UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Master in Innovation and Research in Informatics (MIRI)

Semantic IoT for Reasoning and Big Data Analytics

Candidate: Reinout Van Hille
Supervisor : Prof. Fatos Xhafa

Faculty of Informatics of Barcelona, UPC, Spain

June 28, 2018

Abstract

In recent years, the Internet of Things has been evolving and expanding as a disruptive technology. What started out with the idea of tagging things using RFID tags and connecting them to the Internet has now become the common name for connecting anything and everything. With the IoT being a main source of data generation due to very large number of devices and their rather low cost, data processing in both online and offline modes, has become a challenge due to size, data rate, variety, etc. In this context, the semantic web technologies are sought to enrich the already Big Data with even more data but facilitating processing and reasoning over the data. The benefits of bringing the semantic web to the IoT are twofold. By adding a common model to the data, the interoperability, i.e. the domain where it can be used, is increased significantly. Furthermore, such enriched data streams will allow machines to infer even more information and knowledge from it that would otherwise not have been available.

However, such benefits are not obtained without paying a price. Enriching the data requires additional processing steps and causes an explosion of the already Big Data. Traditionally, dealing with Big Data is a problem that has been tackled using Cloud computing. Voices are rising however, that soon the idea of pushing all data to the Cloud will no longer be sustainable. This is especially fortified by the upcoming 5G technologies, which brings the capacity of generating even bigger data streams at devices. A possible solution is to alleviate the Cloud by performing some computations at lower levels. **The question can be raised as to what exactly should be processed and where, from IoT layer to Cloud layer.** The tasks performed in the Cloud are often of such high complexity that they cannot be performed by any lower level infrastructure. It is thus necessary to identify tasks that can be moved away from the Cloud and pushed towards the edges of the Internet.

In this thesis the usage of the semantic web technologies in an event processing, IoT streaming context is studied. Event processing is particularly interesting, because it implies that only the data contained within the events is data of interest. Thus, by attempting to filter the data to find these events at an early level, a large amount of data can be discarded. This would help in balancing the amount of data that is generated additionally by the enrichment process. The use-case of this study is the real-time detection of potholes in the road. This implies that the study is oriented towards the IoT in non resource constrained environments. Such environments can also be found in other places, such as manufacturing and smart cities.

The aim of the study was to answer the previously raised question of the tasks that can be distributed towards the edge of the Internet, within the context, and to propose an architecture model that uses these ideas efficiently. The architecture model consists of multiple processing layers, namely three layers, each manipulating the data in some way before passing it on to the layer above it, until it finally reaches the client application. To assess the feasibility of this model, a proof of concept application was designed implementing this architecture. Furthermore, an API was developed in order to manage the layers. This would allow the adaptation of the implemented system for other projects. Various experiments are ran in order to assess every layer's performance and judge if the selected tasks could realistically be performed by each layer and by the system as a whole.

The results of this study include promising results about the proposed architecture model for decoupling IoT from the Cloud by introducing intermediate layers of processing at the edge of Internet, its feasibility and real-time performance, despite of the limitations of the testing environment. The study also reveals the benefits of using semantic data enrichment of IoT data stream for reasoning purposes. As a future work, we point out the need to deploy the envisioned architecture in a real computing infrastructure comprising all layers and thus to obtain more accurate information regarding its real-time performance and layering benefits.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In recent years, the Internet of Things (IoT) has become one of the most prominent concepts in the IT environment. Due to the broad range of applications that are associated with IoT, a vast amount of different definitions have been provided for it. Furthermore, the IoT has been observed to be evolving throughout the years, from the tagged objects model to a full interconnected web of things involving social objects.[1] A "definition for the Internet of things for 2020" is given:

> *"A conceptual framework that leverages on the availability of heterogeneous devices and interconnection solutions, as well as augmented physical objects providing a shared information base on global scale, to support the design of applications involving at the same virtual level both people and representations of objects."[1]*

This indicates just how many concepts are associated with the IoT. A major part of this being the aspect of interconnected devices and objects. The rapid developments made in the low power industry have created the opportunity to gather data in various environments using even the most tiny of devices. Similarly, due to innovations in the processing industry and as predicted by Moore's law, the processing capabilities of micro-controllers has also increased dramatically. The combination of these two factors has resulted in an explosive growth in the industry of connected devices, with a stunning estimated 20.35 billion for the past year of 2017[2].

These devices form the basis of the IoT, but as stated in the definition, the IoT is not just about the devices, but also about enabling these devices to be used for various applications. An associated paradigm is that of *Big Data*, which is often described using the five "V's" of volume, variety, velocity, value and veracity. In particular when dealing with the velocity component, a change in processing paradigm from a batch to stream computing model creates both new possibilities and challenges.

IoT event processing A possibility of interest in this thesis, which also presents a challenge, is found in the event processing environments. In a static context, events are limited to a particular data element of interest, or a combination of multiple elements. In a dynamic context, i.e. when dealing with data-streams, patterns in time and sequences of events can also be considered. The detection of these patterns is a field emerging together with the data-streams. In this field the machine learning techniques and neural networks can be considered the highest level of processing, as they deal with the analysis of patterns that are generally too hard to define otherwise. However, for patterns that are known, the approach of complex event processing can be used. In this approach, complex events are considered as a temporal relation of possibly multiple atomic events. When all atomic events are detected following the pattern description, a complex event has been found.

IoT and Cloud computing In recent years, the IoT has also increasingly become associated with cloud computing environments[1]. Often the reasoning behind it is found in the fact that the operating circumstances of the sensors is tied to resource constraints. Hence these devices do not support doing computational work as their energy usage must be minimized. The distributed architecture that these IoT systems bring with them however, often does allow for edge computing. While a single local sensing node may not be capable of doing any computations, the broker node that accumulates the data from various sensing nodes can usually do this, because it is not tied to such resource constraints. Furthermore not all IoT environments are tied to these resource constrains. Making use of these nodes, which are found at the edge of the network in an IoT environment, is exactly what edge computing is about. By performing computations at the edge of the network, various advantages become apparent. First off, the available computational power is used more efficiently, since more nodes participate in the computation process. Furthermore, by doing some computations in the edge the network overhead can be lowered. Lastly, certain information regarding a data element is more readily available closer to the source.

IoT and semantic technologies In regards to this knowledge, the semantic technologies were created to maximize the amount of information available from data. These technologies originated as an approach focused on data on the web quite some time ago. The observation was made that, while having massive amounts of data available, a clear model for describing data and its relationships was not. Furthermore, even when a clear description of the data was available, the information stored inside of is often incomplete. This is because of the existence of implicit knowledge. This is the knowledge that is realized through reasoning over relationships and descriptions that is often forgotten, considered insignificant or obvious, or its presence is not even noticed. It is precisely this, the ability for machines to be able to reason with data in a way that allows them to find this implicit knowledge given explicit descriptions, that is the target of the semantic technologies.

These semantic technologies have recently started to find their way into the IoT. Amongst the developments there exists a tendency towards processing these enriched data streams in the complex event context.

The combination of various paradigms such as IoT, Edge Computing, Semantic Processing and the Cloud is making possible to leverage in almost real time and reliably Internet of Medical Things for patient monitoring, and eHealth systems, more broadly [3].

Use case In this thesis a car's sensory data is processed in order to detect potholes[1] in roads (see, e.g., Fig. 1.1).

The detection of potholes in roads may be used to various ends. First of all, by categorizing the potholes, the scheduling of the repairs may be optimized. Evidently, the most dangerous potholes should be fixed first. During winter periods repairs of smaller potholes may be given higher priority from an economical point of view, as this may prevent frost damage to roads that would otherwise still have been in good quality.

Secondly, the safety of road users may be improved by providing them with warnings. Potholes are especially dangerous for motorcyclists and may even cause accidents. Similarly, a car hitting a pothole without adapted speed may also suffer damage. If the location of these holes are known, warnings

[1]A depression or hollow in a road surface caused by wear or subsidence.

Figure 1.1: A pothole in a road.

can be issued until the repairs are made, allowing traffic to regulate their speed to the circumstances.

Lastly, an analysis of the historical data of these holes may provide more insight regarding the expected lifetime of roads. The lifetime of a road is estimated at its construction, but is greatly affected by its actual usage and degradation rate. A historical analysis of potholes can improve the estimation on both current and future degradation rate, allowing a better scheduling of road traffic.

In this thesis an architecture for processing the car data in a semantic, complex event context is proposed. The focus is set towards the adaptability of the system to reuse it in other contexts, and towards the use of edge computing to improve the system efficiency. The objective is to use these concepts to detect potholes. The timely detection of potholes allows governments to tackle the issue early, thus preventing further damage both to the roads and its users.

Thesis structure and reading

This thesis is structured as follows. In this introduction, the context of the project, the use-case and the most important concepts have briefly been touched. Next, in chapter 2, the problems are identified more specifically and elaborated. The identification of the problems leads to the formulation of the key research question in chapter 3. Using this research question, the project scope and its objectives are determined. Before introducing the proposed solution, it is necessary to elaborate the background knowledge and related work in the field of the study, i.e. defining the state of the art. This is done in chapter 4. For each section of the related work, an evaluation is made. This evaluation helps to understand the architectural design choices. The requirements set by the use-case, lessons learned from the related work and the problems observed form the baseline for the proposed architecture model in chapter 5. This chapter also includes the details on the implementation of the model that serves as a proof of concept to evaluate if it meets the set requirements and objectives. Various experiments and tests are run in order to attempt to understand the strengths and weaknesses as much as possible in chapter 6. Finally, an evaluation of the implemented model is made in chapter 7. The results are analyzed in retrospect of the project objectives. Considerations for future work are also formulated.

Chapter 2

Problem statement: Offloading processing and reasoning to edges of the Internet

As mentioned earlier in the introduction, the cloud computing paradigm is currently highly favored in the IoT scene. Some of the important reasons for this are its more centralized architecture model and the benefits gained from economies of scale. Centralized models offer various advantages: the implementation is less complicated as no decisions have to be made on the delegation of tasks between various components, issues with regard to synchronization of the system become simpler, there are no additional complications with regard to interfacing and communications, a thin client model can be used resulting in easy deployment, etc. Furthermore its key disadvantages of the single point of failure and bottleneck problem are tackled by the cloud infrastructure. Failure can readily be avoided by replicating machines - possibly even at runtime and in different locations to assure data availability. A bottleneck problem can simply be tackled by increasing the computational power in the cloud, which in turn is not that expensive due to the economies of scale. The advantages of a centralized cloud computing approach are numerous and justify the observed movement of the IoT towards the cloud.

Yet there is some criticism to be found to this approach. By selecting a thin client model and moving all computations towards the cloud, not all available hardware is maximally exploited. In an event processing context the possibilities lost are twofold. Not only is the available hardware not being used optimally, but the first step in processing the data when attempting to detect events evidently involves the filtering of the data, discarding irrelevant content. Depending on the context, this may even be the vast majority of the data. Regardless, the overhead on the data transport towards the cloud can be lowered by performing these kinds of operations in the edge. This is crucial as network transport costs for a dedicated link to the cloud are not negligible. In an environment with high data rates this is even more prevalent. With the aspect of 5G on the horizon this problem will become amplified even more, and it is expected that cloud computing infrastructures will begin to struggle to deal with the further increased network requirements [4]. Issues regarding the network overhead and latency are amongst the concerns.

Dealing with the lifetime of events is also a problem for a fully cloud centric model. As data moves further away from the source, determining when exactly it originated and how long it is relevant becomes a more challenging task. The network layer further complicates this task, as data from different locations moves through the network at varying speeds. Furthermore even the origin of data on the same link becomes more uncertain through network jitter. Network jitter can be an especially strong factor to consider in the IoT environment, as data is often not transmitted via wired connections and over long distances. To avoid the influence of the network, the lifetime of a data source is best determined as close to the source as possible (e.g. in an ideal case this is the sensing node itself) and ensuring data locality and locality of computation [**?**].

The aspect of *when* data originated is not the only interesting factor that becomes harder to determine when moving further away from the data source. Answering other questions such as *what* created this particular data element and *where* was it created becomes a considerably daunting task when the data has already moved away from its origin. A possible problem that could occur is a faulty sensor providing the system with bad data. If the origin of the data is known, the problem can be isolated and the sensor replaced. One can imagine the difficulty of even detecting a faulty sensor when there is no knowledge of what data was provided by each different sensor.

Another problem is the *ad hoc* implementation model many IoT applications maintain. The applications have a high coupling between the lowest level of the infrastructure,i.e. the sensor data, and the higher level application layer. This results in projects having a very limited lifespan. As the IoT environment keeps changing rapidly, a system is required to be able to adapt to these changes as well. With ad hoc implementations the necessary changes are often too major, and the existing system has to be entirely discarded when the environment changes. This statement also holds for the usability of the generated data. By creating an ad hoc environment the data is interpreted on a system level, without any considerable effort to annotate it in such a way that it could be reused easily for other projects. In other words, by skipping the step of data enrichment, it is also not possible to make use of other existing systems in the application environment.

The above considerations lead to the problem that, in the long run, an architecture model that is fully Cloud centric may become insupportable. Hence there is a need to identify which parts of the processing and reasoning that are present in the general IoT landscape can be pushed away from the Cloud, closer to the data source. In addition to identifying these tasks, it is also necessary to determine how this can be achieved architecturally. Adaptations made to the processing chain need to be complementary to the IoT environment.

Problem definition

In all, based on the above considerations, the problem can be defined, in a general setting, how to offload part of the processing and reasoning from Cloud platforms to lower levels of IoT, that is, to the edges of the Internet.

In a more technical language, ***the problem can be formulated as that of specification, design and implementation of new layers in the architectural stack from IoT layer to Cloud application layer, where part of processing and reasoning over IoT data stream can be reasonably distributed to avoid performance bottlenecks, i.e. to achieve efficiency and scalability under real time requirements***.

Chapter 3

Objectives and project scope

In the previous sections the application domain and context have been introduced, as well as elaborating some of the problems. The attentive reader will have noticed the huge difference in focus between the introduction section and the following problem statement. The introduction motivates the need to solve the issue of detecting potholes in a timely fashion, this can be formulated more precisely as follows:

> ***What design choices lead to an efficient system for online reasoning over a car's sensory data?***

This question however, is a very niche scenario that holds little value in general. The processing of a car's sensory data to detect potholes can be regarded as a more specific scenario of general event processing in the IoT. Hence, the considerations made in the problem statement lead to the following:

> ***How can the designed model be generalized to an efficient architecture to perform real-time reasoning in an IoT event processing context?***

Which will, more than the specific case of processing a cars sensory data, be the focus of this dissertation.

Guided by these research questions the project scope and objectives can be accurately specified.

Efficiency In this case, by efficiency, the optimal usage of available hardware is considered. It is not the aim to optimize processing algorithms nor is it to lower the overall processor usage of each component in the system. In this regard the previously mentioned edge computing paradigm is key to further enable the usage of all system components.

Flexibility The designed system must offer a certain degree of flexibility, in order to adapt to the possibility of a changing environment. Changes in the environment often include a change in data source or type. Ideally, the designed model has to be both capable to deal with such changes, as well as being easily reusable in other scenarios.

Event-based system The focus of the project is that of event processing. In addition, due to the flexibility objective, a common data model (i.e. the semantic technologies) will be considered. Hence the system focuses on the processing of the data in a semantic and event-based way. Furthermore the assumption is made that it is clearly possible to define these events, so a complex event processing approach can be used. The goal is not a project based on machine learning techniques to find patterns that are unknown.

Real-time reasoning With reasoning in this scenario, the necessity to generate some kind of new result from the data is imposed. The reasoning component is restricted to reasoning over streams in real-time. Batch processing of data is not a part of the project scope.

Car sensory data The use-case and goal for this project is the processing of a car's sensory data in order to detect potholes in the road. This implies some assumptions regarding both the data set and the environment of the sensor node. The sensor node is not a resource constrained device, as a power source is readily available in a car, however, this is not a limitation for the study as any IoT event sensory data can be handled. The nature of the data imposes a streaming context with heterogeneous data, as a car has a wide variety of sensors.

Generality Although the solution in this thesis is exemplified for car sensory data, the proposal is of generic purpose. Flexibility and generality partly overlap, as attempting to create a solution that can be applied in different scenarios already implies a certain degree of generality. In order to preserve the generality of the architecture, the model must be applicable in the majority of IoT event stream processing cases. Hence considerations regarding the expected properties of the various architecture components in the general case, such as expected processing power, will be key for the assessment of this generality.

Requirements beyond the scope of this thesis It should be noted that there are also other requirements such as security, data privacy or energy-aware computing, which are very important in the context of IoT data stream processing but are beyond the scope of the thesis.

Chapter 4

State of the art

4.1 Semantic data streams

In this section, the concept of semantic data streams is introduced. In order to do so, the semantic web technology basis is first elaborated. Then some properties of the different IoT data streams are explained. Finally, the RDF-Stream format is discussed as a means of representing semantic stream data, with TripleWave as a possible means of generating these streams.

4.1.1 Basic concept

The first issue in the process of designing an independent, event driven IoT architecture is the notion of meaning. At the edge of a system, raw data is generated by sensor nodes. Due to its heterogeneous nature, problems arise with relation to interoperability and interpretation. In order to be able to define events, it is first necessary to have interpretable data. To make data machine-interpretable, so called metadata is added. Metadata is data that provides additional information about data, based on pre-existing knowledge. This metadata is often described using markup languages, e.g. XML and JSON. This is also referred to as semantic markup.

The addition of semantic markup allows the definition of attributes through domain specific vocabulary tags. This improves the specification of what kind of data is being passed, but still leaves a few issues. The vocabulary used for the tags has to be agreed on, which results in many domain specific standards for the metadata. Furthermore, semantic markup only defines *what* the data is about. It does not enable the description of *how* it is related to other elements. The latter issue was already discovered and tackled in the context of web technologies, and led to the establishment of the various semantic web standards, set by the W3C.

New challenges were then introduced by the IoT age and adaptations were made, hoping to reuse the advantages of the semantic web technologies in data stream environments.

4.1.2 Semantic Web

The semantic web is an extension of the original web, with the twofold goal of both linking as much data as possible and giving a well defined meaning to the data, such that it is more suitable for both computers and people to co-operate. This resulted in the creation of the semantic web stack.

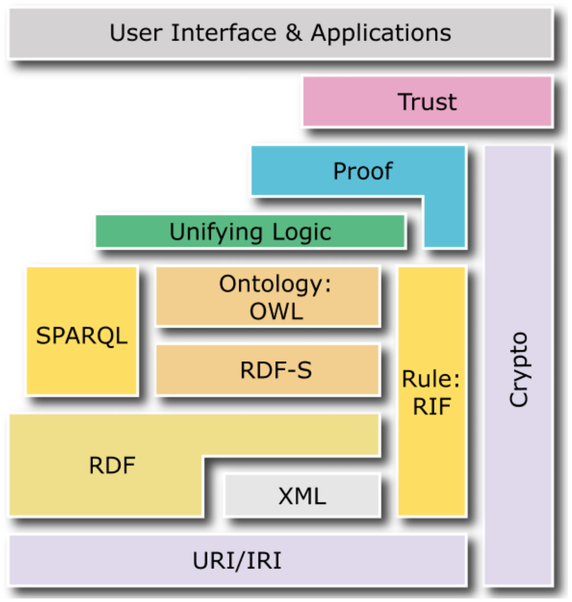


Figure 4.1: The semantic web stack [5].

The basis of the stack shows that everything is built upon on the existence of web-unique identifiers. The Resource Description Framework (RDF) uses these identifiers to describe relationships between resources on the Internet. This is done through statements in the form so called ”triples” that have a *subject-predicate-object* structure. Originally RDF was suggested as a standard to be built upon the XML markup language, but later it was extended towards other formats as well [6].

The basic RDF specification allows for the definition of relationships between objects, but it does not contain a distinct ontology for describing these relationships. A basic ontology that's focused on describing the structure of data, i.e. the hierarchy, was added in the first extension called RDF-Schema (RDF-S) [7]. A far more extensive ontology is included in the Web ontology language (OWL). The OWL standard includes semantics for logical expressions, e.g. equality, restrictions and cardinality [8]. Another key difference between OWL and RDF-S, is that OWL also introduces the concept of invalid usage of the vocabulary. The extended vocabulary in OWL allows contradictions to be detected. An ontology with contradictory definitions is considered inconsistent [9].

The W3C acknowledged that no single one-fits-all rule language could possibly satisfy the needs of different paradigms used in knowledge representation and business modeling[10]. Hence rather than developing a standard for rule systems, the Rule Interchange Format (RIF) was designed to facilitate the interchange between different rule engines. The goal of RIF is to enable rule sets to be communicated from one system to another, using a dialect they both support. This requires the rule engines to have a semantics-preserving mapping to the RIF dialect[10].

Finally, SPARQL provides a means of querying RDF content. At the time of writing, there's no existing standard specifications for the trust, proof and security layers in the semantic web stack. For security, the usage of a digital signature is recommended.

4.1.3 IoT data streams

The IoT type (architecture, application domain requirements, etc.) greatly influences the nature of the available data. This holds in particular in regard to the data generation rate, its heterogeneity and the processing requirements. The challenges to tackle in IoT systems are numerous [11]. Some of these challenges existed already within the scope of the web technologies, but new challenges have emerged due to nature and characteristics of IoT systems.

The semantic web technologies largely considered data in a static context. However, the IoT environment often involves data streams. In order to fully understand the difference in nature between static and stream data, the definition of stream data can be considered as an *unbound sequence of time-varying data.* Dissecting this definition reveals some of the important aspects.

Data variance The time-varying part of the definition actually reveals multiple challenges. The first issue is the variance of the data. The assumption of a static context no longer holds. In fact the data might vary at a very high rate, which is indicated by the term *velocity* in the famous "5Vs" of big data (velocity, variety, volume, value and veracity). The rapidly changing of the data allows the data to be considered as a continuous flow, which was not the case when the SPARQL standard was designed. Hence it does not include any semantics for continuously querying data, nor is it optimized to do so.

Time dimension Another issue is that the notion of time becomes very important. Knowing when a certain data object originated greatly influences reasoning. In many systems, data will have a limited lifespan where it is useful or valid. For instance, for measurements of latency in a network, the relevant value could be limited to the current measurement, while the values of an entire month or year could be relevant for analyzing power grid usage. Furthermore, due to sequential nature of the data, the detection of spatial events is often desired, thus, the ordering of the data is of great importance.

Data generation rate Dealing with the velocity and spatial nature of the data is a general issue in processing data streams, however IoT streams pose additional requirements that are determined by their application domain. For example, in environmental monitoring, devices are required to maximize their battery lifetime. This results in a need for efficient processing and low network overhead. The processing speed, however, is usually not as important in these cases.

In a production environment, the power usage of the sensing devices and processing is not a concern, as power is readily available and the usage is marginal compared to the production line. The data rates in these kinds of environments can be very high and there may be a need for instantaneous processing. An error in the production line that is not detected within seconds or even milliseconds may well have resulted in thousands of erroneous products or worse.

Security, privacy and non-invasiveness The health-care domain imposes various requirements. The body sensor networks used to observe patients also must operate under low power conditions, while further imposing a need for security (privacy) as well as a certain degree of near real time processing. Furthermore, issues such as noisy data or loss of data must be tackled, as crucial data may be lost. Dealing with security and data uncertainty has proven to be a difficult task, and remains an open issue in the semantic IoT.

4.1.4 RDF-Stream

The RDF Stream Processing (RSP) Community group was created to find ways to define a common model for handling RDF Streams. Their research resulted in the proposal of an extension to the RDF standard called RDF-Stream, in particular in order to deal with the time related problems. At the time of writing, it has not been accepted as an official W3C specification and is marked as incomplete[12], even though it has been widely adapted.

In the RDF-Stream draft [12] a proposal is made to use time-annotations. This can be done in two ways: using a description of data as a point in time, or as existing within a specified interval. Three types of metadata are recommended to use in this case:

- In case of a single point in time annotation, the use of either the *production time* (time when the data element was produced, if possible) or the *receiving time* (time the data element is made available to the RSP) is recommended.
- In case of an interval, the annotations *start time* and *end time* can be used to specify the validity of the data.
- A *timestamp predicate* should be used for a timestamp presence. The use of *prov:generatedAtTime* or *ssn:observationSamplingTime*[1] is suggested.

It should be noted that in a lot of cases, this time specification does not have to be an actual time value. Specifying the order of the elements in an

[1]This recommendation was made by the RSP at the time. In the newer version it is now equivalent to sosa:phenomenonTime.

incremental way is sufficient to allow processing of the data. Furthermore, the possibility of using the order of arrival at the RSP as an implicit timestamp is mentioned for RDF streams that do not have time annotations upon generation[12].

An RDF-Stream should be identified with an internationalized resource identifier (IRI). There are no further requirements set regarding this IRI. There is no specific indicator that a IRI is an RDF stream.

As a final note, it is worth mentioning and RDF-Stream is nothing more than a stream of RDF graphs that are required to have at least a time annotation. The timestamps are added using standard RDF triple definitions. Hence RDF-Stream still is a subset of the original RDF standard and is fully compliant with it.

4.1.5 TripleWave

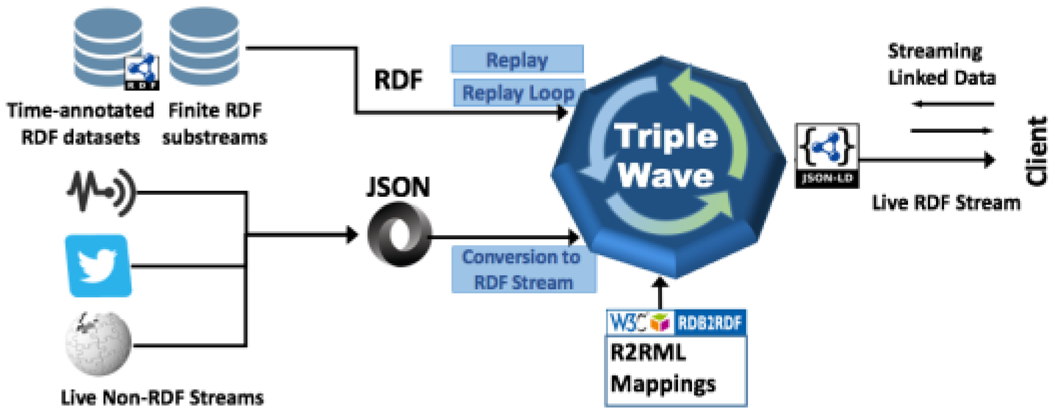


Figure 4.2: The architecture of triple wave, for generating RDF-Streams [13].

TripleWave is an open source tool created in collaboration by the Politcenico di Milano and École polytechnique fédérale de Lauasannea. Its goal is to provide a means of publicizing RDF streams on the web. It is provided as a Javascript and can convert web-streams to RDF-streams. In order to do so it uses the RDB to RDF Mapping Language (R2RML)[14]. It also provides support for replaying an RDF dataset, which can provide useful for applications such as benchmarking, system evaluation, and simulation. Furthermore, it is also possible to loop over these datasets in order to simulate an infinite datastream.

It is important to notice that the TripleWave framework runs on top of JSON formatted input and generates a JSON-LD RDF stream as output. Due to this assumption, additional conversions may be needed to match input data for the TripleWave processing, or output data for the stream processor.

4.1.6 Summative evaluation

The RDF model allows the description of how a resource is related to other resources. This offers two key advantages. The first one being that well known descriptions can be used to describe the relationships between resources, which leads to an extensive, shared data model. The second one being that by using unique identifiers for all resources on the web, the content of the data is greatly enhanced by linking. When creating a statement that uses a referable identifier, the available information is greatly amplified, similarly to a reference list for an academic paper.

However, in order to reuse this model for an IoT streaming context, some of its characteristics may prove troublesome. The time dimension is one of the problems that has been identified by the RSP group, but is not the only property of IoT streams that proves troublesome. The RDF annotations greatly expand the size of the data, which is amplified by the generation rate of the IoT. Furthermore, identifying a stream element is problematic in multiple ways, as the IoT is highly dynamic, so the descriptions may need to be updated often. Additionally, the question can be raised if stream elements should be referable at all, because it is not common for an IoT application to make all of its data permanently available online.

4.2 Semantic stream processing

Stream processing systems have the goal of continuously evaluating incoming data in order to find certain data of interest.This data of interest has to be found real-time (generally within a few seconds) or near real time (within a few minutes). This is in contrast to batch operations, where the data is stored in a storage layer and then queried for processing at a later time. One particular data pattern of interest is an *event*. The most primitive of events are created by evaluating basic relational operators (equality, more or less than, ...) over a single data attribute at a single point in time. Multiple of these events can be combined to create composite or aggregate events.

The relationship between the description of these types of events and the querying of a relational database is obvious. Imagining taking a snapshot of a stream at a single point in time, one could save and perform a relational database query upon the data to find these kinds of events. This approach to reasoning with stream data is called data stream management systems (DSMS) [15]. The shared characteristic of the systems designed using this approach, is that they implement windows to determine which data elements are relevant. These elements could be considered a part of the "snapshot". Windows will either accept elements that have a time specification within a certain interval, or count a specific amount of elements. The usage of these windows encourages a single point in time annotation of the data.

The advantage of these type of implementations is that they can build on the established knowledge regarding relational databases. The disadvantage is that DSMS approach makes it hard to detect spatial events. These are patterns in time, such as sequences, the repetition of a certain event or element, or even the absence of such a pattern (anomaly detection). In order to tackle these kinds of events, the complex event processing (CEP) approach was created. These families of processors have distinct semantics for defining complex events, given the idea of time annotated data and using the principles of primitive and aggregate events. There are multiple approaches to this processing paradigm:using composition operators, production rules, timed finite state machines and logic languages [16]. The *fire* event is a commonly used example, consisting of the detection of smoke followed by the detection of a high temperature within a short time frame.

In the following sections a handful of the various engines that specifically implement these techniques for reasoning with semantic stream data are provided. Starting with the DSMS based approaches, followed by the CEP approaches. The basis for them is the SPARQL language used in the semantic web.

4.2.1 DSMS approaches

4.2.1.1 C-SPARQL

Continuous-SPARQL (C-SPARQL) [17] is a language for continuous queries over RDF data. It extends the SPARQL 1.1 standard, used for querying RDF data in the semantic web. In order to achieve this, a *register* query statement is added, that causes the registered query to be run at frequency specified by a *computed every* clause. The register query also specifies the format of the output - as a new RDF stream or as a query result, possibly in the form of RDF graphs or tables of variable bindings. In case of an RDF stream output, an additional *construct* statement must be added to specify the generated triples. The timestamp of the generated RDF triples will match that of the query execution time, regardless of what the timestamp of the involved triples is. An RDF stream output can be reused for processing in other queries.

In order to execute these stream operations, C-SPARQL assumes that each RDF triple in the stream contains a timestamp that does not have to be unique, but must be increasing. In case multiple timestamps have been bound, the most recent timestamp of the bindings is taken. The absence of a timestamp results in a type error. A time window is then used to query the relevant data from the stream. C-SPARQL implements both a tumbling window and a sliding window.

C-SPARQL does not implement specific operators for performing the detection of temporal events, but does provide a function to query the timestamp of stream data elements. Hence it is still relatively simple to construct queries for doing so by using this timestamp. In this way C-SPARQL supports the processing of complex events to a certain degree. A restriction in this regard is the fact that all the relevant data for a temporal event has to be contained within the same window of the query [18]. A bigger window requires more memory, so the detection of temporal events that are spread wide in time is troublesome.

The architecture of C-SPARQL uses a modular approach. In order to handle the continuous queries, a query translator is used to integrate existing technologies. A continuous engine (using both Esper and Jena) is used to determine relevant triples from the stream. The outcome of this process generates a snapshot of the data, that is processed by a regular SPARQL engine.

4.2.1.2 SPARQLstream

Another approach based on the delegation of continuous queries to existing engines is proposed in SPARKQLstream [19]. While similarities with C-SPARQL are clear, it should be noted that SPARQLstream is in fact very different in execution. C-SPARQL uses a delegated engine to execute the window operators to create a snapshot of the relevant data, but the processor still processes RDF data using a SPARQL engine. In the SPARQLstream approach this is no longer the case. The complete processing of the data is done using existing relational database stream engines.

In SPARQLstream the incoming RDF-streams are translated to relational data using the R2RML language. In order to process querying of the data, a morph-stream engine is used. The morph-stream engine transforms the query in a two-step process. The first step of the process is to translate the query into relational algebra expressions that have been extended with window constructs. In the case of SPARQLstream, this window is exclusively time based. This intermediate step brings two advantages:

- Logical optimizations can be applied (by applying join and union distribution, and pushing down projections and selections)
- More flexibility regarding the used relational stream engine, e.g. translations to SNEE and ESPER [20].

This algebraic form is then translated into the respective language of the engine. This implies that the limitations of the target language have to be taken into account as they may prevent a complete translation, e.g. in SNEE statements for sequences and disjunction are not supported [21].

The final step in the process is to translate the resulting tuples into RDF triples. When this inverse mapping happens, the resulting RDF-Stream is marked as a *virtual RDF-Stream.* Its IRI is accordingly described in a virtual stream element. This annotation is made since the stream is derived from the data resource, unlike RDF-Streams that are created as a result of a query upon RDF data in C-SPARQL.

While the SPARQLstream approach no longer relies on a SPARQL engine, its query semantics still are based on an extension of the SPARQL1.1 standard.

4.2.1.3 CQELS

The Continuous Query Evaluation over Linked Streams (CQELS) language attempted to answer possible performance issues that occur by using the "black box" approach, which uses translations to existing processing engines for dealing with stream processing [22]. In order to attempt to further improve better query optimization, an engine was designed that implements the necessary operations for dealing with streams. More precisely, the CEQLS engine implements:

- window operators: Used for extracting triples from an RDF stream, given a triple pattern and a time window.
- relational operators: Traditional relational database operators, e.g. join and union.
- Stream operators: An operator needed to generate RDF streams from the sets, that resulted from the relational operators.

Details on the semantics for these operators can be found in [22].

The resulting CQELS query language (CQELS-QL) thus extends the SPARQL 1.1 language. A query pattern for applying window operations on RDF streams is included in the StreamGraphPattern. The various key-

```
StreamGraphPattern ::== 'STREAM'[Window] VarOrIRIref{TriplesTemplate}
            Window ::== Range | Triple | 'NOW' | 'ALL'
             Range ::== 'RANGE' Duration('SLIDE' Duration)?
            Triple ::== 'TRIPLES' INTEGER
          Duration ::== (INTEGER 'd' | 'h' | 'm' | 's' | 'ms' | 'ns')
```

Listing 4.1: The StreamGraphPattern, in enquoted capitals key words that are used in the query language [22].

words are used in the creation of different types of windows. For a time based window, the *range* keyword must be used to specify its duration, which in turn is specified by an integer value and a unit of time. Additionally, the *slide* keyword can be used to specify how fast a sliding window moves, otherwise a tumbling window is assumed. For a triple based window, the keyword *triples* is used to specify the amount of triples that are kept within the window. The keyword *now* specifies that only triples that have the current timestamp will be kept. The keyword *all* causes all triples to be kept in the window.

4.2.2 CEP approaches

4.2.2.1 EP-SPARQL and ETALIS

EP-SPARQL (Event-Processing SPARQL) is a processing engine that uses a CEP approach.In fact, it was designed with the issue in mind that the other systems at the time all used a DSMS approach and hence struggled to detected particular sequences in the data.

Like the previous implementations, EP-SPARQL is an extension of the original SPARQL language. More specifically, it implements the logical operators *SEQ, EQUALS, OPTIONALSEQ* and *EQUALSOPTIONAL*[23]. The syntax for defining temporal relations using these operators is the following:

- P_1 SEQ P_2 joins if P_2 occurs after P_1
- P_1 EQUALS P_2 join if P_1 and P_2 occur simultaneously.

The *OPTIONALSEQ* and *EQUALSOPTIONAL* operators are time-sensitive variants of SPARQL's *OPTIONAL* query syntax, which allows to make additional queries that will not cause the overall query to fail if they cannot be met.

EP-SPARQL does not implement a window based approach, but rather assumes the interval based RDF-Stream model. In order to reason with these intervals functions for obtaining the duration, start time and end time of the data are implemented.

Another important aspect to note is that EP-SPARQL does not implement a specific approach to handle negations. In order to achieve negation, i.e. the detection of the explicit absence of a certain triple pattern, the pattern must first be queried in an optional construct with the result binding to a specific variable, followed by a filter that states that the variable is not bound. Hence the optional pattern has failed to find a match, and thus the absence of a pattern has been detected.

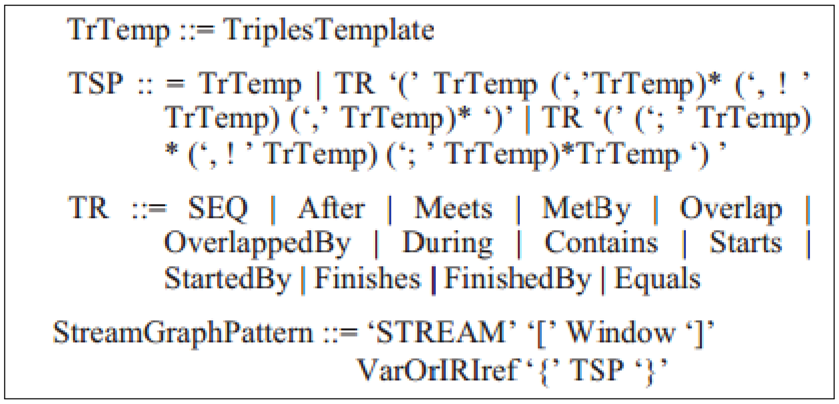
The EP-SPARQL language was developed in order to enable the ETALIS engine to be used in real time semantic web applications [24]. The ETALIS engine is designed to be a pluggable environment for various prolog-rule based systems. These prolog systems are then used to generate event streams, based on prolog code that is generated by parsing the native ETALIS language for events into binary rules and compiling the binary rules into specific prolog code.

By extension this system can be used to process EP-SPARQL queries. This simply includes more parsing steps, to transform the EP-SPARQL syntax into the internal prolog format used in the ETALIS engine.

One of the advantages of this approach is that the background knowledge included in the semantic annotations is also parsed to prolog code and hence passed on to the prolog system. The prolog system may then be able to use this to infer implicit information. Another advantage is created by the pluggable nature of the environment. Certain prolog systems may be more suitable depending on the nature of the queries. If it is known that a specific type of query (e.g. aggregation process for environmental data) will be used more often, an appropriate prolog system with better performance can be selected for this.

4.2.2.2 CQELS-EP

An extension to the original CQELS engine has been made in [25]. While the CQELS engine does implement a native, "white box" approach to processing semantic streams, it is incapable of temporal correlation of data. To this end, the aforementioned StreamGraphPattern is modified, by replacing the *TriplesTemplate* component with a *triple sequence pattern*. This consists of multiple triple templates connected by temporal relations. Other adaptations



```
    TrTemp ::= TriplesTemplate

    TSP :: = TrTemp | TR ‘(’ TrTemp (‘,’TrTemp)* (‘, ! ’
             TrTemp) (‘,’ TrTemp)* ‘)’ | TR ‘(’ (‘; ’ TrTemp)
             * (‘, ! ’ TrTemp) (‘; ’ TrTemp)*TrTemp ‘) ’

    TR  ::=  SEQ | After | Meets | MetBy | Overlap |
             OverlappedBy | During | Contains | Starts |
             StartedBy | Finishes | FinishedBy | Equals

  StreamGraphPattern ::= ‘STREAM’ ‘[’ Window ‘]’
                         VarOrIRIref ‘{’ TSP ‘}’
```

Listing 4.2: The new StreamGraphPattern in CQELS-EP. Various temporal relations have been defined.

that are made are in regard to processing these complex events. CQELS assumes a point-in-time data model, hence an event expires when it is no longer part of the window. A correct window must be found, such that the complex event is always detected. Furthermore CQELS has an incremental implementation, which discards previous results even when they have not yet expired. Such behavior is undesired, as previous results may be needed in the processing. This leads to possible incorrect output or a need to reprocess a query to re-obtain the previous result. To solve this, caching mechanisms are implemented.

The implementation of the temporal logic is largely based off an earlier adaptation [26, 25]. It appears to be the case that both implementations are being made separately. At the time of writing, one of the two implementations [25] has not been made publicly available, while the other one [26] is announced to be released on the CQELS github page, June 2018 [27]. This adaptation promises support for CEP, persistent streams, RDFS/OWL reasoning and websockets, as well as supporting the syntax recommendations made by the RSP group [12].

4.2.3 Summative evaluation

The various processors are designed to continuously query data from a stream source, in order to find possible data of interest. More development will be needed in order to achieve full fledged IoT semantic stream processing. The processing engines are all stand alone models, which are suited for processing on a single centralized device. Such a centralized, standalone approach does not match with the need for clustering and cloud computing that exists for dealing with the overhead generated by the *Big Data* associated with the IoT. Although some have been extended towards this end [28], their code base is not publicly available[2] and there is still the problem regarding their expressiveness.

Event processing in an IoT streaming context implies dealing with multivariate data and possibly highly complex patterns. Since the current expressiveness of these engines is mostly limited to simple relational database calls that do not support temporal patterns, they will be insufficient for a large amount of scenarios. The planned expansion for CQELS to support

[2]Although a distributed version of the ETALIS engine appears available, `https://github.com/sspider/etalis/tree/master/dEtalis`, it is no longer being maintained.

temporal patterns such as sequences is an important step along the way for semantic stream processing, especially considering a cloud implementation for the original CQELS engine was made. The expressiveness of the added CEP operators will greatly determine the usability of the systems, as its scope is limited by the possibility to accurately specify the event of interest.

4.3 Reasoning

The detection of events involves a process of selecting data of interest from a dataset or stream. Reasoning takes the next step. The goal is to produce new results from original data. In this regard, a very important concept within the semantic context is that of inference. Inference is the process of drawing a conclusion from observations. Due to its importance in this context, it is elaborated briefly in subsection 4.3.1.

The inference of new knowledge is just one of the possibilities for reasoning over events. Another possibility involves the handling of events. When an event occurs, certain actions may need to be taken. In this context two groups of reasoning systems can be considered: production rules and ECA rules. Production rules or business rules specify actions to be taken when a certain event occurs in an *on* event *do* actions format. The name production rules is related to the fact that a lot of these systems were created in a database context. Hence the action to be taken is the production of a certain element in the data base. ECA rules are similar to production rules, but include a condition that has to be validated before the action is executed.

4.3.1 Inference

The semantic web expands the "meaning" in the available data. The addition of these semantics has the goal of enabling machines to process the data more efficiently, but also more intelligently. One goal in particular is to expand the general knowledge, by inferring implicit information from explicit data. The inference process is often referred to as *reasoning* in the semantic context, and the RDF and OWL standards help in doing so. However, because of the extensiveness of OWL's semantics, the inference of new information is often a very computationally expensive process.

In some cases, the reasoning process can even become indecisive [18]. A simple example taken from the W3C OWL reasoning examples [29] shows how the inference process could work (see Listing 4.3).

```
Class(a:bus_driver complete intersectionOf(a:person
        restriction(a:drives someValuesFrom (a:bus))))

Class(a:driver complete intersectionOf(a:person
        restriction(a:drives someValuesFrom (a:vehicle))))

Class(a:bus partial a:vehicle)
```

Listing 4.3: An example of class inference possibilities in OWL

The OWL statements define a class *bus_driver* to be someone who drives a bus. Furthermore a driver is someone driving a vehicle and a bus is a subclass of the vehicle class. Hence the engine will infer that a bus driver is actually a subclass of the driver class.

In a similar way, the inference process can also be used to find inconsistencies in the data. Such inconsistencies can be for example multiple class definitions of the same object, or contradictory relations e.g. X is a child of Y followed by a definition Y is a child of X.

While the concept of inference is in fact one of the important aspects when discussing semantic technologies, the scope of this project is limited to reactively handling events in a semantic context.

4.3.2 Production rules

Production rule systems are systems that use rules to solve a certain problem, mimicking the reasoning process of a human expert. Hence they are also known as production systems or expert systems. The basis of the system are rules, which are composed in *if-then* statements. The *if* clause is called the premise or condition, and the then part is called the consequent, conclusion or action [30]. As can be derived from the various terms, rules can roughly be used for two different purposes, one being the aforementioned inference process.

The other is to aid in the decision making process of systems. A production rule system can be used to perform various tasks such as classification, monitoring and prediction [30]. Important in this regard is the size of the rule set, which greatly influences the performance. Additionally, if more rules apply to the same given fact, a conflict resolving strategy has to be applied.

4.3.2.1 Drools

JBoss Drools [31] is an open source business rule management system. It provides an extensive framework for setting up rule based systems. It consists of several major parts:

- Drools Workbench: A web user interface for managing, authoring and testing rules.
- Drools Expert: The business rules engine.
- Drools Fusion: Adds complex event processing features to the system.
- jBPM: A modeling tool for managing work flows. It is meant to bridge the gap between software developers and business analysts. By using flow charts and domain specific representations, the process becomes more easy to understand by business users. It also increases the systems overall flexibility.
- Optaplanner: A constraint resolving engine for determining efficient paths. It solves scheduling puzzles.

Within the project scope it is most interesting that drools combines both the idea of business management and event processing. The choice to combine these two fields was made because of their similarities in requirements and goals. From a business rule point of view, rules are often defined based on the occurrence of certain events, effectively using them to perform a kind of event processing. Furthermore, both events and business rules need to adapt to rapidly changing environments [32].

Other similarities can be found both in regard to their functional and non-functional requirements. Business rules often need to perform pattern matching tasks and are often tied to processing time constraints [32].

4.3.2.2 Jena

Jena [33] is an open source framework specifically created for building semantic web applications. The Jena framework allows for reading, writing, querying and storage of RDF data. Furthermore it has components for setting up a SPARQL end-point to expose data on the net.

While the Jena framework is not designed as a production rule system, it does include various rule engines for inferencing over semantic web data. Because of its specific field, it has to be considered as an option when discussing the possibilities for the rule based, stream processing of semantic data events. In this regard, a study was done to compare the performance and possibilities of Jena and Drools in an event processing context [34]. One of the important conclusions is that Jena is unsuited for performing an inference task over stream data, due to the way the model works. Furthermore, although the inferencing process itself outperforms that of Drools when using semantic data, it is compensated by the time needed to insert the data into the model. Hence Drools is better suited for stream reasoning, unless it is necessary to be able to query the data or memory constraints apply [34].

4.3.2.3 Easy-Rules

Where Drools provides an extensive framework for creating a business rule management system, Easy-Rules [35] is designed to provide a simple framework for rule based processing in Java. The lightweight API allows users to define rules in various ways: using annotations, in an expressive way, or even using a rule descriptor in YAML format. Due to its simplicity, Easy-Rules allows for fast design of a rule system, but it is not concerned with issues such as conflict resolving, planning or process design which are integrated in Drools. It does not offer support for processing data in a streaming fashion either.

4.3.3 Event-Condition-Action (ECA) rules

ECA rules, originally proposed for active databases [36], are often used as a basis for reactively handling changes in event based systems. An ECA rule contains a specification of the *event*, which is a certain object, pattern or signal of interest. When such an event occurs, the engine will evaluate the *condition* specified. When the condition evaluates to true, the action that is specified in the *action* part of the rule is executed. In a CEP environment, these rules can be used to handle an occurrence of an event of interest. Such an event could be e.g. the detection of a hole in the road, for which the position has to be added to the database.

Various extensions to this model have been proposed depending on application context. One of the possible extensions is the realization that often when for an event certain conditions are not met, other actions may need to be taken depending on further conditions. In this regard two extensions were suggested [37], both combined resulting in an extended ECAA pattern.

```
RULE [ $R_{name}$
        EVENT       $E_{name}$
        CONDITION   $C_1 \ldots C_k$
        ACTION      $A_1 \ldots A_n$
        ALT ACTION  $AA_1 \ldots AA_n$         ]
```

Listing 4.4: The ECAA rule pattern [37].

The first extension is the suggestion to consider an entire list of possible conditions for a certain event and actions that are associated with it. This way there are less rules necessary to handle each event, making conflicts less likely. The second extension then is the realization that often, when conditions are not met, there is an action that should be taken exactly because an event occurred and its conditions were not met. A simple example for this case could be a web request without the necessary credentials. The server does not execute the request, because the condition of having the credentials is not met. However, it is still necessary to notify a user why his request was denied. Combining these two ideas resulted in the extended ECAA, event condition action alternative pattern. The specified alternative actions are to be executed in case the condition does not evaluate to true.

Some other adaptations to tackle possible issues include the ideas of ECA-P, ECAP and ECAS patterns [38]. In the ECA-P (ECA - Post condition) pattern, post conditions are added in order to avoid conflicting rules. The ECAP (ECA Parameters) pattern adds parameters to the ECA concept. These parameters are then used in the action process. By doing so the amount of data passed to the calculation engine can be lowered in object driven environments. Lastly, in ECAS (ECA sequence), the idea of using the sequence of previously taken actions in order to detect anomalies is introduced. The common factor among all these adaptations is the need for events to be handled instantaneously, and the usage of events as triggers to start the actions.

4.3.4 Summative evaluation

While the aspect of inference is important in the semantic web context, more efforts are needed to find ways to achieve inference over streaming data in an IoT context, due to the computational expensiveness of the process.

Production rule systems and streaming contexts however, are greatly complementary, especially for a task such as classification of data. Push-based semantics are desirable in a streaming context where the reaction time of the system is crucial. In this regard, the integration of complex event processing and streaming features into the Drools rule management system is most promising. In many cases however, there is no need for such an extensive rule system.

The ECA pattern and its various adaptations all bring a similar, reactive processing style. Rather than having extensive frameworks, the majority of these ECA based systems are created using simple scripts that use declarative programming languages. Such a design choice can be justified to a certain degree, because in the majority of these examples the reasoning layer is the both the final step in the processing chain, and performs a simple classification task. This results in small scripts for performing these tasks, where the expressiveness of the used declarative programming languages allows them to be adapted relatively easily if needed.

Chapter 5

System architecture

In this chapter the adopted architecture model is elaborated. In order to justify design choices a closer look at the use-case is needed in order to fully understand the systems functional and non-functional requirements. This is discussed in section 5.1. Next, in section 5.2, a more in depth look at the processing model allows the identification of complementary architecture models for event processing. The combination of the requirements and the processing model then lead to the designed architecture in section 5.3.As the proposed model considers multiple processing layers, it is important that the system is manageable in order to adapt to changes in the environment, or update processing tasks. To this end an API is developed and introduced in section 5.5.

5.1 Requirements

In order to derive the system requirements accurately and unbiased, it is necessary to take a step back from the project objectives and look more specifically at the use-case environment. The use-case of processing a car's CAN data specifies a clear goal of the processing: detecting potholes. Furthermore it provides information about the context: the challenges of big data and the IoT will need to be faced.

5.1.1 Functional requirements

The use-case scenario is that of multiple cars used to gather information about the road conditions such as *holes*. In order to gather this information, a cars CAN data is analyzed. The scenario imposes an environment in which processing done inside of the car is necessary, as the amount of generated data is high, but the network transport has to be done over a relatively big distance. This makes its cost considerable, especially when high throughput is required. The network transport is an even bigger concern for roads in remote areas, where 4G coverage may not yet have reached. In these cases the network may even become a bottleneck. Furthermore, the presence of a processing unit in the cars may not only be assumed, but is in fact a requirement in order to extract the CAN data and transmit it.

In reality, a car's CAN data is not accessible directly. This is because, while standards for CAN buses do exist, car manufacturers hide the unique configuration used for the bus in order to secure its data. In the project, the reverse engineering of the CAN data in order to reveals its configuration is considered to have been performed already, i.e. its configuration is considered to be known. The reason for this will be made clear in section 5.4.1.

The CAN data consists of many different types of data, and has to be filtered in order to extract data that is relevant to the pothole detection process. The filtered data then has to be processed in order to perform the actual detection of potholes. Finally, the detected potholes must be classified in order to allow them to be handled adequately.

The functional requirements generated by the use-case can hence be determined to be the following:

- The system must process a heterogeneous data stream.
- The system has to be capable of processing a big number of streams concurrently.
- The system must be capable of processing the incoming data in order to detect potholes.
- The detected potholes must be classified to allow the them to be handled correctly.
- The throughput the system has to handle is up to a few Mb/s for each car.

- The car's CAN data has to be filtered inside of the car, to comply with network restrictions.

Reconsidering the projects objectives, one more functional requirement can be added to this list. This is the one in regard to the modeling of the data in order to improve the flexibility:

- The system must add a common data model to the stream.

This requirement further stresses the need for filtering done at an early level, as the enrichment of the data is an extension and thus further increases the overhead.

5.1.2 Non-functional requirements

The majority of the non-functional requirements are related to the objectives of the project. In order to guarantee flexibility in the system, low coupling of each module of the system is needed. Furthermore the system should be easy to manage, in order to allow it to be reused in different contexts. This also requires that the coupling with the data source is low.

The system has to perform a data enrichment task in order to improve the processing efficiency at later levels. The semantic web technologies are an established standard in this field, and recently have started to find their way into the IoT [1]. Hence the system is required to be able to adapt this model.

As the project is focused on reactive reasoning with data, all processing is required to be done at least in Near-Real-Time. From a use-case point of view, it is hard to determine exactly how fast the reaction to sensory data that indicates a pothole has to be made. In this situation, a delay of up to a few minutes could even be considered acceptable.

Lastly, because this is an academic project, only open source modules should be considered for usage in it. In this way it is guaranteed that the used architecture may be reconstructed at later time if needed. Summarizing each of the non-functional requirements:

- The system must perform all of its task in at least Near-Real-Time.
- Only open source modules can be used.
- The proposed system has to be reusable in other contexts, without requiring major re-engineering.

- The semantic web technologies have to be integrated.
- The modules of the system must have low coupling with regard to each other and the data source.
- It has to be possible to manage the tasks of the system at a higher level.

It should be noted that there are also other non-functional requirements such as security, data persistence and energy consumption, which are very important in the context of IoT data stream processing but are beyond the scope of the thesis.

5.2 Event processing model

The goal in an event processing system is to discover a certain pattern of interest in a dataset. Depending on the complexity of the pattern, the approaches to do so vary.

The simplest type of event detection can be considered the detection whether or not data is present. The complexity of the pattern is minimal. Any data element available is an element of interest.

Going up in complexity the next level are the atomic events. These are events that are created using relational and/or logical operators on a single attribute of the data, in a single point in time. While the amount of operators does increase the complexity of an atomic event, the restriction on one attribute and a single moment in time determine its atomic nature.

A higher level of complexity is created in two different ways, that can possibly be combined.

One way is to complicate the description of the pattern in time. Depending of the complexity of such a time based pattern, the distinction can be made towards a complex event processing approach or a machine learning approach. A CEP approach requires the pattern to be definable. This is contrasted by the machine learning approaches, which are used in cases where patterns are too complex to be accurately specified. An interesting aspect in this regard is that the two approaches may be used to complementary. In some cases, describing the exact pattern may be too hard, but a more broad description can first be used to narrow the data down using the CEP approach. This reduces the computational time needed by the clustering unit, while the clustering unit can improve the accuracy of the processing done by the CEP layer [39].

The second way to increase complexity is to compose events that consist of multiple coexisting events. These composite events are particularly interesting from an architectural point of view, as the detection process can be split into multiple steps. First, each unique event can be detected separately, and then the coexistence can be detected. Furthermore by doing so, the various events composing such a composite event can be abstracted. By abstracting the events, the different processing layers become more decoupled, both from each other and from the data source.

While a hierarchical processing model with multiple layers appears to be a logical approach, there's one important issue to be tackled for this. As apparent from the overview of current approaches to event processing in a semantic context in section 4.2, the majority of approaches uses a point in time based model for the data. Therefore, implicitly the data's lifetime is limited to the used time window length. An event processor using such semantics, situated at a higher level in such a processing hierarchy, can only consider the time window it uses itself, and the timestamps included in the events provided to it. However, it does not have knowledge of the lifetime of each specific event it receives, and thus may conclude invalid results.

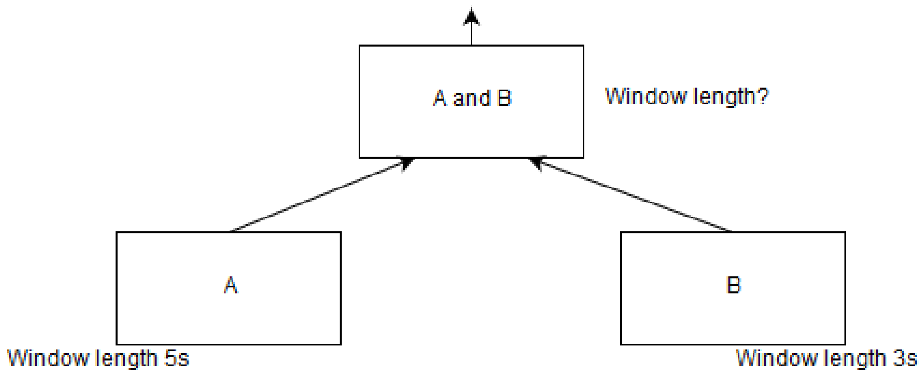


Figure 5.1: An example for the issue of determining a window when using point in time data representations. There is no single appropriate window for containing both A and B without altering one of their lifespans.

In Fig. 5.1 a simplified example to such an issue is provided. The top node should produce an event, when both event A and B occur at the same time.

However, event A and B have a different lifetime. If the window is chosen to be that of event A, a detection may be made even though the event B contained inside of the window has already expired. However, if the window size of event B is selected, some events may fail to be detected as some events of A are discarded prematurely. Furthermore, in an ideal scenario the top node should not be aware of the processing layers underneath it, so it does not even possess the knowledge of these windows.

5.3 Architecture overview

The architecture model adopted considers multiple processing layers in order to create an efficient and scalable environment to deal event processing in a semantic IoT context. In this section the general architectural model is elaborated. A more specific implementation of the model is described in section 5.4. A graphical representation of the architecture is shown in Fig. 5.2. As can be seen, the architecture is structured into five main layers, starting from the sensing layer at bottom to application layer, at top.
Hereafter each layer of the architecture is briefly introduced in terms of their characteristics, computation requirements and functions/tasks to be allocated at each of them.

Data sensing In any IoT architecture the lowest level is that of the data sensing (also referred to as Data generation). In the application case study considered in this thesis, the data is sensed / generated by a car's CAN bus and the on board GPS device. The generation layer is key in determining the system's throughput characteristics given that there is a data generation rate at this layer, which has to be accommodated in upper layers in order to ensure events are not lost.

Data preprocessing The preprocessing layer is situated in the first computationally enabled device closest to the data source. Both its tasks and the location of the layer may vary depending on where this device can be found, and how powerful it is. For instance, there could be a device with computational computing power to perform simple event filtering, detecting event boundaries, etc. before transmitting the data to its closest upper level. In general the possible tasks can considered to be the following:

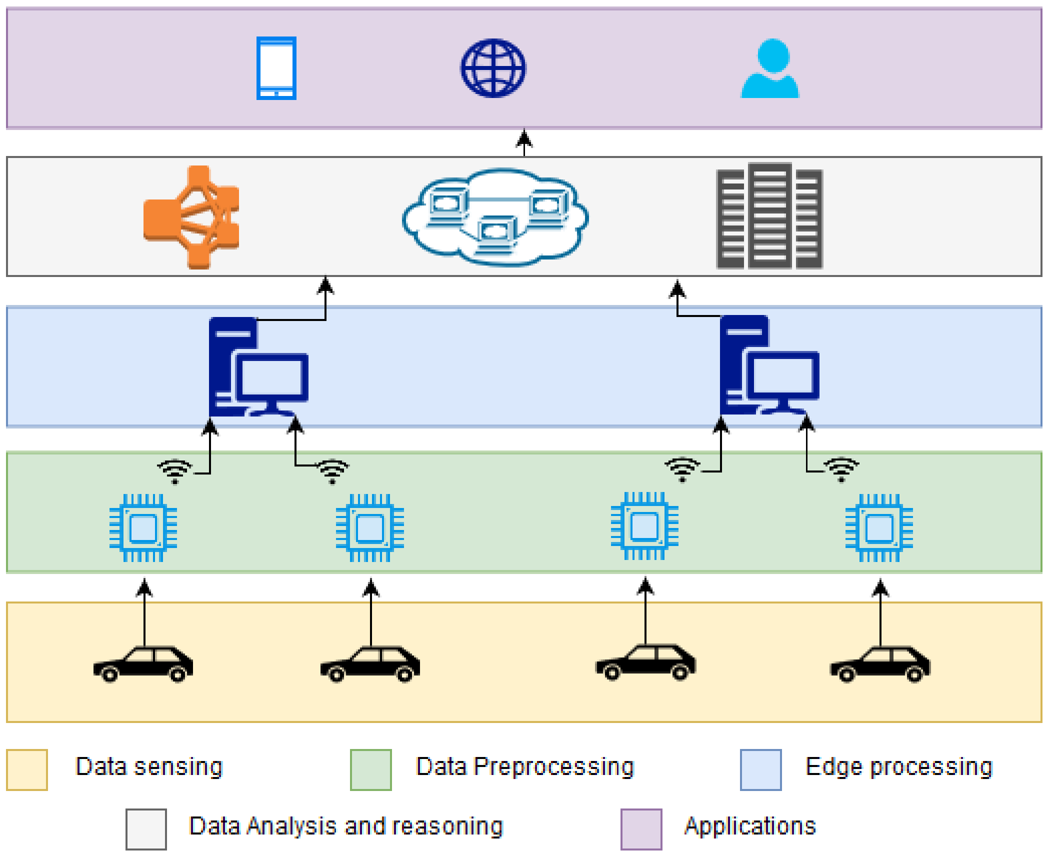


Figure 5.2: Overview of the general architecture model.

1. **Filtering:** A basic filtering process of the data. There are two possible tasks to perform in this process. One task is the filtering for data of interest, selecting the appropriate data attributes for the given context in case of multivariate data streams. The other task can be dealing with noise and erroneous data in uncertain environments.
2. **Event detection:** Any event stream processing assumes that some function can be implemented to detect the boundaries of an atomic event, that is, determining when the event start and when it finishes within the stream. It should be noted that this is possible for most event-based streams, which are built by appending events at the end of the stream as they are generated by

the system.

3. **Data enrichment:** This process involves the labeling of the raw data and attempting to add as much information as possible to it. This improves the processing of the data later on. An important task in this regard is adding the generation time to the data, for ordering purposes.

4. **Data analysis:** If the system is powerful enough, options of even performing more advanced computations such as CEP or machine learning algorithms become available.

Edge processing The edge processing layer can be situated at the first node where data is obtained from multiple sensory devices. If a preprocessing layer is not available, its task are performed here. This will often be the case in resource constrained environments. In the vast majority of cases where preprocessing is possible, the device will not be powerful enough to perform all of the aforementioned tasks, or it is not desirable to stress the device as such. Hence, the preprocessing will often be limited to filtering and simple event detection. Thus, the tasks of data enrichment is often a core task of the edge nodes.

The main functions of the edge computing layer are aggregating data and preparing it for further processing. Data enrichment is an important concept in this regard. Another good way to prepare the data is to intelligently group and segment streams. This way a high grade of parallelization can be obtained. For example, consider an edge node that receives data from multiple cars in the surroundings. The stream data can be grouped to create a stream of information for each road, which can then be computed in parallel. But, because this is an aggregated stream, it would require higher computing performance. Yet the job can be parallelized further by segmenting the stream into multiple time frames. Each segment can then be processed in parallel as well. Such a high grade of parallelization is possible in an event processing context, as events are independent of each other and can thus be identified, extracted from the stream and processed.

Reasoning and intelligent data analysis This is where core system tasks are performed. The architecture of this layer may vary depending on the tasks that are needed to be performed but will in any case offer a variety of reasoning and intelligent data processing (e.g. machine learning) functionalities. Typically these will be cloud or cluster computing environments, possibly linked to reasoning or machine learning libraries.

Applications On top of this architecture various applications can be developed.

5.3.1 The Frontier between IoT and Edge Processing layers: thin *vs.* fat IoT layer

In IoT systems, the layers of data sensing and data processing are usually not split on an architecture level. In general, there are two common cases to be found. Either an IoT device is present that is simply capable of sensing and transmitting data, or what can be referred to as a *smart device* is present, that is capable of performing some computations. Hence, a distinction can be made between computations performed inside the *IoT Layer*, grouping IoT sensing and processing, and at the consecutive edge processing layer. A decision needs to be made on the exact tasks to be performed in each of the layers. A major factor in this decision making is the computing capacity available at each level. If the IoT layer only has data sensing capacities, no more processing tasks can be performed there. However, in the case of a smart device, more tasks such as filtering, simple statistical analysis, error detection, etc. can be performed here.

The main contrast between the two layers is to be found in the expected computational power of the devices. While the presence and processing capacity of smart devices has been increasing, it is bound by limitations such as available space and energy consumption. By contrast, in the edge a certain computing capacity can be assumed (e.g. that of one or several Raspberry Pi) and the layer can handle processing tasks such as semantic processing and data aggregation. These are by nature the tasks of the edge computing layer. Semantic annotation increases the size of the data, and is hence not desirable at lower levels where data rates are more concerning. Aggregation is the natural task, as the edge was described as the first layer obtaining data from multiple sensory systems. Yet more preprocessing tasks can be added, depending on how "fat" the IoT layer is.

Deciding on a "*thin vs. fat*" IoT layer should take into account some other factors than just the available computational power. A thin layer minimizes the used power resources at IoT level. It also simplifies the model and does not set any hardware requirements. However, moving to a more fat IoT layer allows more data to be filtered through analysis, lowering the necessary network and computational specs at subsequent layers. Furthermore, certain tasks may require a low response time, which is easier to obtain if they are performed closely to the sensing layer, as network delays are eliminated. These advantages come at the price of needing more computational power at the IoT level, implying a higher power consumption. In the filtering process the decision making becomes more complicated, because the accuracy of the filtering has to be considered. A rough analysis implies the amount of data passing the filter is either too much, requiring more network bandwidth, or too little, losing some relevant data. A more accurate analysis, however, again requires more processing power. This results in a three way trade-off between system specs at later levels, computational overhead in the IoT layer and processing accuracy of the overall system. A balance has to be found between them depending on the specific system context.

5.4 Implementation

In this section we discuss the implementation of the architecture (see Fig. 5.2) presented in section 5.3. Additionally, the created API for managing the system is described.

5.4.1 Data sensing and Date rate

The data on a CAN bus is not publicly available as such. While the CAN standard has been defined, companies are free to select the IDs they use, the signedness of the signals, etc... As these are specific for each car, a generic reverse engineering approach has been attempted [40]. Unfortunately, the described hardware solution is currently still unavailable. Thus, obtaining real measurements of a car's sensory data while driving the road is not possible. Simultaneously, an ongoing master's project at the University of Antwerp is studying the simulation of CAN data. The data generated from the simulation project is used as the data source in this project.

In a real scenario, data is transmitted at various rates on the CAN bus, up to 1 Mbit/s [40]. If all data would simply be transmitted to the edge without filtering, this peak rate can be considered the necessary bandwidth to avoid possible network congestion. This is the necessary upload speed, that must be achieved real-time and over long distance. The primary candidates for transmitting at high data rates consistently, while covering a large area, would be the cellular network technologies. However, the upload speeds available in 3G network environments would prove insufficient. A study comparing Wi-Fi to 3G performed measurements driving a car at up to 30 MPH, and found an average data rate of 130 kbit/s for 3G [41]. Its successor 4G is capable of providing the necessary upload speeds [42] to transmit the full CAN data. The lack of 4G coverage in certain areas still remains a problem.

The data rate of the simulation can be determined more accurately. The dataset consists of sensory data generated at a constant rate of 100Hz. For each sensor, the transmitted data consists of a frame containing its ID, the length and 8 bytes of data, creating a frame of 10 bytes total. The ID is used to identify the content of the message. The length specifies how many bytes of the data are actually used. All unused data bytes are set to zero. A single message may contain content from multiple sensors, e.g. the rotation speed of each wheel is grouped together in a single message. Considering the throughput to be the required upload speed to upload all data, it can be found as in Eq. 5.1.

$$T = 100Hz * 10byte * n \tag{5.1}$$

where T represents the total throughput and n the amount of sensory systems. In case only a single sensory system is considered, this creates a throughput of approximately 1KB/s. Even if multiple sensory systems are considered, the data rate of 1 Mb/s seems to be far off. The differences

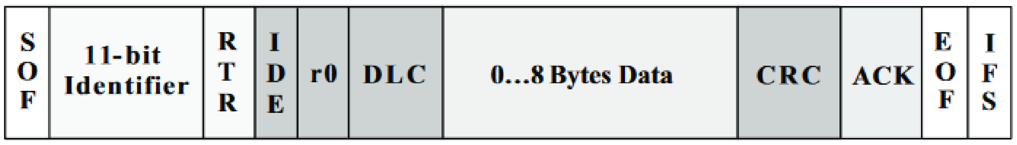


Figure 5.3: Structure of the standard CAN frame [43].

between the real and simulation data rate can be attributed to two factors:

- The constant generation rate of 100Hz is used to match the behavior of the VBox used in the reverse engineering approach [40]. However, in an actual CAN bus these generation rates are unique for each sensor.

- The protocol overhead in the simulation is considerably lower. As depicted in Fig. 5.3, a CAN frame has more mandatory fields than just the ID and the length indicator (DLC, data length code). A total of 51 bit of mandatory header content is added for every 0-8 bytes of data, whereas in the calculation for the simulated data an assumption of 2 bytes is made for ID and length field.

5.4.2 Data preprocessing

The data preprocessing is to be done inside the car. In the project, the tasks performed by the preprocessing layer are limited to the filtering of the multivariate data for its relevant attributes and anomaly detection. There are two reasons for limiting the tasks at this point.

The first being that, while the actual reverse engineering process is considered to be transparent, it should still be taken into account that the full computational power of the raspberry pi used in the car is unavailable. Secondly, a raspberry pi is an unusually powerful device to have this close to the sensing layer. In most cases, this will either be a smart device or a less powerful micro-controller. Leveraging the full power of a raspberry pi would create an unrealistic processing model for the vast amount of IoT applications, losing the generality of the architecture.

Anomaly detection

Discovering potholes in the data can be considered a specific case of anomaly detection. In the general case, anomalies are patterns in data that do not exhibit the expected behavior. The problem of anomaly detection has been studied extensively, because anomalies contain data of interest in a broad range of applications. Consequently, a broad variety of approaches exist [44]. Some of these can be adapted for processing streaming data.

Statistical approaches The oldest approaches to anomaly detection are the statistics based approaches. Assuming the dataset is generated by a statistical model, the chance that a particular data element is obtained from it is known. Values that have a significantly low chance to occur are considered outliers, which in turn can be considered anomalies. In the case of a Gaussian model, a simple method is to use a box-plot, which equates to considering any values that are not within 3σ from the mean to be anomalous. In a data streaming context, these kinds of methods offer a low computational complexity, but in many cases the data will not be generated following some statistic distribution.

Similar to the statistical approach, are the regressive approaches. In these approaches, a regression model is fitted to the data. The difference between the actual data and the regression model, often called *residual* here, can then be used to determine how anomalous the behavior is. Once again the main issue is that data is often not generated following a clear model [44].

Nearest neighbor-based techniques build upon the idea that normal data instances have close neighbors, while anomalies occur far away. These algorithms use the distance to its k nearest neighbor, or compute the density of the data around an instance. The local outlier factor technique [45] is one of the major contributions in this field. It compares the local density of an instance to that of his neighbors in order to detect anomalies. Nearest neighbor techniques are computationally more expensive than the previous methods, but can be applied more often in the general case.

Clustering techniques Clustering techniques can be considered similar to density based techniques, in the regard that they often require a kind of distance computation in order to do the clustering. The key difference is that each instance is compared to the cluster it belongs to, rather than the local neighborhood. In the case of clustering based techniques, an anomaly is either not part of a cluster, part of a sparse cluster or should lie close to their respective cluster centroid. One of the main concerns for clustering based anomaly detection is the high computational complexity [44].

Hierarchical Temporal Memory (HTM) algorithm In recent work, the Hierarchical Temporal Memory (HTM) algorithm has been adapted in order to do online, unsupervised anomaly detection [46]. HTM is a machine intelligence framework based on neuroscience. It models spatial and temporal patterns in time sequences. The framework itself does not produce anomaly values, but was adapted to do so by using some of its internal data. First, a raw anomaly score is made using the prediction vector $\pi(x_t)$ and the actual value $a(x_t)$. Both are binary vectors, which is the data representation used internally in the HTM framework. The raw anomaly value is computed by comparing the actual value with is prediction. This value is then used to the compute the anomaly likelihood. In order to determine the anomaly likelihood, a rolling window is used to calculate a normal distribution using the past raw anomaly values. The mean u_t and standard deviation σ_t, along with a moving average computed over a smaller range $\tilde{u}_t$ are then used in a Gaussian tail function Q. The anomaly likelihood L_t thus can be determined as in Eq. 5.2.

$$L_t = 1 - Q(\frac{\tilde{u}_t - u_t}{\sigma_t}) \tag{5.2}$$

An anomaly is detected by applying a threshold to this value. Interesting about this method is that it simply offers a means of comparing an anomaly score given the recent history of anomaly scores. Hence, this part of the detection method could be extended to other methods using anomaly scores as well. It should be noted however that this algorithm is more complex and require considerable work to tune it for achieving desirable accuracy. Likewise, it also requires more computing resources than, for instance, statistical based methods.

Data filtering

The filtering of the data is a simple process of reading only the relevant sensory data from the data file. In the use case of detecting potholes, the data of interest is the rotational speed of the individual wheels. Each wheel will be considered separate from the others. A smaller sample from the dataset is given in figure 5.4.

Approach using differences There's a clear difference in size between the spikes of the detected holes and the spikes in the data that occur when the speed of the car increases. A very simple idea is thus to try and measure the difference between consecutive points, and apply a threshold to the distance between them to determine if the cause of it was a hole in the road or not. Essentially, the pattern that occurs when the car drives through a hole is simplified to its peak to peak value this way. The result of the algorithm is a simple classification - a hole is either detected or not.

Simple statistical approach Another approach implemented is based on the concept of outliers in statistics. Although there is no clear presence of a distribution, this approach uses the observation that, when there is no pothole, the data is either stable or exhibits a step-like function (which is a

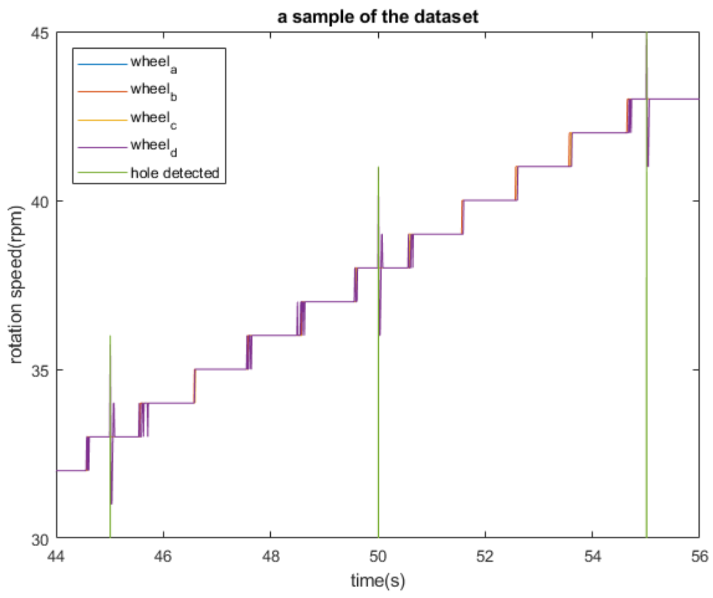


Figure 5.4: A sample of the dataset

result of the quantization levels of the sensor). When there is a pothole, the values spike up and down for a short period as depicted in figure 5.5. The lower values and higher values cancel each other when calculating the mean. In the case of a spike, it can be expected to be more standard deviations removed from the mean than a step.

- Center a window with N points around the point to calculate the anomaly score for.
- Calculate the mean and standard deviation of the window.
- Calculate the amount of standard deviations the current value deviates from the mean.
- Using the z-score, obtain the confidence interval associated with the z-score. The anomaly score is the inverse of the confidence.

This leaves two parameters to decide: the size of the used window, and the threshold for determining a value to be anomalous. These will later be investigated when evaluating the algorithm in section 6.3.3. An advantage of the used method is that the resulting anomaly value lies in a normalized range by default.

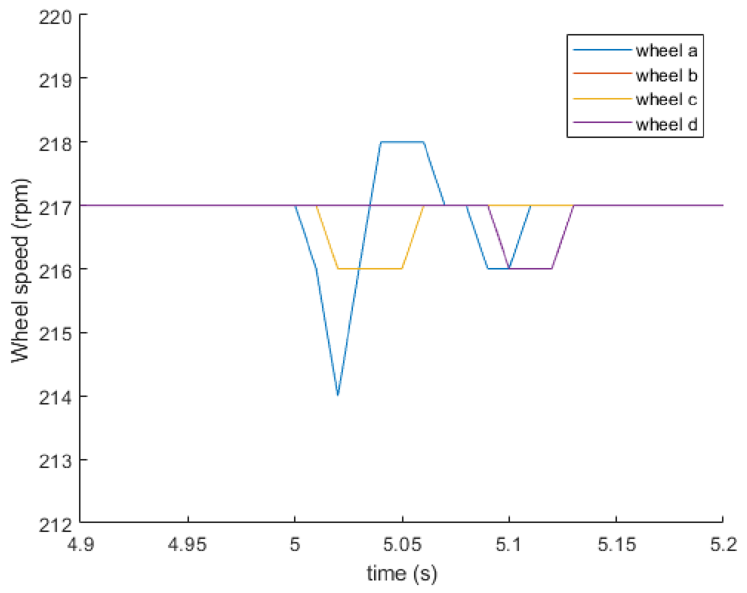


Figure 5.5: Occurrence of a pothole at wheel A. The rotation speed of the wheels fluctuates during a period of 0.15s.

Data communication

To communicate the data to the edge, the publish-subscribe based MQTT (Message Queuing Telemetry Transport) is used. It is an established standard [47] for topic based communication in IoT environments. It is lightweight, requiring little device resources. Furthermore, it has a low network overhead, which is highly desired in this use-case. The MQTT standard also provides options for delivery assurance (at most once, exactly once and at least once mode) and data availability for bad network environments [48]. Lastly, implementations exist for various programming languages, increasing the portability of the system. The topic based model itself simplifies the aggregation of the data, and can also later be reused in the API to manage all nodes that are located at the same level in the layered model.

5.4.3 Edge processing

Data enrichment

The first task of the edge processing node is data enrichment. The data enrichment process involves transforming the data stream into a format for semantic stream processing, as recommended by the RSP group [12]. However, a choice still has to be made regarding the used data descriptions. In this regard, the semantic sensor network (SSN) ontology [49] originated as a collaboration between the W3C and the OGC (Open Geospatial Consortium). Its core component is the SOSA ontology for describing sensor features, divided into four major sections: observation, sample, sensor and actuator(hence the name SOSA). The SSN further compliments this with vocabulary to elaborate the sensing systems details. Due to the collaboration, the ontology is largely built upon the standards that were already used by the OGC, namely SensorML [50] and Observations and Measurements(O&M) [51]. As the proposition is currently pending to become an official OGC standard [49] and is recommended by the W3C, it is highly likely to be adopted widely in the IoT sensing layer for semantic applications. Hence it will be the ontology of choice for this project as well.

The SSN and its core SOSA ontology mainly focus on the interaction between devices and systems. However, the ontology does not include a data model for describing measurements themselves in terms of their unit and value. The SOSA ontology includes two different ways of describing measurement results: the simple *hasSimpleResult* predicate, which expects a literal value, and the *hasResult* predicate, needing a resource of the class Result. In order to clearly describe a measurement result in terms of value and unit, it is necessary to use other ontologies.

Using the simple result, the suggested approach is to use the set of custom data types [52]. It leverages the idea of the unified code for units of measure (UCUM [53]), which provides a standard for writing scientific measurement values and its unit similar to the style used when calculating equations on paper. However, this implies that the data is outputted with the unit and its measurement value grouped together in a single string, which seems a bit unusual to do in object oriented programming.

Using *hasResult*, the unit and value can be split as individual properties of the result instance using existing ontologies. The ontology of units of measure and related concepts [54] and quantities, units, dimensions and data types

(QUDT) ontologies [55] are both possible options for doing so. The latter focuses solely on units and the conversions between them, and has a more extensive vocabulary in this regard.

In Fig. 5.1 a sample of the generated data for a single wheel is given, serialized in the Turtle [56] format. One of the main advantages of this serialization format in python's *rdflib* is the automatic generation of namespace prefixes. The usage of relative IRI's using a prefix is both beneficial for the readability of the document, as well as its size when a namespace is used multiple times. The various prefixes provide an overview of the used vocabularies:

- **qudt** : Describing the unit and value of a measurement.
- **sosa** : Linking sensor, observation, feature of interest and its observed property to each other.
- **prov** : An established ontology used to describe provenance of data. In this case, only the *generatedAtTime* predicate is used to identify time in stream data, as recommended by the RSP group.
- **schema.org** : An initiative founded by Google, Microsoft, Yahoo and Yandex that aims to create shared vocabularies for structured data. In this case, the *PropertyValue* pair is used to describe the anomaly score.
- **ssn** : To link feature of interest and observed property.
- **rdf** : The core RDF terms, for defining classes, types and properties. Here also used to define the data generated as a *Bag* container. This way all measurements are at least linked to the generation time of the data graph.
- **rdfs** : Used for annotating the data with human readable information.
- **xsd** : The namespace for XML schema, used to define the data type of literals. Note the definition of the type of the anomaly value and the numeric value of the result are done implicitly. A Turtle parser will interpret the former as an integer and the latter as a double, based on the syntax.

In this example, the base directive causes all non absolute references to be expanded as relatives using the base URL. All identifiers should be unique in order to avoid IRI collision. The domain of example.org is reserved for experimental purposes, but should not be used in real scenarios. In a real life scenario, a public IP-address can be used to avoid IRI collision, or the usage of universally unique identifiers (UUID) [57] can be considered. For readability, the example domain has been used in this sample, using a limited range of integers for the preprocessing and edge processing identifiers. Both processing layers include the option of generating a unique identifier if necessary.

Aggregation and communication

The aggregation of the data in the edge is a given due to its function as a broker. As of this point, further data transmission using the MQTT protocol does not make much sense. The generated output is now a continuous stream. When dealing with continuous streams, it makes sense to use a more connection oriented protocol for the transmission of the data. Hence, the data between the edge layer is transmitted using TCP sockets.

5.4.4 Data Analysis and Application

The goal of this study is to make a proof of concept for the desired architecture in order to assess it's feasibility. Unlike the previous layers, the tasks needed to perform by the analysis and application layer may vary largely depending on the application context. Hence the tasks performed here are minimized to showing the idea of pushing the data through multiple layers of processing, to finally arrive in the desired form at the client. Thus, the analysis and application layer are merged in this case. It processes the incoming enriched data from the edge nodes, and generates human readable notification messages regarding the detected holes. The notifications are made available on a TCP socket as well.

```
@prefix ns1: <http://schema.org/> .
@prefix ns2: <http://www.w3.org/ns/sosa/> .
@prefix ns3: <http://qudt.org/schema/qudt#> .
@prefix ns4: <http://www.w3.org/ns/prov#> .
@prefix ns5: <http://www.w3.org/ns/ssn/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@base <http://example.org/data/>.

<edgeprocessor/45277/data/4> a rdf:Bag ;
    rdf:li <car/15990/measurement/4/value/0> ;
    ns4:generatedAtTime "2018-05-28T14:40:30.214794"^^xsd:datetime .

<car/15990/measurement/4/value/0> a ns2:Observation ;
    rdf:Property [ a ns1:PropertyValue ;
            ns1:name "AnomalyValue" ;
            ns1:value 8.7e-01 ] ;
    ns4:generatedAtTime "2018-05-28T14:41:18.008286"^^xsd:datetime ;
    ns2:hasFeatureOfInterest <car/15990/wheel/0> ;
    ns2:hasResult [ ns3:Unit <http://qudt.org/vocab/unit#RevolutionPerMinute> ;
            ns3:numericValue 2.17e+02 ] ;
    ns2:madeBySensor <car/15990/wheel/0/rpmsensor> ;
    ns2:observedProperty <car/15990/wheel/0/speed> .

<car/15990/wheel/0> a ns2:FeatureOfInterest ;
    rdfs:label "wheel 0 of car 15990" ;
    ns5:hasProperty <car/15990/wheel/0/speed> .

<car/15990/wheel/0/rpmsensor> a ns2:Sensor ;
    rdfs:label "a sensor observing the RPM of a cars wheel" .

<car/15990/wheel/0/speed> a ns2:ObservableProperty ;
    rdfs:label "the rotation speed of wheel 0 of car 15990" .
```

Listing 5.1: A sample of the produced enriched data, serialized in Turtle format.

5.5 The API

In order to adapt to changes in data of interest, its source and the knowledge available about data, an API is needed to manage the system. The used MQTT communication can be reused to manage an entire group of devices based on their location in the layered architecture model. However, it should not be necessary for a client to have insights into the structure of the system in order to manage it. Hence a REST, JSON based API is created to communicate with the various layers of the architectural model.

An advantage of having a single API server available that translates the REST interface calls to MQTT messages for the processing layers, is that it provides a point of consistency inside the system. When a new processing node is started, it requests and loads these settings from the API server.

Defining the data

Internally, the assumption is made that data is inputted into the system in a csv format. This is both a strong and weak assumption, as this is a clear requirement for the data source. Without making a single assumption however, little processing can be performed. Furthermore, the csv format is highly common, thus the assumption is considered reasonable.

Table 5.1: A small extract taken from the dataset, from left to right: message identifier, field length, 8 columns of data values, the timestamp indicator, the identifier for the next message, it's length and 3 more columns of data.

0	8	38	38	38	38	38	0	0	0	0	49.99	246	8	38	38	38
0	8	38	38	38	38	38	0	0	0	0	50.00	246	8	38	38	38
0	8	38	38	38	41	38	0	0	0	0	50.01	246	8	38	38	41
0	8	38	38	38	41	38	0	0	0	0	50.02	246	8	38	38	41
0	8	38	38	38	41	38	0	0	0	0	50.03	246	8	38	38	38
0	8	38	38	38	36	38	0	0	0	0	50.04	246	8	38	38	36

The column is the primary indicator of the type of the data and thus its needed associations. The given sample in Table 5.1 includes 16 out of the 67 columns of the csv dataset. Notice the default length for the CAN messages is 8 bytes, but often not all values are used.

Settings that can be made affect how the processors process these columns. Necessary features regarding the data include marking a column for anomaly processing and deciding if its anomaly score should be outputted, linking the column to possible timestamps in another column, updating the annotations associated with a column, etc.

From a user point of view, the majority of these are straightforward and require no expertise. The exception is found in the annotations that need to be added to the data that are application specific. This is a byproduct of the underlying data model. An RDF statement requires a predicate, object and subject, and each of them needs to be identified. An API client may provide abstractions for more well known predicates and subjects, but the necessary application specific context cannot be predicted, so these ontologies and objects will need to be provided to the system manually. In this system, *profiles* are used, which automatically set a number of necessary associations when elected. By using these, a balance can be found between automatic and manual annotation of the data. As an example, the extract in Listing 5.1 can be regenerated setting the profile 'wheel_speed' on a given column, which automatically sets the annotations.

Some more problems regarding the identifiers are tackled as well. In a streaming context, some identifiers will need to be incremented, while others do not need to be. For example each unique measurement made needs its own unique identifier, but the sensor that made them is consistently the same, so it does not. To tackle this problem, the identifiers inputted by users may contain regex-like syntax to indicate the need for dynamic identifiers.

5.6 Implementation Language

The project was entirely developed using python. The main reason for this choice is the need for rapid development, as the project had to be fully completed within a short time frame.

Chapter 6

Experimental study

In this chapter, the goal is to assess the performance of the architecture model proposed in chapter 5. This will be done by performing measurements on the described implementation, and extrapolating these towards a more general scenario. The analysis layer can reasonably be situated in a Cloud or clustering environment. Because of this, there is no need for major concerns regarding the needed resources at this level, as these can be increased easily on demand. Furthermore, in this implementation the analysis layer is a simple application. Hence, the focus will mostly be on the edge and preprocessing layers.

6.1 The experimental design

Computational overhead

At the preprocessing level, both the **computational overhead** of the algorithm as its accuracy have to be observed. The computational overhead is observed using the processing time, which shall depend on the chosen algorithm and the associated window size. In the experimentation context, the window size shall always refer to the amount of points included in it. The accuracy of the algorithm is determined by analyzing its false negative and positive rate. It is of the utmost importance that an algorithm can be found that yields both acceptable results and a reasonable computational complexity, as in many IoT contexts the computational resources available at this level will be minimal. An algorithm with too high complexity creates an unfeasible solution, but an overly inaccurate algorithm is unacceptable as valuable data could be lost.

Data generation rate

At the edge computing level, the influence of the data enrichment process has to be measured. The main concern here consists of the **generated data rates**, as well as the computation time needed to perform the data enrichment process. The data rate is a key issue in any stream-based computing. Indeed, as data arrives in a continuous mode, the system's throughput is conditioned by data rate in input. The larger the data rate, more stressful the computing process will be. In other words, studying the variability of the data rate in input should shed light on the computing requirement of the system able to accommodate the data generation rate. Accordingly, the system should be able to use a parallel mode of processing to accommodate higher data generation streams, which for instance in our context, would result if more cars are considered as contributors to the data stream generation.

Scalability

Additionally, the **scalability** of the system will be addressed. The main goal is to assess if an edge node can serve a reasonable amount of preprocessors in the enrichment process. While the edge layer does not have similar resource constraints as the preprocessing layer, it does not boast the on demand resources that exist in a Cloud environment. An occasional need to deploy additional edge nodes in a growing region may be acceptable, but the edge layer should not become a data center.

The methodology

The methodology of the experimental study is to first evaluate the above mentioned performance indicators separately at each layer and for its modules and then provide performance results for the system as a whole, where modules are chained from IoT layer to application layer. In this case, we would be interested to see the **response time** from reception of events in data stream to a final alert to users about events of interest. It should be noted that the response time will serve as a measure of QoS of the system and would therefore be tuned (by adding more computing resources through horizontal or vertical scaling) to match a desired, *a priory* QoS level.

6.2 Testing environment

The ideal testing environment requires a full deployment of the envisioned layered architecture model. This would require multiple microprocessors, edge devices, etc. Due to the time limitations for this study, all tests will be performed running the various components of the architecture model on a single device, as separate processes. For completeness, the most important processing specifications of the testing device are given:[1]

- Processor: Intel® Core™ i5-4210H, clock speed 2.90GHz up to 3.50GHz, dual core, 3MB cache, maximum of 4 threads active.
- Memory: 8GB DDR3 RAM at 1600MHz.
- Storage: Seagate ST1000LM014 SSHD (hybrid). 5400rpm, 8GB SSD cache.

Obviously, using a single computer simplifies the computing environment and various assumptions on the model. This also implies that any communication between the different layers will be performed using the internal network of the computer. Delays generated by network traffic can not be measured. Nevertheless, separately the performance of every layer in our model can be studied as well as the full system performance. The aim is therefore to show the feasibility of the stream processing.

6.3 Performance and accuracy of the preprocessing

6.3.1 Considerations regarding the dataset

As this section will assess the performance of the implemented algorithms, it is necessary to consider the dataset used in the process. More specifically, it is important to realize the effects of using simulation data that is both realistic, but still incomplete as it is an ongoing research process. The following considerations were realized at a later stage, but heavily impact the results of the system.

[1]Due to the age of the system and it having a number of specific tweaks, a web page with the complete overview cannot be provided.

Quantization The output of the sensor is a byte, a digital value transmitted on the CAN bus. This implies that an analog to digital conversion has already happened, and the amount of quantization values is restricted to a range of 255. Due to this, the generated output of the sensor can be considered to be a severely rounded version of the actual analog value, which is unavailable. The result of this is a severe distinction between the detectable holes. A small hole's influence on the analog value simply gets filtered by quantization, making it invisible in the digital data. Hence, only bigger holes actually make their way through the quantization process. The quantization, combined with the influence of noise on the measurements, is also what causes the fluctuations over time that are visible when the car speeds up or slows down (as could previously be seen in Fig 5.4).

Behavior In the dataset, the car is both speeding up and slowing down over time. Other than this, there is the presence of holes while driving. In reality, this actually already contains the majority of relevant events for the experiment. There's a number of distinct events that would normally be hard to discern from a hole that are not yet included however. The main consideration can be for example bumps of any kind, e.g. hitting a speed bump or having to cross the sidewalk to enter the garage.

6.3.2 Result of the algorithm using differences

The algorithm using differences greatly benefits from the quantization previously described. Due to the data being discretized, an absolute threshold can easily be determined for which the transitions in speeds are not measured, but the holes are. The accuracy of the algorithm can be defined as follows. From the observations, it is known that a hole causes fluctuations in the data for a time of 0.15 seconds or 15 measurements. The labels in the dataset indicate a flag for the occurrence of a hole at the start of this period. Hence a hole is correctly detected, if a detection occurs and such a flag can be found in the last 15 measurements. Using these settings, and setting the minimum difference between consecutive points to 2 causes all holes in the set to be detected, with no false positives or negatives. The resulting accuracy is thus 100%.

This accuracy however has to be seen in the context of the considerations made, especially the behavioral. This algorithm will by no means be able to discern between a speed bump and a hole in a real scenario, but could possibly serve as a good preliminary filtering for possible data of interest.

As this only involves a single subtraction and the comparison to the threshold value, execution time is in a range of O(1).

6.3.3 The simple statistical approach

When considering this approach, the assumption was made that a good balancing point could be found where the threshold for the amount of standard deviations would clearly divide between the spikes of the holes in the set, and the fluctuations generated by the transitions in speed. Testing however, quickly indicated that such an assumption is not to be made lightly. Using identical definitions for the accuracy of the detector as the previous experiment, a first set of measurements is made to observe the influence of a changing window size, i.e. the number of points used in the calculation, at a lower threshold level. In these results, the amount of false positives is immense. In order to be able to display both the amount of false positives and the amount of correct detections, the y-axis scale is logarithmic.

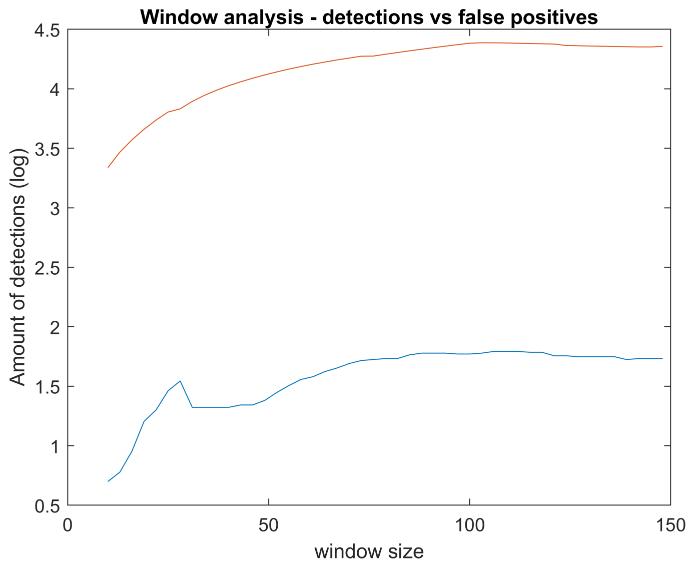


Figure 6.1: Window size (amount of points used in the calculation), along with the number of correct and incorrect detections. Axis on a logarithmic scale due to the high amount of false positives(red) to correct(blue) ratio.

Furthermore, even at a low threshold the algorithm fails to detect all holes. However, a point of interest can be observed around a window size of 28, where the graph suddenly drops. Hence for this value, the threshold was gradually increased in an attempt to lower this false positive ratio. Although the increasing threshold did improve the amount of false positives, more of the correct measurements are lost as well. This reached a critical value for a threshold of 0.85, for which any further increase of the threshold caused all false positives to be eliminated, but only 5 out of 86 holes are actually detected. At this critical threshold itself however, only 30 of the 86 holes are detected, with over 30 times as much false positives. The window size of 97 yielded the highest amount of detections, so the same test was also run for this size. Although not having a similar critical point, this test also showed horrible results regarding accuracy. Due to the long running time of these experiments and the clear indications that this algorithm is unsuited for this problem, measurements regarding accuracy were concluded here.

The processing time to process the whole dataset, which implies very large number (nearly 50000) of loops of the calculation process was also logged. The processing time needed is expected to be linear to the window size, as is observed from the processing times.

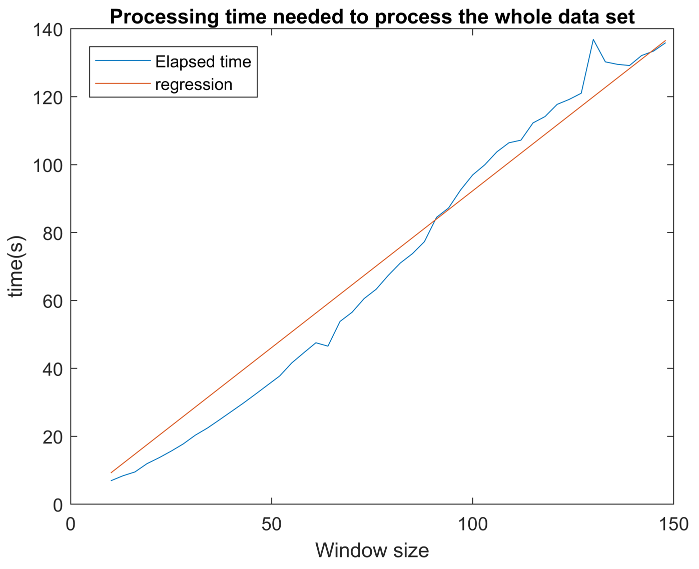


Figure 6.2: The elapsed time to process the full dataset, corresponding to the amount of points included in the calculation.

6.4 Performance and data rates in the edge layer

6.4.1 Effect of output format

The data enrichment process can roughly be divided in two steps. First, the graph is built internally adding the triples from the list of associations defined by the user. After this the graph is parsed into a serialized format ready for transmission. In the python libraries used in this project, the available options are: xml, n3, turtle, n-triples, pretty-xml, trig and json-ld. While each have their own syntax, the following should be noted:

- The formats n3, trig and turtle are closely related. Turtle is compatible with n3, and trig is an extension of turtle.
- The key difference between xml and pretty-xml is the use of some shorthand notations introduced in RDF1.1 [6]. Pretty-xml is thus more compact and readable, as its name indicates.
- As mentioned in the data enrichment section 5.4.3, one of the key advantages of the turtle family languages in the python library is the automatic generation of prefixes. JSON-LD has the equivalent of contexts for this, but the automatic generation of these contexts is not available.

For this test, the computational time needed for the full data enrichment is measured. In this case, the edge processors broker was flooded with the data at a high rate (500Hz). This causes the broker to act as a buffer, while processing is running at full capacity. The annotations set used contain the full expansion of the data for a wheel as seen in Fig. 5.1, for all four wheels. This setup is representative for the desired application. The average of the computation time is computed for a total of 2,000 expansions. The results of this measurement are given in Fig. 6.3. Full histograms for each individual measurement can be found in appendix A.1. Note the final value of the x-axis indicates the maximum observed value, but its occurrence rate causes the bin to be too small to be visible.

The performance time of the `nt` serialization is significantly better than that of the other formats. There's also a significant difference between the

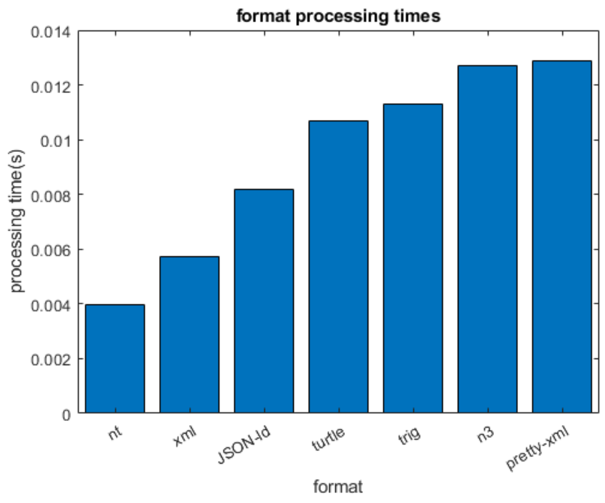


Figure 6.3: Performance of various formats for RDF serialization.

regular xml serialization and the pretty version. In order to put these processing times in perspective, there should be some considerations regarding the generated output. The nt format may be the fastest in terms of processing time, but it is close to raw outputting of the triples inserted into the graph. This affects the readability, but more importantly the size of the generated data. The sizes of the generated enriched data for each format were thus also measured. A chart similar to the processing times can be made for the file sizes.

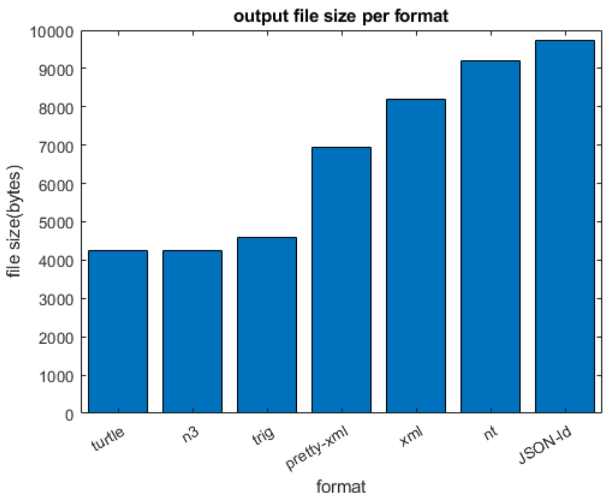


Figure 6.4: File size for various formats of RDF serialization.

There is a clear price to be paid for the processing speed gained using the nt format. The generated output files are more than twice the size of the turtle format. As the turtle format provides the best trade-off between performance and size for the application context, it will be the format of choice in later experiments.

As a last note to this small comparison between the serialization formats, it should be noted that the JSON-LD serialization is both favored and punished by the lack of the ability to generate the contexts. This lowered its processing time in the first experiment, but results in the observed explosion of the data.

6.4.2 Output data rate

To determine the expected output data rate, a calculation can be made. None of the preprocessors will output continuous data rate, as data is only outputted when an anomaly is detected. However, it can be assumed that for each output format, a balance point can be approximately reached where the buffer is filled at the same rate as it is processed. The processing time needed at this point is equal to that of the previous experiment in this scenario, as the processor is running at full capacity. Using the data obtained by the previous experiment, the data rate r can thus be determined as the product of the processing time t and the file size s.

$$r = 1/t * s \tag{6.1}$$

Using Eq. (6.1), the maximum, minimum and average expected data rates are found. Times are given in seconds, rates in bytes per second (see Table 6.1). For visual representations of the data, see Figs 6.5 and 6.6.

Table 6.1: The expected data rates at full running capacity for each format.

Format	min time	max time	avg time	min rate	max rate	avg rate
n3	0.0109	0.0536	0.0127	79,340	388,260	334,220
Turtle	0.0100	0.0555	0.0107	76,497	424,900	398,090
trig	0.0099	0.0555	0.0113	82,633	460,750	405,060
pretty-xml	0.0114	0.0551	0.0129	125,559	604,517	537,916
json-ld	0.0077	0.0232	0.0081	419,686	1,263,778	1,119,001
xml	0.0054	0.035	0.0057	235,818	1,511,316	1,427,567
nt	0.0036	0.0382	0.0039	240,281	2,492,199	2,312,840

The highest data rate is found for nt, which boasts the highest processing speed, while also having a reasonably big file size (as seen before in Figs 6.3 and 6.4). Yet there is still no need for major concerns of the network layer becoming a bottleneck, as a worst case speed requirement of approximately 2.5MB/s is still acceptable. There are two key side notes to make to this observation:

- Due to the implementation choice of using python, multithreaded processes are virtualized. Actual parallel processing can only be obtained by using multiprocessing. Such an implementation has been made, but the high amount of shared variables between the enrichment processes causes the performance to drop to a point where it is worse than the virtualized multithreading. Hence only a single core is actually leveraged. The amount of data generated could thus be a number of factors higher depending on the number of available cores in the edge system.

- This performance table is an analysis for the desired application system. The influence of the number of statements that need to be expanded on the output file size and processing time has to be observed.

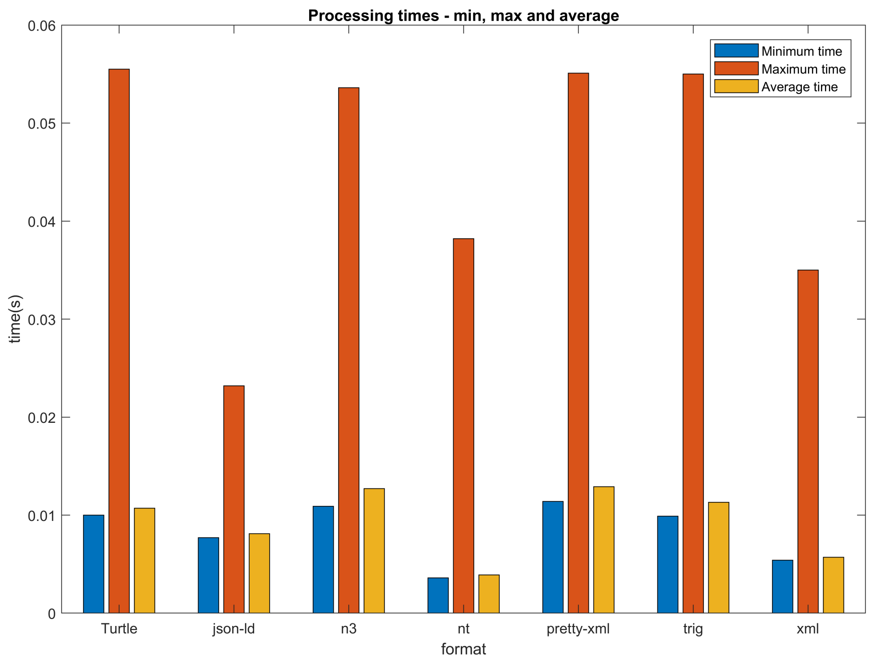


Figure 6.5: Minimum, maximum and average processing time per RDF serialization format.

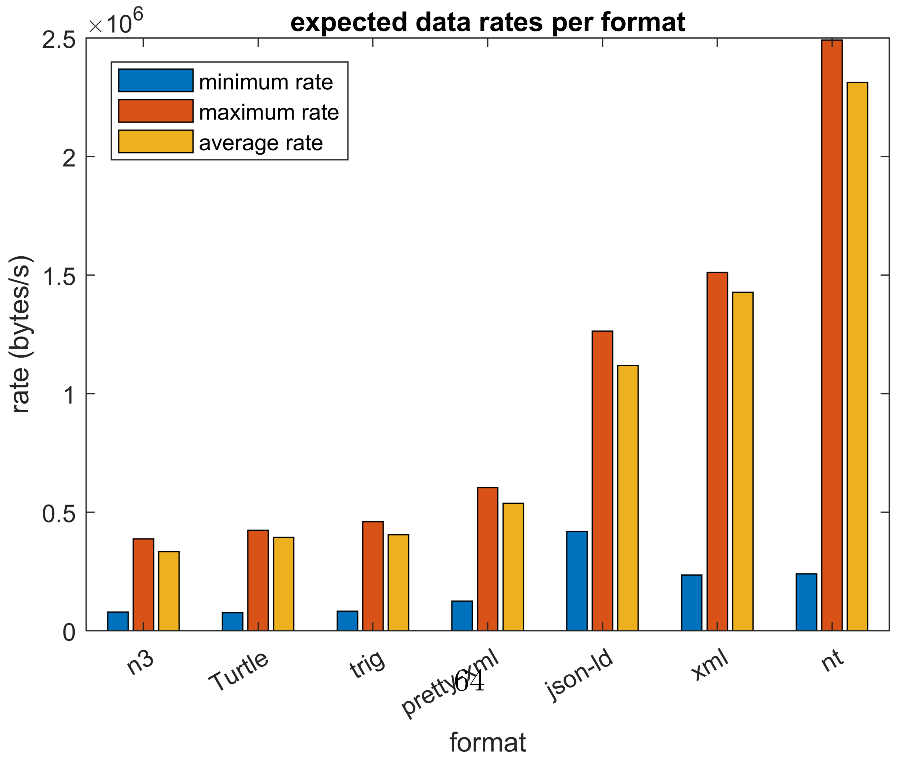


Figure 6.6: Resulting data rates for each format.

6.4.3 Effects of the amount of statements

The time needed to perform the enrichment process is related to the amount of triples that need to be added to the data. In the previous sections the focus was on the performance time and data rate given the context application. The amount of statements needed in this case equals 82. In this section, the effects of increasing amounts of statements on the processing time is observed. Observing the impact on the file size is more difficult, as the size of the generated output is related to the length of the identifiers used in the graph. Furthermore when identifiers reoccur, the relative expansion using prefixes can be used, which lowers the output size. The worst case scenario for a given triple increases the file size by the sum of the lengths of the identifiers in the triple and the additional overhead generated by the output format.

For this test, an increasing amount of statements is added to a single column. For each amount of statements, the average of the computation time is calculated 30 times. The reason the amount of expansions is lower for this experiment mainly involves the running time. In this test, the amount of statements is increased 200 times starting from 3 with a step of exactly one additional statement. The result for turtle is displayed in Fig. 6.7.

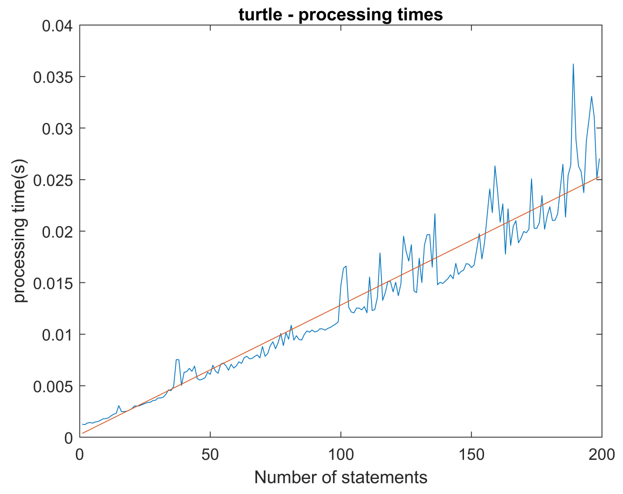


Figure 6.7: The processing time as a function of the number of statements using turtle serialization.

A full overview of the resulting measurements can be found in appendix A.2.

As the amount of triples increases, the pattern of the resulting means becomes more and more unstable. The linear trend remains visible, but the regression is clearly affected by the influence of the spikes. These spikes can be related to outliers that occur during the measurement, which appear to be occurring more often for higher numbers of triples. This can also be seen if the standard deviation of the measurements is plotted along the graph of the means Fig. 6.8. Further increasing of the amount of measurements for each amount of statement is thus likely to smoothen the graph, but would increase the running time of the experiment to overly high levels.

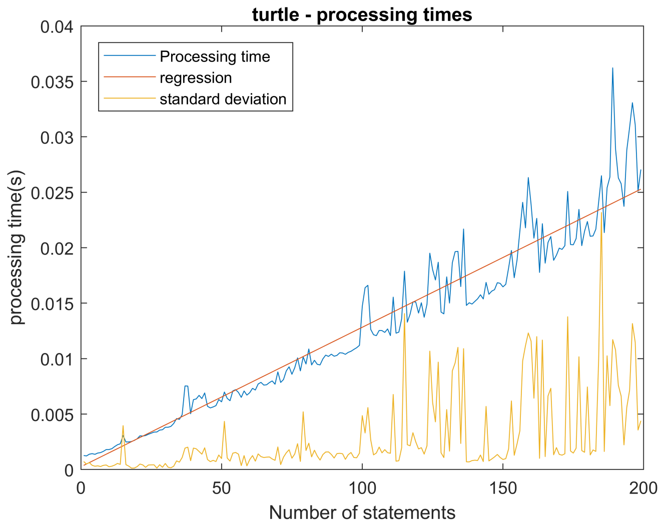


Figure 6.8: The same measurement result as Fig. 6.7, with the standard deviations plotted. The spikes in the mean and the standard deviation show clear correlations, indicating the influence of outliers.

Regardless, it can be observed that the processing time increases in a linear fashion in regard to the amount of triples used for reasonable amounts of triples.

6.5 The system response time

The response time of the system can be theoretically determined as the sum of:

- The processing time needed at each level.
- Time lost to access the network I/O interface.
- The time lost by the network latency.
- Latency incurred by the algorithm.

Since the testing is done internally, the latter can not be observed. Latencies introduced by algorithms exist for all algorithms that need later measurement samples in order to calculate the result for the current. It can easily be determined by the amount of later samples needed and the sample frequency. The time lost to obtain data from the I/O buffer, however is harder to measure. In the previous tests the focus was solely on the processing time caused by the algorithm itself, excluding time elapsed to read the I/O buffer. In order to get an accurate overview of the total response time, the time elapsed between the availability of the data to the system and it's arrival at the client is measured. This is done by adding the original generation time as the time stamp of the data that is used in the stream processing. This test uses the application context, with turtle serialization format and anomaly detection.

Difference-based algorithm

Using the difference based algorithm the result yielded an average of 0.0287s to obtain the alert message at the client. As with previous measurements, the distribution of the results is right skewed. The obtained results over a total of 3000 measurements are given in Fig. 6.9.
Naturally, the results of this algorithm can be assumed to be a best case scenario as it has no inherent latencies and a minimal computational overhead. Hence, the previously implemented statistical algorithm's running time is also observed as a more reasonable reference, despite it's inaccuracy.

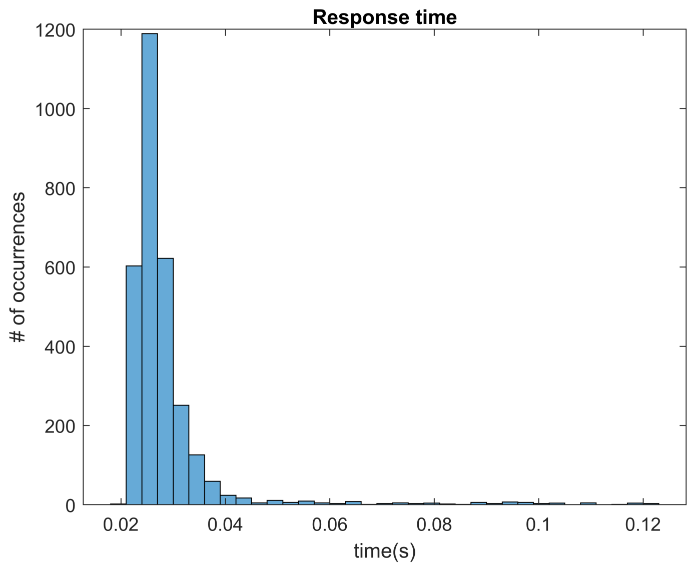
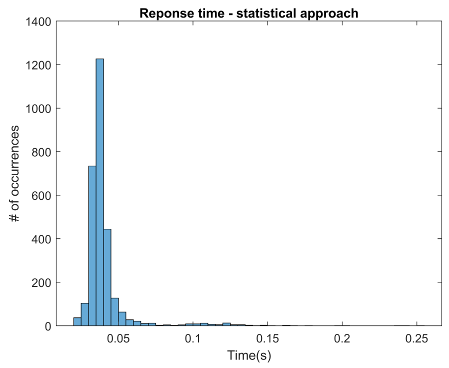


Figure 6.9: The system response times when using the difference algorithm. On average a message reaches the user within 0.0287s, network latencies excluded.

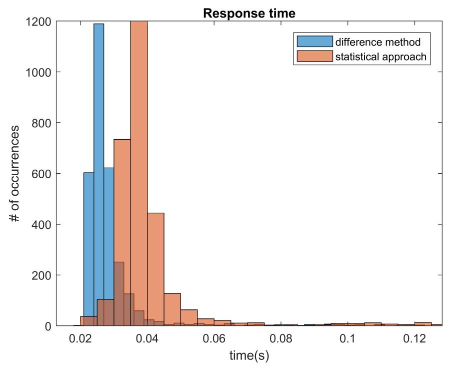
Statistical approach

The running time of the statistical approach has a natural latency of half the window size due to the centering applied. In a problem where the dataset actually following a statistical distribution, this would be an unnecessary step. As a point of reference, the previous experiment was repeated using a window size of 50 points. The average response time for this observation was 0.0406s (excluding the additional latency introduced by the centering of the window).
The influence of the additional processing time needed is clearly visible (Fig 6.10). This influence can be expected to be higher in the case of a preprocessing device, which can normally has less computation power. Having a worst case scenario of a processing time in a range of 0.3s seems acceptable for this application.

As previously described the delay generated by the need for further samples can increase this further. If this increase was considered in this experi-



(a) Response time of the statistical approach using a window of 50 points.



(b) The response time of both methods compared.

Figure 6.10: Comparison of response times

ment, the resulting delay t could thus have been calculated by Eq. (6.2).

$$t = W/2 * f \tag{6.2}$$

With W being the window size and f the frequency. In this case, the additional latency is equal to 0.25s. The consequences of the need for later samples is more concerning in environments with low frequencies.

6.6 Scalability

In order to further understand the behavior of the system and its scalability, the response time is observed at varying circumstances. In order to simulate these, a set number of 20 preprocessors is setup to produce data at a consistent frequency. A test is run increasing the generation rate of each of the preprocessors in steps of 1/20 Hz for all preprocessors. Thus, the total generation rate of the data is increased in steps of 1Hz. For each rate, the response time of 400 messages is measured. A visual representation of the means for each generation frequency is given in Fig. 6.11.

The measurements were ended at these frequencies, because it can be observed that in fact the system has already reached its saturation point. Comparing the individual measurements reveals a distinct difference between the result that is obtained at a generation rate of 53Hz and that of 54Hz. At the former rate, the response time fluctuates a lot, but appears to be

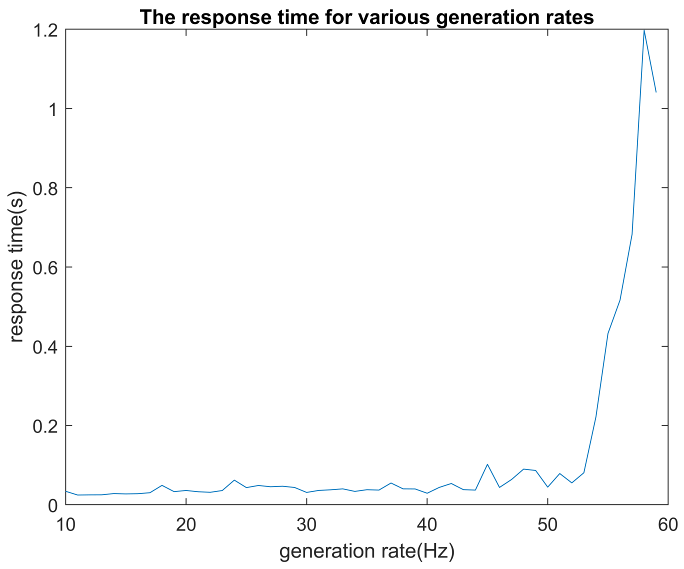
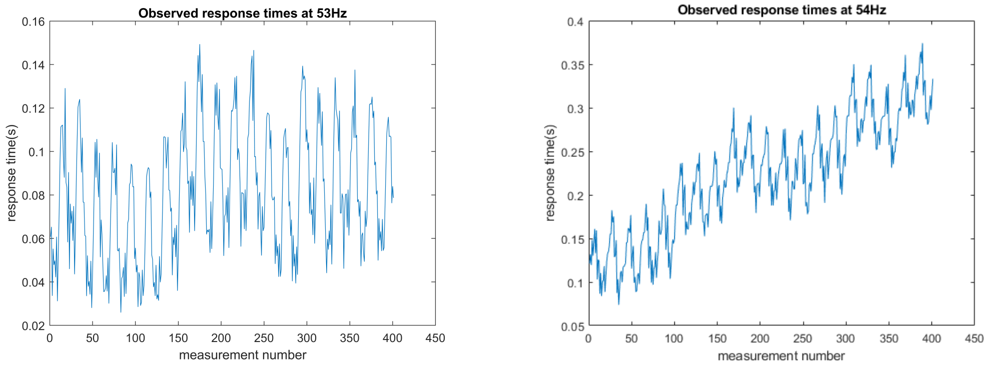


Figure 6.11: The response time for increasing data rates using multiple clients. For higher frequencies, the response time clearly increases.

somewhat stable. The latter however, displays a clear increasing trend as the time goes on.



(a) The observed response times at 53Hz.

(b) The observed response times at 54Hz.

Figure 6.12: Comparison between individual measurements at the critical generation rates.

Another interesting observation can be made regarding the measurement itself. Recalling the histograms of the measurements performed using only a single processor Fig. 6.10, the distribution displays a clear peak of a response time that reoccurs far more than the others. Such a thing is not apparent in the time graph in Fig. 6.12a. Furthermore, the mean of the observed value appears higher than the observed value in the experiment as well. What causes this is the buffering of the messages in the MQTT broker. In the previous experiment, a single processor was running and the time difference between two samples is never smaller than the processing time needed by the edge processor. However, this is not the case when running a number of preprocessors simultaneously. If two of the processors happen to be transmitting messages at times very close to each other, one of the messages will get buffered until the edge processor has finished processing the first. In this experiment, the preprocessors were started one at a time at random time intervals from each other, so there is a reasonable chance this occurs. This creates the highly unstable pattern in the response time and the increased average.

One of the big questions remains: how many cars corresponds to the saturation frequency? This question is hard to answer and could span an entire study by itself. To answer this question one would have to know the generation rate of events for a real car. If the used algorithm is considered to be the difference based approach, it'll detect speed bumps and other things as well. So the question can be reformulated as: how much % of the time is a car driving flat road? If the occurrence rate of the set is considered representative, it is about 0.17%. In this case, an edge node could possibly serve up to a total of approximately 29000 cars! While this may seem like an awfully low percentage, at a measurement frequency of 100Hz it means a car is likely to introduce an event approximately every 6 seconds. While this does not seem overly unreasonable, a careless statement on this cannot be made without further information.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis the problem of determining tasks that can be moved away from the classic Cloud processing and pushed towards the edges of the Internet in a semantic, IoT stream processing context was studied. In this case, the option of performing early event detection and data enrichment was considered.

The problem is important from an architectural point of view, because the simple solution of pushing all data straight towards the Cloud may not be sustainable anymore in the foreseeable future. Furthermore, the available processing power both in the edge and sensing layer of the IoT keeps growing, yet remains unused. From an application point of view, it is even more important to perform such early filtering of the data, because of the possibility of a bottleneck occurring at the network level and the possible cost incurred by transmitting a high data rate source over a large distance.

To answer this, the layered architecture model consisting of a total of five layers (IoT layer –the lowest layer–, three intermediate key processing layers and a final application layer) was created. The tasks identified to push towards the edges were the anomaly detection, the filtering of multivariate data for its relevant components and the enrichment of the data. The anomaly detection was pushed the closest to the source, with the reasoning being that it should be avoided as much as possible to process data that will eventually end up getting discarded. Furthermore, if the enrichment were to be performed at a more early level, the generated data rates would be immense.

The created architecture was validated using simulated data. This choice is mainly made because actual deployment is not straightforward as it would require cars equipped preprocessing devices. First an assessment is made regarding the accuracy of the algorithms used in the preprocessing layer and their processing time. Both needed to be capable of being at an acceptable level in order to possibly run in the preprocessing layer in a real scenario. Next, the processing time needed to perform the enrichment process was observed for various output formats. Finally, the response time was used as a metric to describe the quality of service and scalability of the system.

The reasoning and accuracy of detecting the event of interest depends on the specific environment of the desired application, as well as the used algorithm. Due to the high amount of different approaches for detecting anomalies, some insight in the behavior of the data and knowledge regarding the algorithm is needed. If the behavior of the data is well known, simple algorithms can often be implemented efficiently to detect anomalous events. For example, using the statistical approach for data that is know to occur following a certain distribution, or a regression based approach if the data is know to have a certain trend. If the computational power available allows it, more complex algorithms such as the HTM (Hierarchical Temporal Memory) algorithm can be considered.

The results of the study show the feasibility of the proposed architecture, that is, the computing tasks for processing an IoT data stream can be distributed along layers of the architecture. This way the Cloud is offloaded, which enables a faster and more reliable response time.

In conclusion, it is believed that the thesis has achieved the initially proposed objectives by presenting a full cycle study from problem definition, specification and requirements to final testing and validation. It can be stated that the proof of concept architecture fulfills the main requirements and objectives that had been set.

7.2 Future work

The created implementation is merely a proof of concept and can be improved in various ways. The most principle work that must be done first is the full deployment of the architecture in a real scenario. This was not possible within the time limits of the project as well as lack of infrastructure to support all layers (certainly the RDLab of Computer Science[1] can support upper layers of reasoning and application layers, the preprocessing layer remains an issue as mentioned earlier), but would allow for a better assessment of the architecture model. Specifically, the available preprocessing power and the network latencies in the system can be observed and data rate processing can be more realistically evaluated. Because the architecture is application independent, various applications sharing the output can be developed as well, given that they fit in the context of less resource constrained IoT processing.

In particular, a mobile application for drivers, that could be used to warn drivers about potholes in their region, save car from damages, avoid potential accidents, etc. would be interesting. In a much further future work that is far beyond the reach of this project, the output of processing the data stream could be included in the traffic info of GPS systems. Such additional information would be most beneficial to avoid accidents caused by these holes.

Finally, another aspect of interest would be to make the enriched data persistent for more deep analysis. Specifically, a historical analysis of the data would be interesting. This is something that cannot be performed online, and is best done in batch operations or offline. In time, historical analysis may be able to generate information regarding where potholes are likely to be found, before detection. A reanalysis of the data of the system may also be applied using more complex algorithms, in order to obtain more accurate information regarding the the detected holes.

[1]RDLab: https://rdlab.cs.upc.edu/

Acronyms

API Application Programming Interface.

C-SPARQL Continuous-SPARQL.

CAN bus Controller Area Network: a robust vehicle bus standard designed to allow micro-controllers and devices to communicate with each other in applications without a host computer.

CEP Complex Event Processing.

CQELS Continuous Query Evaluation over Linked Stream.

CQELS-QL CQELS Query Language.

CSV Comma Separated Value.

DSMS Data Stream Management System.

ECA Event Condition Action.

EP-SPARQL Event-Processing SPARQL.

HTM Hierarchical Temporal Memory.

IoT Internet of Things.

IRI International Resource Identifier.

MQTT Message Queue Telemetry Transport.

OGC Open Geospatial Consortium.

OWL Web Ontology Language.

Pothole A depression or hollow in a road surface caused by wear or subsidence.

RDF Resource Description Framework.

RDF-S Resource Description Framework Schema.

RIF Rule Interchange Format.

RSP RDF Stream Processing.

SOSA Sensor, Observation, Sample and Actuator.

SPARQL Spark Processing And RDF query Language.

SSN Semantic Sensor Network.

W3C World Wide Web Consortium.

Bibliography

[1] L. Atzori, A. Iera, and G. Morabito, “Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm,” *Ad Hoc Networks*, vol. 56, pp. 122 – 140, 2017, http://www.sciencedirect.com/science/article/pii/S1570870516303316.

[2] Statista. Last visited on 18-06-2018. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

[3] P. Ritrovato, F. Xhafa, and A. Giordano, “Edge and cluster computing as enabling infrastructure for internet of medical things,” in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA-2018).* IEEE 2018, 2018, pp. 717–723.

[4] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, “A survey on the edge computing for the internet of things,” *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[5] S. Bratt. (2007) Semantic web, and other technologies to watch. W3C. Last visited on 18-06-2018. [Online]. Available: https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)

[6] Y. Raimond and G. Schreiber, “RDF 1.1 primer,” W3C, W3C Note, Jun. 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[7] D. Brickley and R. Guha, “RDF schema 1.1,” W3C, W3C Recommendation, Feb. 2014, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[8] B. Parsia, P. Hitzler, P. Patel-Schneider, M. Krötzsch, and S. Rudolph, "OWL 2 web ontology language primer (second edition)," W3C, Tech. Rep., Dec. 2012, http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[9] B. Parsia, B. Motik, and P. Patel-Schneider, "OWL 2 web ontology language structural specification and functional-style syntax (second edition)," W3C, W3C Recommendation, Dec. 2012, http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[10] H. Boley and M. Kifer, "RIF overview (second edition)," W3C, W3C Note, Feb. 2013, http://www.w3.org/TR/2013/NOTE-rif-overview-20130205/.

[11] L. Farhan, S. T. Shukur, A. E. Alissa, M. Alrweg, U. Raza, and R. Kharel, "A survey on the challenges and opportunities of the internet of things (iot)," in *2017 Eleventh International Conference on Sensing Technology (ICST)*, Dec 2017, pp. 1–5.

[12] D. Anicic *et al.* (2016) RDF stream processing: requirements and design principles. RDF Stream Processing Community Group. Last visited on 18-06-2018. [Online]. Available: http://streamreasoning.github.io/RSP-QL/RSP_Requirements_Design_Document/

[13] A. Mauri, J.-P. Calbimonte, D. Dell'Aglio, M. Balduini, M. Brambilla, E. Della Valle, and K. Aberer, "Triplewave: Spreading rdf streams on the web," in *The Semantic Web – ISWC 2016*, P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, Eds. Cham: Springer International Publishing, 2016, pp. 140–149.

[14] S. Sundara, S. Das, and R. Cyganiak, "R2RML: RDB to RDF mapping language," W3C, W3C Recommendation, Sep. 2012, http://www.w3.org/TR/2012/REC-r2rml-20120927/.

[15] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[16] S. Helmer, A. Poulovassilis, and F. Xhafa, *Reasoning in event-based distributed systems.* Springer, 2011, vol. 347, pp. 51–64.

[17] D. F. Barbieri, D. Braga, S. Ceri, E. D. Valle, and M. Grossniklaus, "Querying rdf streams with c-sparql," *ACM SIGMOD Record*, vol. 39, no. 1, pp. 20–26, 2010.

[18] A. Margara, J. Urbani, F. van Harmelen, and H. Bal, "Streaming the web: Reasoning over dynamic data," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 25, pp. 24 – 44, 2014, http://www.sciencedirect.com/science/article/pii/S1570826814000067.

[19] J.-P. Calbimonte, O. Corcho, and A. J. G. Gray, "Enabling ontology-based access to streaming data sources," in *The Semantic Web – ISWC 2010*, P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, and B. Glimm, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 96–111.

[20] J.-P. Calbimonte, H. Y. Jeung, O. Corcho, and K. Aberer, "Enabling query technologies for the semantic sensor web," *International Journal on Semantic Web and Information Systems*, vol. 8, no. 1, pp. 21. 43–63, 2012.

[21] RDF Stream processing community group. Last visited on 18-06-2018. [Online]. Available: https://www.w3.org/community/rsp/wiki/RDF_Stream_Processors_Implementation

[22] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth, "A native and adaptive approach for unified processing of linked streams and linked data," in *International Semantic Web Conference.* Springer, 2011, pp. 370–388.

[23] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic, "Ep-sparql: A unified language for event processing and stream reasoning," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: ACM, 2011, pp. 635–644, "http://doi.acm.org/10.1145/1963405.1963495".

[24] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, "Stream reasoning and complex event processing in etalis," *Semantic Web*, vol. 3, no. 4, pp. 397–407, 2012.

[25] J. Chu, H. Fu, F. Gao, and D. Zhao, "Towards complex event processing in linked data stream," in *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, June 2017, pp. 1016–1021.

[26] M. Dao-Tran and D. Le Phuoc, "Towards enriching cqels with complex event processing and path navigation." in *HiDeSt@ KI*, 2015, pp. 2–14.

[27] D. Lephuoc. Official java implementation of the cqels excecution framework. Last visited on 18-06-2018. [Online]. Available: https://github.com/cqels/CQELS4J

[28] J. Hoeksema and S. Kotoulas, "High-performance distributed stream reasoning using s4," in *Ordring Workshop at ISWC*, 2011.

[29] S. Bechhofer. Owl reasoning examples. Last visited on 18-06-2018. [Online]. Available: http://owl.man.ac.uk/2003/why/latest

[30] C. Grosan and A. Abraham, *Rule-Based Expert Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 149–185, https://doi.org/10.1007/978-3-642-21004-4_7.

[31] Official drools website. Red Hat Inc. Last visited on 18-06-2018. [Online]. Available: https://www.drools.org/

[32] Drools fusion documentation. Red Hat Inc. Last visited on 18-06-2018. [Online]. Available: https://docs.jboss.org/drools/release/6.2.0.CR3/drools-docs/html/DroolsComplexEventProcessingChapter.html

[33] Apache jena. The Apache Software Foundation. Last visited on 18-06-2018. [Online]. Available: https://jena.apache.org/

[34] A. Fobel and N. Subramanian, "Comparison of the performance of drools and jena rule-based systems for event processing on the semantic web," in *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*, June 2016, pp. 24–30.

[35] M. B. Hassine. Easy-rules github page. Last visited on 18-06-2018. [Online]. Available: https://github.com/j-easy/easy-rules

[36] D. McCarthy and U. Dayal, "The architecture of an active database management system," *SIGMOD Rec.*, vol. 18, no. 2, pp. 215–224, Jun. 1989, http://doi.acm.org/10.1145/66926.66946.

[37] R. Adaikkalavan and S. Chakravarthy, *Generalization of Events and Rules to Support Advanced Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 173–193, https://doi.org/10.1007/978-3-642-19724-6_8.

[38] S. Hu, M. Huang, W. Feng, and Y. Zhang, "A smart health service model for elders based on eca-s rules," in *2017 IEEE 15th International Conference on Software Engineering Research, Management and Applications (SERA)*, June 2017, pp. 93–97.

[39] H. M. C. Chandrathilake, H. T. S. Hewawitharana, R. S. Jayawardana, A. D. D. Viduranga, H. M. N. D. Bandara, S. Marru, and S. Perera, "Reducing computational time of closed-loop weather monitoring: A complex event processing and machine learning based approach," in *2016 Moratuwa Engineering Research Conference (MERCon)*, April 2016, pp. 78–83.

[40] T. Huybrechts, Y. Vanommeslaeghe, D. Blontrock, G. Van Barel, and P. Hellinckx, "Automatic reverse engineering of can bus data using machine learning techniques," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, F. Xhafa, S. Caballé, and L. Barolli, Eds. Cham: Springer International Publishing, 2018, pp. 751–761.

[41] R. Gass and C. Diot, "An experimental performance comparison of 3g and wi-fi," in *Passive and Active Measurement*, A. Krishnamurthy and B. Plattner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 71–80.

[42] J. Budomo, I. Ahmad, D. Habibi, and E. Dines, "4g lte-a systems at vehicular speeds: Performance evaluation," in *2017 International Conference on Information Networking (ICOIN)*, Jan 2017, pp. 321–326.

[43] S. Corrigan, "Introduction to the controller area network(CAN)(rev. b)," http://www.ti.com/lit/an/sloa101b/sloa101b.pdf, Texas Instruments, Incorporated, Jun 2016.

[44] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[45] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," in *ACM sigmod record*, vol. 29, no. 2. ACM, 2000, pp. 93–104.

[46] S. Ahmad and S. Purdy, "Real-time anomaly detection for streaming analytics," *arXiv preprint arXiv:1607.02480*, 2016.

[47] "Information technology – message queuing telemetry transport(ISO/IEC 20922)," International Organization for Standardization, ISO standard, 2016. [Online]. Available: https://www.iso.org/standard/69466.html

[48] (2014) Official MQTT documentation. Organization for the Advancement of Structured Information Standards. Last visited on 18-06-2018. [Online]. Available: http://mqtt.org/documentation

[49] A. Haller, M. Lefrançois, K. Janowicz, S. Cox, D. L. Phuoc, and K. Taylor, "Semantic sensor network ontology," W3C, W3C Recommendation, Oct. 2017, https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/.

[50] "OGC sensorML: Model and XML encoding standard," Open Geospatial Consortium, OGC standard, 2014, last visited on 18-06-2018. [Online]. Available: http://www.opengeospatial.org/standards/sensorml

[51] "Geographic information — observations and measurements (ISO/DIS 19156)," International Organization for Standardization, ISO standard, 2011. [Online]. Available: https://www.iso.org/standard/32574.html

[52] M. Lefrançois and A. Zimmermann. Custom datatypes: Towards a web of linked datatypes. École Nationale Supérieure des Mines de Saint-Étienne. Last visited on 18-06-2018. [Online]. Available: https://ci.mines-stetienne.fr/lindt/v2/custom_datatypes.html

[53] G. Shadow and C. J. McDonald, "The unified code for units of measure," http://unitsofmeasure.org/ucum.html, UCUM Organization and Regenstrief Institute, Inc., 11 2017.

[54] H. Rijgersberg, M. van Assem, and J. Top, “Ontology of units of measure and related concepts,” *Semantic Web*, vol. 4, pp. 3–13, 01 2013.

[55] R. Hodgson, P. J. Keller, J. Hodges, and J. Spivak. (2014, March) Qudt - quantities, units, dimensions and data types ontologies. [Online]. Available: http://www.qudt.org/

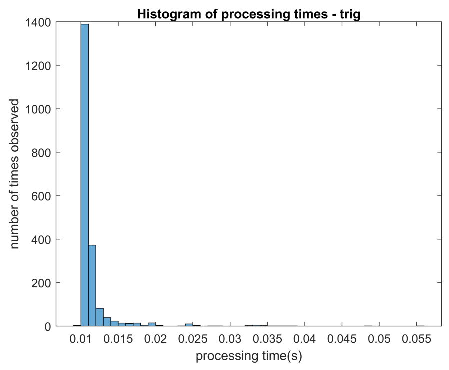
[56] G. Carothers and E. Prud’hommeaux, “RDF 1.1 turtle,” W3C, W3C Recommendation, Feb. 2014. [Online]. Available: http://www.w3.org/TR/2014/REC-turtle-20140225/

[57] P. J. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” 2005.

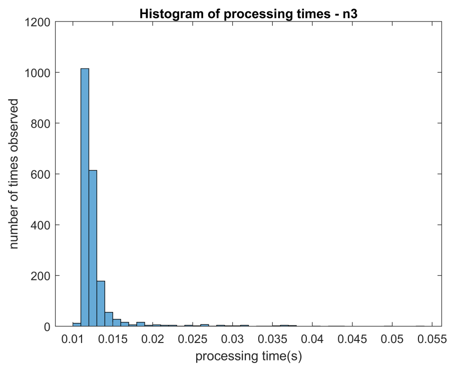
Appendix A

Additional measurement results

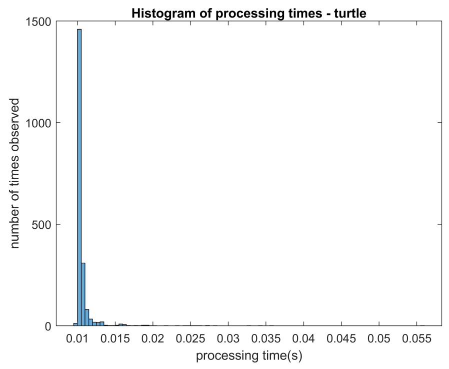
A.1 Distribution of the processing times at full capacity.



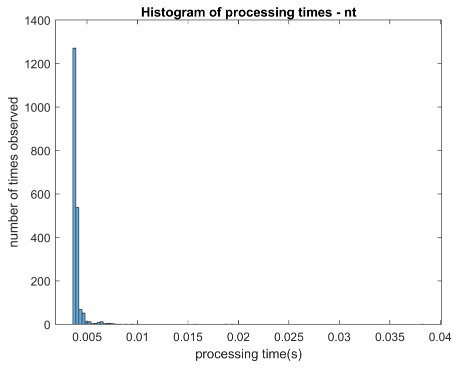
(a) Observed processing times of trig.



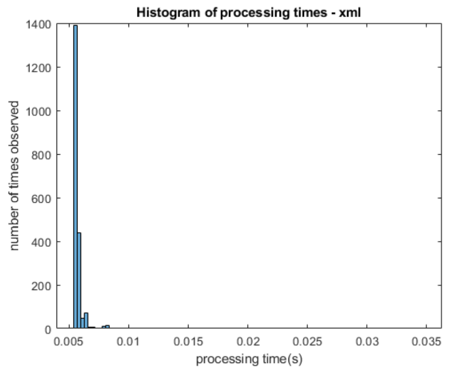
(b) Observed processing times of n3.



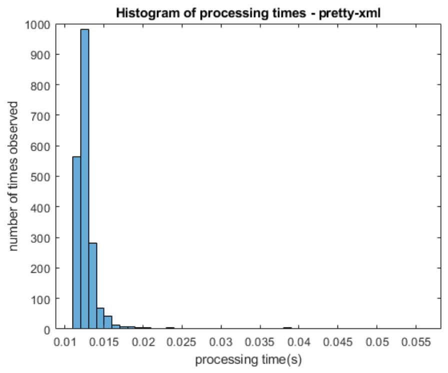
(c) Observed processing times of turtle.



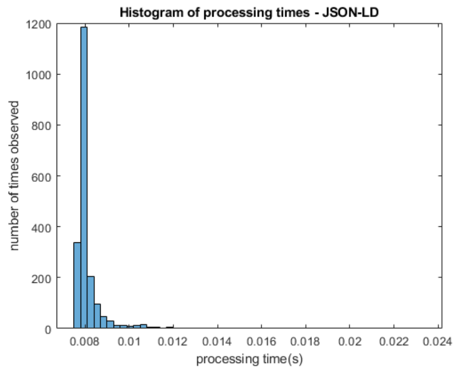
(d) Observed processing times of nt.



(e) Observed processing times of xml.



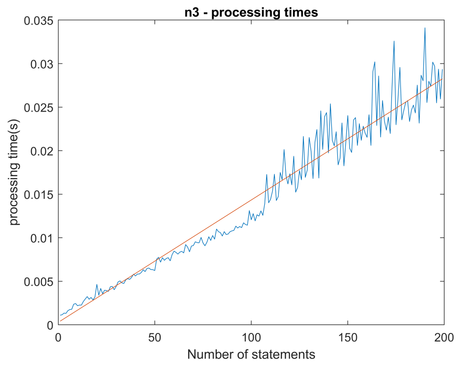
(f) Observed processing times of pretty-xml.



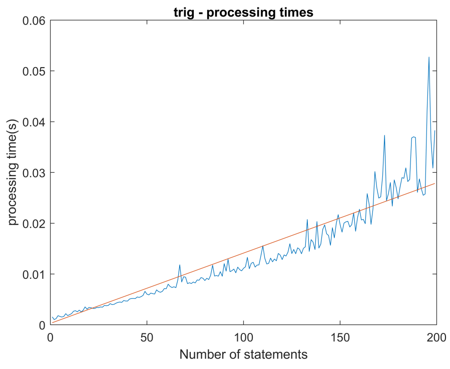
(g) Observed processing times of JSON-LD.

Figure A.1: Distribution of the processing times when running at full capacity.

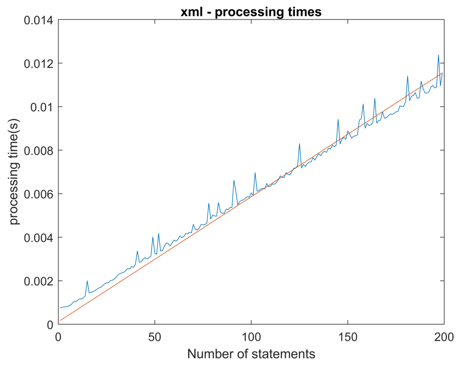
A.2 Processing times as a function of statement count.



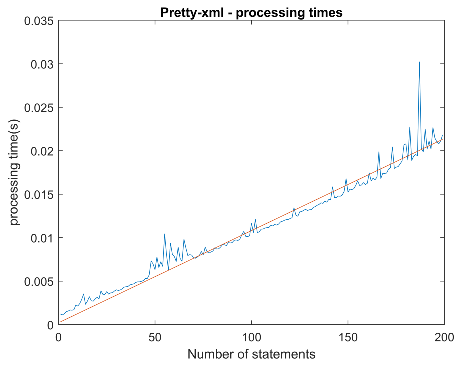
(a) Processing times for n3 format.



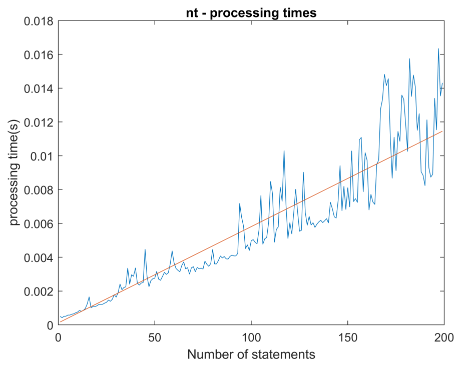
(b) Processing times for trig format.



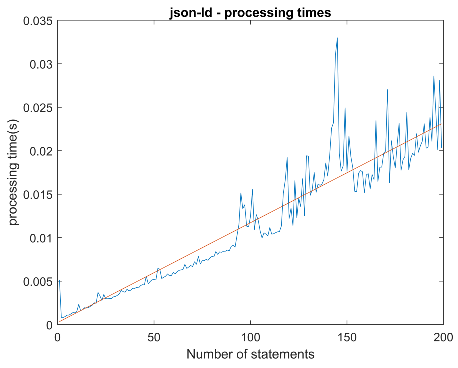
(c) Processing times for xml format.



(d) Processing times for the pretty xml format.



(e) Processing times for nt format.



(f) Processing times for json-ld format.

Figure A.2: Processing time as a function of amount of statements for each format.