which it is possible to build and implement all the new middleware components.

B.1.2 Channel

The Channel, that can now be considered as the central communication handling module, exposes the following APIs, to allow the sensors-to-system (and vice versa) communication:

IOTask submit(IORedemand, IOMethod) Submits an `IORedemand` for execution, and specifies the callback handler to be invoked when the result is ready.

setAsyncIOHandler(IOHandler) Sets a callback handler for managing asynchronous communications initiated by the remote device.

As can be seen by examining the methods in the `IOHandler` interface, also the Channel – like the FPC – can now handle asynchronous operations by means of the following methods:

complete(IORedemand, Payload) Asynchronously invoked when the I/O request completes successfully.

error(IORedemand, error) Asynchronously invoked when the I/O request completes with an error.

B.1.2.1 HTTP Channel implementation

To understand how the new communication paradigm is managed by the channel, we propose in the following a detailed look at the `HttpChannel` implementation that has been realized starting from the classes described in Chapter 5, Sect. 5.1.5.

The implementation is based on the following seven software objects:

HttpChannel is the object responsible of performing HTTP call, using `HttpChannelRequest` and returning a `ChannelResponse`. 

196
B.1. Middleware Reengineering

(HttpChannelRequestBuilder) generates the (HttpChannelRequest)

(HttpChannelRequest) wraps the HTTP request with completed URL (host, path and query), HTTP method and eventually entity and content-type

(HttpChannelFactory) validates the channel descriptor and creates the object (HttpChannel)

(HttpChannelRequestBuilderFactory) parses the XML file, wrapped in its own HttpRequestDescriptor, and creates an instance of HttpChannelRequestBuilder for the related FPC

(HttpChannelDescriptor) is a channel instance responsible of the HttpChannelRequests. It is a java object that translates the channel tag having like attributes just an id

(HttpRequestDescriptor) is the java class mapping HTTP request XML tag inserted in the device descriptor.

Each object and the main methods that it provides will now be analyzed in detail to give a closer look of its implementation.

(HttpChannel) Is created by HttpChannelRequestBuilderFactory during the PerLa start-up and associated to the correspondant FPC. It is then invoked by the FPC using the method submit that, after having consumed a HttpChannelRequest, returns a ChannelOperation containing the logic with call result. This method is inherited by the AbstractChannel object, that implements the Channel interface.

The submit operation is performed by means of the handleRequest method that dispatches the GET, POST, PUT and DELETE HttpChannelRequest, respectively to handleGetRequest, handlePostRequest, handlePutRequest and handleDeleteRequest. All these methods are implemented following the REST architecture.

For the simple and standard implementation proposed in this work the Apache HTTP Component library\(^1\) has been used.

The object CloseableHttpClient, created into HttpChannel constructor, works as transporter of HttpUriRequest, built by PerLa Channel in handle methods.

The class diagram of the HttpChannel related classes is shown in Fig. B.1(a).

\(^1\)https://hc.apache.org/
Appendix B. PerLa Reengineering: Implementation Details

![Class diagrams of HTTP PerLa components.](image)

**HttpChannelRequestBuilder**  The `HttpChannelRequestBuilder` object is used by a FPC to create a `HttpChannelRequest` before invoking a HTTP call. Once built, it gets all needed parameters to correctly configure each new `HttpChannelRequest`. Created by `HttpChannelRequestBuilderFactory`, it instances an `HttpChannelRequest` through the `create` method.

The class diagram is reported in Figure B.1(b).

**HttpChannelRequest**  `HttpChannelRequest` models the content to be sent through the channel, so has essentially four parameters: (i) HTTP method, (ii) URI, (iii) Content-type and (iv) Entity

The Entity parameter is dynamically set by the FPC using the `setPayload` method.

`HttpChannelRequest` implements methods of `ChannelRequest` interface used by FPC on a higher abstraction level (Fig. B.1(c)).

**HttpChannelFactory**  This class is responsible of the `HttpChannel` creation starting from the `HttpChannelDescriptor`. It validates the descriptor and initializes the channel applying the attributes (e.g. id, channel identifier).

**HttpChannelRequestBuilderFactory**  It is one of the critical point of HTTP integration because it must validates the XML request tag an checks its consistency with REST architecture. In Table B.1 are shown the attributes that are: allowed (A), not allowed (NA) and mandatory (M) for each HTTP Method.
B.1. Middleware Reengineering

<table>
<thead>
<tr>
<th>method</th>
<th>response</th>
<th>content-type</th>
<th>entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>M</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>POST</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>PUT</td>
<td>NA</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>DELETE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table B.1: HttpRequest Attributes consistency

Attributes id, channel-id and host are always mandatory while attributes path and query are always allowed.

All controls are performed into the create method that throws an InvalidDeviceDescriptorException when a check fails.

**HttpChannelDescriptor**  
HttpChannelDescriptor is the POJO (Plain Old Java Object, an ordinary Java object) translating the XML channel tag. The channel identifier is used for associating each request with its channel. An example of the channel tag is provided below:

```xml
<channels>
  <ht:channel id="http_ch_01">
  </ht:channel>
</channels>
```

Listing B.1: HTTP Channel XML descriptor – I

**HttpRequestDescriptor**  
Request descriptor is a POJO that maps all the requests related tags. For correctly understanding the mapping it is necessary to know how the XML request tag is composed. All request tags are characterized by nine attributes:

- **id** is a required string identifier of request.
- **channel-id** is a required string identifier of channel sending this request.
- **host** is the required string (URL) identifying the host to which the HTTP request has to be sent.
- **path** is the optional identifier of tag message that represent a dynamic path.
- **query** is the optional identifier of tag message that represent a dynamic query.
Appendix B. PerLa Reengineering: Implementation Details

**entity** is the identifier of tag *message* that represent the entity (content) of HTTP request and it is required for POST and PUT requests

**method** is an optional enumeration value that specifies HTTP method for the request (get, post, put or delete); the default value is *get*

**content-type** is the optional content-type specified in HTTP request. The default value is */* (Known as wildcard content-type)

**response** is the identifier of the tag *message* that represent the response content of HTTP request. It is required for post and get request

The following script represent a POST request example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<device ...
 xmlns:ht="http://perla.dw1.org/channel/http">
  ...
  <requests>
    <ht:request id="post_req"
      channel-id="http_ch_01" 
      method="post"
      host="http://mysite.com"
      path="req_path_id"
      query="req_query_id"
      content-type="application/x-www-form-urlencoded"
      entity="req_entity_id"/>
  </requests>
</device>
```

**Listing B.2: HTTP Channel XML descriptor – 2**

Looking at the class diagram in Figure B.1(d) it is possible to understand the relationship between methods and attribute of the *request tag.*

*HttpRequestDescriptor* implements the *getMessageIdList*, an abstract method inherited by *RequestDescriptor*, usable by FPC to parse the request.

### B.2 PerLa on Motes

The PerLa nesC implementation described in Sect. 5.1.6 is split in four main files:

**PerLaSensorAppC.nc** This file contains the top-level configuration; it wires together the component that are used in the application

**PerLaSensorC.nc** This file contains the implementation of commands/ events provided/used by the application. It is the biggest part of the code, that specifies the behaviour that motes will assume in different situations (at boot, when a timer fires, when they receive a message and so on)
B.2. PerLa on Motes

PerLaMessage.h This file specifies the structure that a message must assume in the PerLa system

PerLaSetting.h This file contains some structures, used in the application, that producer can modify as it prefers. This will allow him not to touch the code of the application so deeply by accessing only this file.

B.2.1 Design Details

In order to guarantee an easier migration from TinyOS to some other embedded operating systems or a more comfortable maintenance of the code, some functions and a task that separate the logic from the applicative code have been implemented:

splitMessage() it is called when a message is too long to be sent in a single packet. It takes the big message, split it into chunks that, with an header replicated for every partial message, can fill the maxPayloadSize of a TinyOS packet. Moreover it fills the field of the header in which it is specified with the total number of packets in the sequence and the number of each packet in the sequence itself. When a perlaMessage is composed, this function puts it into a queue of transmission and starts the task used for sending packets

sendMessageInQueue() this is the task responsible to send packet in queue. It checks if the queue is not empty and sends the first packet. Once it sent the packet, it calls itself recursively until the queue is not empty

setHeader(uint16_t type) it takes as an input the type of the message (specified by the protocol) and fills the id, timestamp (both automatically got) and type fields of the header of a perlaMessage.

composeAndSend() it takes the header and the payload with the field already compiled, puts them into a perlaMessage and sends it. If the size of the perlaMessage is too big with respect to the size of a TinyOS message, it calls the splitMessage() function

report_error(), report_sent(), report_received() they make action with respect to the specified event. At the state of art, they simply toggle leds

Considering timers and execution of multiple threads, also sensors, even if they are unique on the board, have been abstracted with more than one logical representation at software level. This is necessary because more
Appendix B. PerLa Reengineering: Implementation Details

than one action could run at the same time on the same sensor and on the same mote. So we have the PeriodicTemperatureSensor, the EventTemperatureSensor and the OneShotTemperatureSensor. The same thing has been done for Humidity Sensor.

Currently, the packet splitting operations is performed on motes, while the recomposition is not performed on the sink, but it is performed directly on the channel, at a higher level, since the complexity of this type of operation can be better managed from the Java channel, making more uniform the behaviour of the sink that only receives a packet and sends it to the channel (independently from the type of packet received).