Fast Data Processing in Hyper-Scale Systems

Marcel Tilly

Department of Computer Science

University of Leicester

A thesis submitted for the degree of

Doctor of Philosophy (PhD) in Computer Science

Dec 2014

Abstract

The deluge of intelligent objects that are providing continuous access to data and services on one hand and the demand of developers and consumers to handle these data on the other hand require us to think about new communication paradigms and middleware.

Based on requirements collected from scenarios from connected car, social networks, and factory of the future this thesis is developing new concepts for fast data processing for hyper-scale systems. In hyper-scale systems, such as in the Internet of Things, one emerging requirement is to process, procure, and provide information with almost zero latency. This thesis is introducing new concepts for a middleware to enable fast communication by limiting information flow with filtering concepts using event policy obligations and combining data processing techniques adopted from complex event processing.

Fast data processing has to deal with continuous data streams of events, providing a set of operators to manipulate, aggregate, and correlate data. This processing logic needs to be distributed. Distribution helps us to scale on one hand in terms of numbers of data sources (e.g. phones, cars, sensors) and on the other hand to parallelise processing in terms of grouping and partitions (e.g. regional).

In our solution, event policies are injected as close as possible to the place where the data is born to optimise traffic. Filters, aggregations and rules help to process the data accordingly. Finally, communication paradigms or interaction patterns support mediation between classical service based request-response interaction and event-based data exchange.

This all together builds a middleware enabling fast data processing for hyper-scale systems.

Declaration

The studies outlined in this dissertation were undertaken in the Department of Computer Science, University of Leicester and supervised by Dr. Stephan Reiff-Marganiec. I declare that this dissertation is my own account of my research and contains as its main content work which has not previously been submitted for a degree at any tertiary institution. All of the work submitted for assessment is my own and expressed in my own words. Any use made within it of works of other authors in any form are properly acknowledged at their point of use.

Research work presented in some sections has been previously published, in particular: the scenarios presented in Chapter 1 were published in (TRM11, TRMJ12, KHJ+14, ST13), the idea of low-latency data processing (Chapter 3) has been published in (TRMJ12), data processing in large, scale, distributed systems (Chapter 4) has been published in (MTJ14), and usage pattern for mediation (Chapter 5) is published in (TRM11), and parts of the evaluation (Chapter 6) is published in (MTJ14). I participated also in the projects described in section 2.5 and section 2.6 (IMC-AESOP and KAP) as part of my work. I present some results which I have created or to which I have contributed within these projects as background rather than claiming credit for them as thesis contributions.

Marcel Tilly

83737 Irschenberg, Germany

December 2014

To my lovely family

Acknowledgements

The list of people I want to thank is pretty long and I guess I might have missed some.

But first of all I would like to thank my Ph.D. supervisor, Dr. Stephan Reiff-Margeniec. His motivation and support was key for this thesis. When writing such a thesis there are periods when someone might get lost or just get unmotivated. In these periods Stephan was able to motivate me and find the right scope to continue. His guidance and experience was a helping hand for me to get through the jungle of writing a thesis.

I also want to thank my second supervisor, Dr. Neil Walkinshaw. I remember the first meeting with him when I was explaining the core idea of the taxi sample and he just implemented a small demo out of this discussion in the evening. I was impressed and it was the right thing showing that ideas we have discussed make sense at all.

Another person I want to thank is Dr. Helge Janicke. He has helped me to validate some ideas on top of ITL and as a result we have published two papers. Thanks Stephan, Helge, and Neil without your support and knowledge this thesis would have never happened.

I also want to extend my gratitude to Computer Science department at University of Leicester. All people, especially Dr. Fer-Jan de Vries, were very supportive and helpful with all the administrative work.

I am very grateful to Microsoft who gave me the opportunity to work on this thesis. There are plenty of people who contributed by discussions and opinions to this work. Let me thank Heike, Olivier, Ivo, Louis, Christian and Panos. Especially I would like to thank John

Lefor for his support on sorting out paperwork at Microsoft and also for his support in writing an understandable English. Thanks John and all the other guys, I enjoy the strong discussions with you. You just make work a place I want to be.

I believe this is also a good place to thank my parents for all their support through my life. Without you I would not be where I am today.

Writing a thesis is not only a personal mission but also a family affair. I know I was often sitting in my room trying to work on the thesis and do some research work on weekends and in the evenings. Well, normally this is the time I spent with my kids and my wife. Clearly, through the last 5 years I have taken most this common time from my family to work and focus on this thesis. Therefore and for thousands of other reasons a special thanks from the bottom of my heart to my wife Gesa and my kids, Luca and Jannik. I know there were periods, which were hard; I was not relaxed and very harsh when things were not going the right way. But I can tell you, this thesis contains a lot of your support, motivation, and love. Thanks for being there whenever I need you!

Contents

List of Figures

Chapter 1

Introduction

"If you want your children to be intelligent, read them fairy tales. If you want them to be more intelligent, read them more fairy tales."

– Albert Einstein

Today, there are various mega trends; people are talking about big data, cloud computing, service-oriented architecture (SOA), or the Internet of Things (IoT); just to name a few. All these trends have at least one common aspect: Data! There is a huge amount of data produced by a vast amount of heterogeneous sources, e.g. sensors, phones, cars, etc.. This data needs to be filtered, processed, and procured. Besides simply collecting all this data, there is rapidly growing demand to create timely insights into data. These insights can provide competitive advantages to business. Extracting relevant information from data or correlating data with other data sets as fast as possible is becoming a key factor for success. Latency, the time needed to process data, is getting more and more critical. It requires handling a huge volume of data, meditation between various data structures and processing a large number of sources.

Therefore, this thesis will investigate the following directions:

1. How can this data get processed as fast as possible?

2. How can relevant data be separated from irrelevant data?

3. How can data get filtered efficiently and in a scalable manner?

4. How can data from distributed, heterogeneous data sources and services be integrated into a system?

5. How to combine different technologies and different interaction patterns to make data flow efficient?

6. How to bridge between services, data, and consumers?

7. How to mediate between service offerings and consumer request in a timely manner, with almost no latency?

We should note that the huge data volume does not arise (at least in the scenarios we consider and address) from large single items of data that occur in scientific computing but rather from a very large amount of small items such as sensor readings.

To achieve almost zero latency data processing, data must be available at the place where the user needs it, such as a data provider. So, instead of pulling data at request time from data sources, data should be pushed to such a data provider. This is only the first step towards a faster processing of data in terms of providing results with low-latency. If the data sources are continuously pushing data to a data provider (e.g. the selector) there is a vast amount of overhead by unnecessarily transferring data – a waste of bandwidth. Especially for mobile devices the cost of bandwidth needs to be taken into account.

For examples there is a data provider, the one who interacts with the user. He knows when the user needs updated data and the intelligent data sources know about their situation. Thus, the data provider informs the sources under which changing situation (when) the sources should inform the data provider about the change of their properties (what). What and when can be expressed with event policies which are injected into the data sources, so that we can really make use of their intelligence. Thus, each data source will be responsible to make the projection from its own fine-grained, raw data to some more high-level, complex data which the data provider - and at the end the user - is interested in.

Data processing technologies, especially Complex Event Processing (CEP), are tackling the aspect of processing data with low latency. Some CEP systems are based on temporal algebra and provide consistent query languages based on strong formal languages. Therefore, the connection between service offerings, enriched by time information, and CEP-based events has to be made to improve processing time. A combination of service based systems and CEP, or more general event processing systems sounds logical. By extending event processing logic and rules for data to be processed already on the service side, close to the sources, is reducing the amount of data which needs to be sent around. The logic can also be used to correlate and aggregate data (events) on its origin.

This approach can be used to overcome the scaling problem when we talk about millions of services, data sources or simple sensors. The mediation between consumer requests and service offerings is the most challenging aspect because it requires pattern mining and detection. Pattern mining and detection can be achieved by learning from interaction between users and services. As soon the pattern is learned it can be expressed in terms of a query on the stream of incoming data or can be pushed to the service side.

This research investigates in combining existing paradigms, such as pub-sub approaches for processing service offerings and mediations with classical request-response SOA approaches for consumer requests, facilitated by data processing technologies, such as Complex Event Processing (CEP).

1.1 Motivation Scenarios

In real life there is a plethora of examples for hyper-scale setups, such as connected car green wave, smart factories or smart cities. In the connected car green wave scenario, car data and traffic signal data needs to be collected. The system will recommend to the driver the correct speed so that she can, depending on current traffic, catch the next signal light at green and does not need to stop. The current light signals state, the position and speed of the driver and the speed of the drivers nearby are relevant to find the correct speed. Clearly, some processing must happen in the cloud and maybe some processing in each drivers' car. The

calculation of speed and direction is the timely insight and has to happen ad-hoc and with low-latency otherwise the driver would not need it. This can happen locally. The correlation between cars has to happen in the cloud as central point for intelligence. Here, the logic has to partition cars in regions with the same direction (ideally on the same street) and calculate the ideal speed for them.

This scenario makes clear how important local and global processing is to enable fast insights and recommendation for the ideal speed for cars. However, the entire processing could happen in the cloud but this would not be ideal since a lot of raw, unneeded data would be sent around and would impact on bandwidth and network traffic. To find the right balance between what can or should be processed locally and what needs to be processed globally is actually one challenge and depends on the scenario.

In the following we present four scenarios from various domains to highlight the demand for a system which enables fast data processing for a huge number of data sources - hyper-scale system.

We introduce examples from fleet management, social network systems, insurance companies and manufacturing to motivate our approach. However, the approach is not limited to these scenarios and can be applied in a wide variety of applications where cloud services are matching a large set of potential consumers to providers, such as sensor network or logistics, e.g.

- Traffic sensor networks to monitor vehicle traffic on highways or in congested parts of a city.
- Parking lot sensor networks to determine which spots are occupied and which are free.
- Geo tracking of vehicles to support optimised routing of deliverables.

1.1.1 Fleet Management

In fleet management, like taxi companies, with a large amount of taxis it is almost impossible to use the classical request-response approach to find the nearest taxi for a given user location (see (TRM11, MTJ14)). Therefore, the fleet management

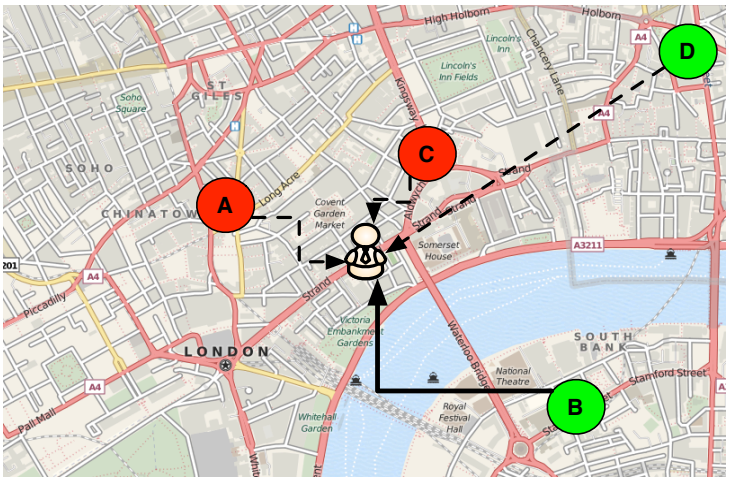


Figure 1.1: Fleet Management Sample

must be aware of the taxis location at any given time. There is usually no need to store all the provided locations of all taxis forever. The management system only requires the latest data to process a user request to locate the nearest taxi. There is no necessity to persist the data for later use. In the scenario (see Figure 1.1) there is a customer with a given context, his geo location, requesting a taxi. The fleet management system has to identify the most relevant taxi in terms of (1) availability and (2) proximity to the customer's location. There are two taxis, A and C, which are close to the customer's location but they are not available. Taxi B is the closest which is available. Of course the fleet management could take traffic information into account, and then maybe taxi D becomes the best solution because it is reasonably close, available and might arrive earlier because of beneficial traffic conditions.

This scenario shows

1. how different kind of properties of taxis (here: availability and geo location)
2. properties of different services (here: taxi and traffic) are used to select services, and
3. that taxis have to pro-actively inform the fleet manager about their location to enable fast and reliable responses to customer requests

Furthermore, the geo location and the traffic information are data which changes rapidly and it does not make sense to store all of this data because it is only short-lived and hence only the current values are relevant when a service has to be selected.

1.1.2 Friend-Near-by

Another example is extracted from social networks. In this example we are connecting virtual friends in real-life. Users of a social networks can provide their location information via a smart phone or tablet PC. Correlating this information with social network data, such as friends and their position, a service can send a notification back to the phone whenever one of the friends is close-by so that the user can meet his friends in real life (TRMJ12). To do so, the user has to provide his geo position and information about his social network, his friends, to a service. In this scenario we have two types of data: (1) location data which is time dependent and might change frequently and (2) social data which can be considered static. The location data needs to be sent regularly to the cloud and might cause a lot of data traffic if we consider a large number of users.

Assume there are Bob, Alice, Chris and Dave and they are all part of the same social network (see Figure 1.2). Alice is a friend of Dave, Chris and Bob, and Dave and Bob are also friends. While Bob is sitting at home, Alice is traveling with the metro from A to B, Dave is going by car from C to D, and Chris is in a rush and drives from E to F. Since Bob's geo position is almost static he only needs to provide his position once. In contrast, Alice, Chris and Dave are moving and hence need to provide frequent updates of their geo positions. The cloud service itself collects and processes all data. In case there is a match the service can inform the users. So that for example Alice and Dave can be informed at time t and position X that they are close to each other so that they can take this chance to talk to each other face to face. Since Chris is in a hurry, he has no time to meet and talk to one of his friends, he has set his availability to false. Although he could meet Alice as well in Y, his device should not provide any position data since there is no time anyway. The service would not propose that Chris and Dave should meet in Z because they are not friends.

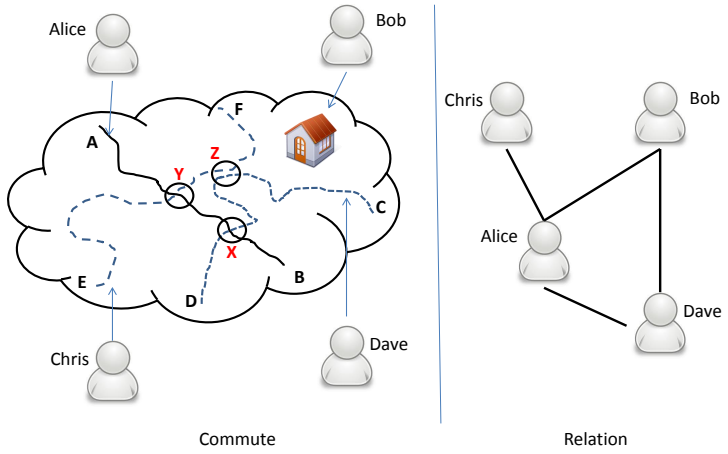


Figure 1.2: Friend-Near-by sample

What can we learn and extract from this scenario? This scenario shows

1. the need to process different kinds of properties (here: availability and geo location) in a timely manner

2. properties of different services (here: geo position and social network) are used to match user data

Thus, it makes sense to have rules running on devices which define which data should be used (*event*), when data should be sent (*condition*) and what data should be send (*action*). While each specific rule is very specific to a scenario, they help to balance between optimising data traffic and data accuracy in a cloud system (and actually in terms of privacy it is even better to keep less).

1.1.3 Pay-as-you-Drive

The privacy aspect becomes even clearer within this scenario. Pay-as-you-drive auto insurance is a type of automobile insurance whereby the costs of motor insurance is dependent upon the type of vehicle used, measured against time, distance, driver behaviour and place (see (CC12)). This differs from traditional

insurance, which attempts to differentiate and reward "safe" drivers, which is a reflection of history rather than present patterns of behaviour. This means that it may take a long time before safer (or more reckless) patterns of driving and changes in lifestyle feed through into premiums. With the pay-as-you-drive insurance model it is possible to provide a new model for costumers and taking physical data coming from the car and deriving behaviour.

We take the pay-as-you-drive insurance as motivating example for processing data already at the car level. The in car calculations can avoid on one side that raw, high frequent data is send and on the other sides it aggregates data and creates information in a way that privacy at some level is preserved. This requires a programming model to express logic in a way to aggregate raw personal data, such as GPS positions, and the ability to correlate results of aggregation with other data (e.g. weather).

This scenario can be used to consider some local calculations of the behaviour, acceleration, lane changes or heavy breaking and enriching it with more global available data, such as weather or traffic information. However, data sources in the car are the odometer giving the speed, or the breaks. The in car calculation will provide the basic driver index (BDI) (see Figure 1.3).

In the scenario there is Bob. Bob is having a fast car and he likes driving fast and changes lanes whenever there is a free spot. Sometimes he has to break heavily. All this makes him an aggressive driver. On the other side there is Chris. He thinks economically and adjusts his speed to the traffic. He avoids heavy breaking or any unnecessary lane switches. He is a safe driver. And then we do have all the other drivers in between.

So, we can define our BDI for this example as follows:

$$BDI = \sum_{i=1}^{n} w_i * f_i(p_i) \tag{1.1}$$

with

p_i : Properties contributing to index, such as speed, breaks etc.

f_i: A specific function or rule to extract a value from property

w_i: Weight of the value

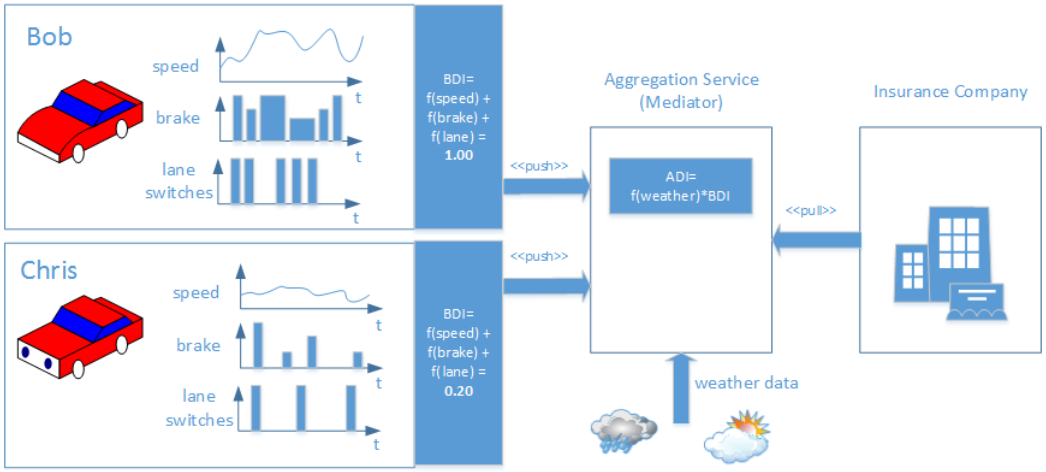


Figure 1.3: Pay-as-you-Drive Sample

There is also an adjusted driver index (ADI) which is the basic driver index adjusted by some external data, such as weather or heavy traffic. For example the BDI is calculated with perfect road and weather conditions, but if it rains or it is icy drivers needs to reduce their speed and have to drive even more carefully. This adjustment can happen in an additional service. We call it aggregation service (or mediator). This service has all external data sources available and can aggregate the weather information with the BDI coming from the cars. To preserve privacy in this setup the BDI cannot contain the exact GPS position but maybe a region; this will be sufficient for processing needs since weather information is also regional. Alternative the car could also calculate the ADI itself, if the car has a rain sensor for example. The ADI would be calculated as follows:

$$ADI = \sum f_i * BDI \tag{1.2}$$

The only information the insurance company would get is the adjusted driver index per driver; based on this the company can calculate the insurance fee. Well, this would be ideal and in reality the company needs more data as the BDI or ADI is more complex, but already the reduced example shows the potential of our approach.

This scenario shows

1. how local aggregation can support preservation of privacy
2. the need to mediate between different service interaction models, such as data streams (speed, breaks ...) and weather information pulled from weather services
3. the need for more complex logic and rules

Again, we have a rule running on the device which defines which data should be used (*event*), when data should be sent (*condition*) and what data should be send (*action*) and we do have a similar setup in the cloud performing the ADI calculation before the insurance company can access the data.

1.1.4 Shop Floor Monitoring

While in the pay-as-you-drive scenario privacy preservation is a predominant feature, latency and efficient data processing is more relevant in industry scenarios (ST13). In a typical scenario (e.g. within a manufacturing shop floor), sensors are connected to programmable logic controllers (PLCs), where each PLC is responsible for monitoring and controlling one machine. The PLCs forward information to a backend system where the signals can be correlated with manufacturing workflow data or enterprise resource planning (see Figure 1.4).

The operators' dashboards are reading and presenting the data from the backend system. Typically, there are already some static pre-filtering or down sampling rules on the PLCs (for example in normal situations, the PLCs should forward the temperature, vibration and rotation aggregated readings only every second). However, the PLCs have access to much higher data rates (up to 10 KHz). Suppose then that the PLC detects that the temperature is above a given pre-defined threshold and reports an alarm to the backend. In this situation, the operator would like to get more information from the machine/PLC associated with the alarm. Therefore, he grabs his mobile device (e.g. tablet or smart phone) and goes to the shop floor to check the machine personally. In front of the failing machine, he would now (1) like to connect to the machine directly and (2) inspect the high-resolution data streams.

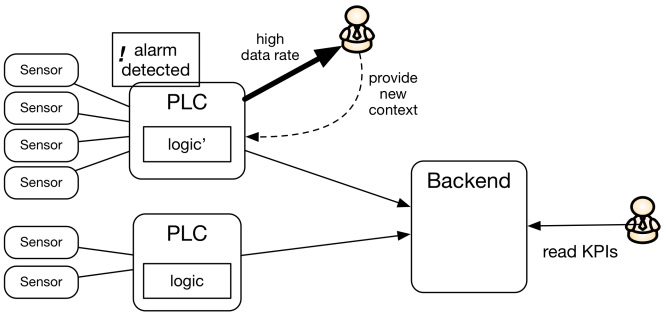


Figure 1.4: Shop floor monitoring sample

To enable this, the operator's mobile device must be able to run local queries and the system managing the analytics must be able to detect the new context of the operator (in front of the machine) in order to re-arrange the setup of logic, so that he, the operator, can receive the high-rate data streams from the PLCs.

This scenario shows

1. the demand to enable the integration of different systems (sensors, MES, ERP)

2. the demand of data processing rules acting on context information to adjust behaviour

However, the system needs to adjust its behaviour provided by context information. Here, we have the source of data, the temperature (*event*) and the defined threshold (*condition*) so that the operator can receive an alarm (*action*). In cases the operator is in front of the machine the condition, the sampling rate, and the action (send data to different endpoint) are changing.

1.1.5 Alarm Management

The shop floor monitoring sample is highlighting the demand of fast data and context relevant data processing. This gets even clearer within alarm management (see (KHJ+14)).

In alarm situations in industry, such as refineries, operators have not to deal with one single alarm but with a so-called alarm cascade. A huge number of

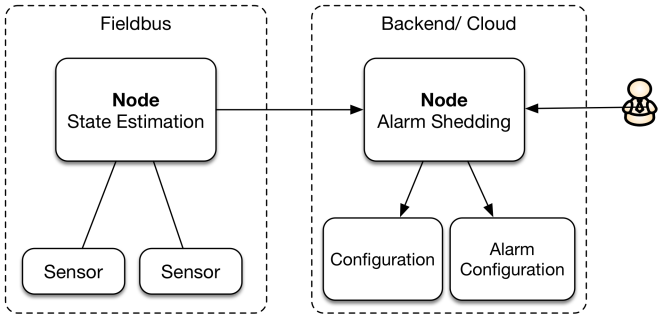


Figure 1.5: Alarm management sample

alarms from various sensors and systems are forwarded and flooding the entire system and the operators dashboard – leaving them to determine the root cause of the problem. Here, it would be very helpful if the system could hold back minor prioritised alarms and give high priority to crucial alarms and data. It would be good to have a set of rules on sensors and higher-level devices to decide which alarm (or event) should go through in which context. This has to happen as fast as possible and instantaneously.

In our example we have a two stages approach (see Figure 1.5). First there is the state-based alarming. This is an alarm management method based on switching of the alarm system configuration to the settings which correspond to the identified process state. As stated in (KHJ+14), this technique aims at eliminating occurrences of alarms inappropriately triggered by normal process changes within given process state. This can already help to reduce the number of alarms. As an example for potential states:

- State 0: normal state (the alarm limits design corresponds to light crude oil fed into the column at medium flow rate)
- State 1: light crude oil and low input flow rate
- State 2: light crude oil and high input flow rate
- State 3: heavy crude oil and high input flow rate

These states can be forwarded to some backend node which can use the state to pick the right configuration for state and alarm to enable alarm load shedding.

In alarm load shedding the operator is interested in prioritising of actions during alarm floods (i.e. an abnormal situation in operations when the number of alarms exceeds the operator alarm handling capacity.). The alarm load shedding method limits the rate of displayed alarms with a targeted alarm rate or approximately one alarm per minute (when feasible), which is considered to be the alarm rate that the human operator can still handle. This method is based on ranking alarms with respect to their urgency and delaying less urgent alarms from being displayed. Given its nature, this method is suitable for abnormal process situations, which have not been handled by state-based alarming. As a result of this method, the alarm load is distributed more evenly for the operator and most urgent alarms are handled first.

Examples are:

- Percent of time in flood state: The proportion of time that the operator console is flooded with alarms. The rate at which a single operator is overwhelmed by alarm activations (i.e. when the alarm count per 10 minutes exceeds 10 alarms).
- Average number of alarms in 10 minutes: The alarm rate that the operator is able to efficiently handle in long term is less than 1 alarm in 10 minutes.
- Peak number of alarms in 10 minutes: The maximum rate for the most active 10-minute interval within the evaluated time period is 10 alarms.
- Peak Alarm Minute Rate: The target peak minute rate for the most active minute within the evaluated time period. Target = 2 per minute

This scenario highlights the need to express conditions on which actions are triggered based on some highly frequently changing data (sensor data) and static definitions of configurations. Furthermore, it show the requirements of temporal computations, or windowing, as we have seen above.

1.1.6 Summary

In general we can extract the following list of requirements from provided scenarios:

Scenario	Requirement
Fleet Management	• [R1.1] Processing of dynamic and static data and dynamic properties • [R1.2] Data traffic reduction by local processing running on nodes • [R1.3] Timely insights through recurrent data processing
Pay-as-you-Drive	• [R2.1] Privacy preservation through local data aggregation and calculation • [R2.2] Global correlation of local results with global services • [R2.3] Support of huge number of data sources
Friend-near-by	• [R3.1] Support of huge number of data sources (users and phones) • [R3.2] Privacy preservation through local data aggregation • [R3.3] Fast processing of dynamic and static data
Shop Floor Monitoring	• [R4.1] Timely insights in production process (KPIs) • [R4.2] Easy to adapt setup to context changes (operator position) • [R4.3] Integration of heterogenous data sources in terms of number and interaction • [R4.4] Smart local aggregation to avoid unneeded data traffic
Alarm Management	• [R5.1] Timely insights in alarm situations • [R5.2] Easy to adapt setup to context (alarm state) • [R5.3] Processing of dynamic data to detect alarms and adapt to static configuration • [R5.4] Integration of potentially many different data sources in terms of number and interaction • [R5.5] Local processing for pre-filtering and global processing for adaption • [R5.6] Temporal or windowed aggregations (e.g. every 10 minutes)

1.2 Research Challenges

Based on research direction described above, the following concrete topics have been identified and will be addressed at the different phases of this PhD project. In the following, each topic reflects one research direction discussed in the previous section (see Figure 1.6).

Speed: Fast processing of data to provide timely insights
Requirements: [R1.1],[R1.3],[R3.3],[R4.1],[R5.1], [R5.6]
There is a demand on getting results as fast as possible. New paradigms are needed to improve the speed on processing service requests or in processing data provided by data sources, such as sensors. Basically, it makes sense to rethink classical request-response SOA approaches and to ensure that service offerings process data in almost real-time. Furthermore, we require a way to express data processing in a way that it fits in certain systems. This is a key challenge for moving forward towards to the next generation of the SOA The research issues are:

- How to ensure that service offerings and data are processed as fast as possible?
- What interaction pattern to choose to enable provisioning of information with almost no latency?
- What model to express data processing?
- Which set of operators is required?
- How to enable aggregations over time windows?

Scale: Be smart in terms of which data to process and where to optimise data traffic.
Requirements: [R1.1],[R1.2],[R2.1],[R2.2],[R2.3],[R3.1],[R3.2],[R4.2],[R4.4],[R5.5]
Considering sensors, cars, phones or other data sources as potential services, which need to be integrated and their data be processed, requires to rethink the pure forwarding of raw data to a single process or backend. Instead of bringing the data to the processing we have to consider bringing the processing to the data

instead. Logic should run on nodes and be smart in terms of forwarding data. Ideally a master service can control when data is forwarded and which data is forwarded through terms of aggregation or batching. A higher level of control of transferred data is required to optimise data traffic. Controlling the data flow and the amount of data without missing any insights enables scaling up to millions of services sending data around. The research questions are

- How to reduce network traffic in large scale service networks?
- How to express logic for nodes?
- How to distribute logic within the topology?

Mediation: Combine request-response interaction with event-based interaction efficiently.
Requirements: [R1.1],[R2.2],[R3.3],[R4.3],[R5.3],[R5.4]
In context of this thesis mediation is used to describe the combination of request-response interaction pattern and event-based interaction. These different pattern needs to be combined and there must be concepts enabling their seamless integration.

- How to mediate efficiently between consumer requests and service offerings?
- Who is defining mediation rules? Is this automatic or defined by domain experts?
- How to combine push- and pull-model in an effective way?

The research presented in this dissertation addresses these 3 challenges. In particular, we present a novel approach for processing temporal data in an efficient way and a novel approach to use the actor model for highly-distributed data pipelines. Based on these concepts, we develop a new architecture for fast data processing in hyper-scale systems.

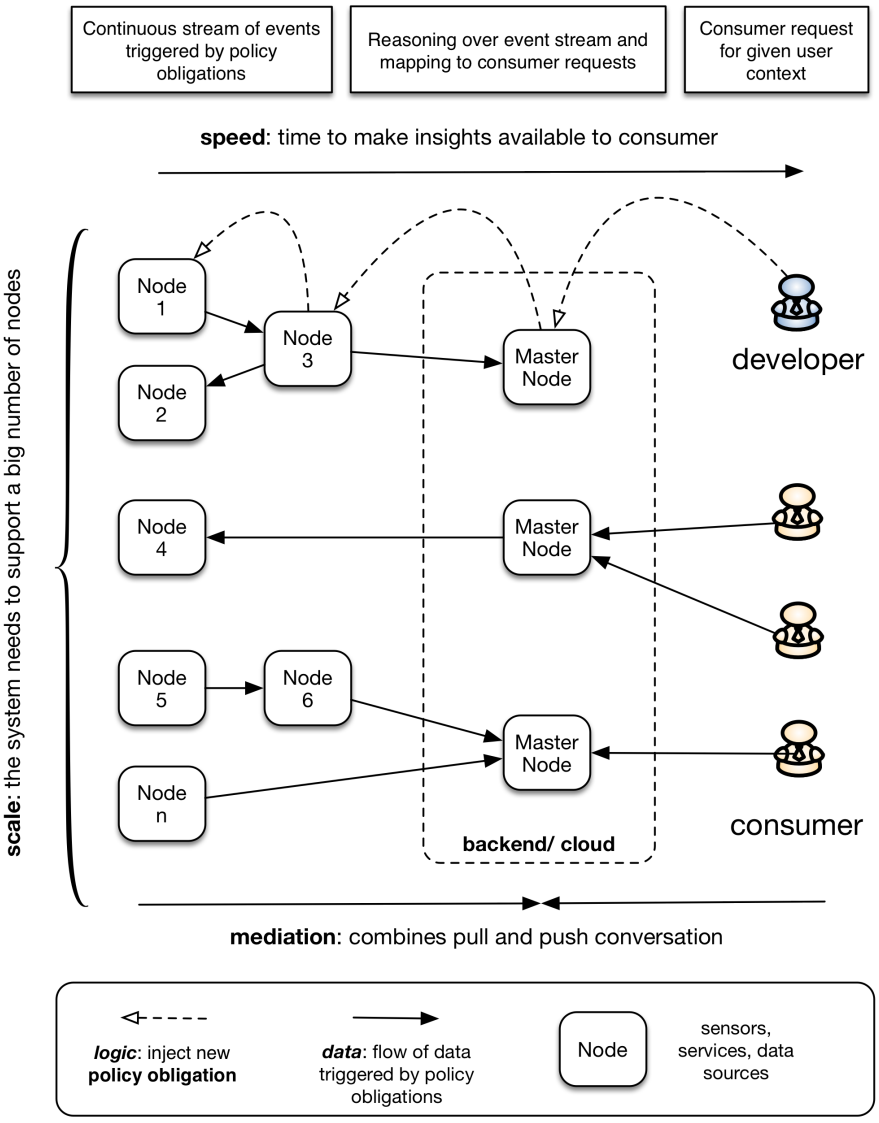


Figure 1.6: Challenges of the thesis

1.3 Research Aims and Statement

The overall aims of this dissertation are developing a model for enabling fast data processing in hyper-scale systems. It is useful to further divide the aims into more specific problems, objectives and questions.

Aims

1. Data processing model to process temporal service data with near zero latency.

   This aim is looking into a formal way to express data processing as a pipeline of transformations from sources to action. Ideally, the model is based on formal logic as grounding.

2. Model to enable data processing in highly-distributed and large systems.

   This aim is tackling the scale challenge. The model has to scale up to a huge number (millions) of different data sources. With millions of data sources or services the model has to enable a way to distribute logic over data. It ensures to bring the processing to the data instead of bringing the data to the processing. This includes the development of flexible distribution concepts and methods to be smart in terms of bandwidth usage and network traffic.

3. Develop an architecture to mediate between different interaction pattern.

   Within a heterogenous setup we can expect different interaction pattern, e.g. request-response or event-based. An architecture has to provide means to integrate these patterns in an coherent way.

There are a number of objectives that need to be met by the approach.

Objectives

1. The formal model for fast data processing shall respect continuous data streams and time related properties.
2. The formal model shall be easy to use and shall support simple types of data.
3. The underlying architecture shall support the distribution of logic to nodes to enable filtering at the source.
4. The architecture shall be capable to deal with a large number of services, sensors and data sources.
5. The mediator architecture shall have low complexity to integrate push and pull conversations.
6. The mediator architecture must provide an effective way to handle large amount of data.

To address the first two objectives, we need to answer the following questions.

- Which model can provide a good foundation for processing time relevant properties?
- How can we ensure correctness of processed data?
- What does correctness mean?
- How to connect the model with the physical world in terms of data sensor readings and actuations?
- What is an easy way to create a pipeline to process service or sensor data?
- What influence does the pipeline have on latency?

Regarding the third and fourth objective, three research questions should be considered:

- What kind of lightweight model can run on nodes?
- Is there a way to distribute logic across a network?
- How to combine the architecture with the formal model?

The last two objectives are related to the following research questions.

- How can we measure scalability?
- Is there a way to optimise network traffic?

In this dissertation, we address all these questions, however, some aspects are considered in greater depths.

Thesis Statement

The dramatic increase of data coming from heterogenous data source and the need to process and procure these data to provide insights in short timeframes is a key challenge in software engineering. We show an approach to enable fast data processing in a highly distributed setup with different interaction patterns in a feasible and practical way.

1.4 Key Terminology and Assumptions

To ensure a focus on the outlined research aims, objectives and questions, some crucial terminology and assumptions have to be established. Some of these terms are already defined in other scopes or might have ambiguous definitions. Therefore the following collection is summarising terms which are crucial to put this thesis in the right context. Furthermore, in the young field of data processing in cloud scale systems different researchers use some terms in slightly different ways. To avoid ambiguity we present our use of the relevant terms.

- In general we are talking about *big data* in this thesis. In contrast to some other definitions big data is in this thesis the result of small chunks of data produced by a huge amount of various data sources. These data sources are producing the data recurrent and frequently, e.g. one data point every second or even in sub-second intervals. We can find this setup in IoT, like SmartCities etc.. A huge amount of sensor across a city are producing temperature, traffic or air pollution data (just to name a few). In addition to these sensor data we can find car telemetry data or weather information from services. All this data is usually small in terms of size but big in terms of numbers and data rate. As a result we are talking here about big data, too.

- With this thesis we would like to introduce the term *fast data*. In contrast to big data which usually just tries to characterise the plain quantity of data the term fast data tries to focus more on the speed aspect the data needs to be processed. While speed is just the time between when the data is born up to when relevant insights are available.

- We have mentioned *hyper-scale* systems already sometimes. In scope of this thesis a hyper-scale system has to deal with the huge amount of data sources, services and sensors. There is no number to quantify a hyper-scale system, but we would except thousands and thousands of data sources.

- *Data sources, services and sensors* are basically used within this thesis to provide examples for nodes which can deal as a source of data. We would like to call all of them services, but sometimes we are using the term data source or simply node as a synonym.

- Sometimes we might use the term *real-time data processing*. We just want to point out that real time in scope of this thesis is more soft real-time or near real-time. It is not meant as hard real-time as we can find in industrial control systems or other critical systems. The real time in this scope is not ensuring that a result is available within a given time. It is more about processing the data as fast as possible without any additional saving to databases.

- The data sources and services are producing a continuous flow of data. There is no beginning and end. Thus, we are talking about a *stream* of data points.
- A data point is also called an *event*. We will provide a more solid definition later in this thesis but for general understanding an event could be seen as a data point at a given time.
- So far the term *logic* was mentioned. Logic is a piece of executable software code. Later we will replace the term logic with the term expression.

1.5 Methodology and Contributions

We use the case study based research methodology. Our research began with analysing the real world data processing scenarios taken from IoT. The alarm management scenario is a result of discussions we had in the IMC-AESOP project (HMT+14), (JKB+14), and (KHJ+14).The other scenarios were taken from customer interactions in the scope of manufacturing, smart cities and fleet management. Based on discussions and requirements collection with customers the presented scenarios were developed and representing the state of the art of current discussions within this domain.

The main contributions of this dissertation are:

- A syntax to describe data processing pipelines from sources to actions. The main focus of the syntax is on expressing transformations from input data and trigger actions on the output side (Chapter 3).
- An architecture to enable a distributed pipeline setup over a heterogenous set of nodes (Chapter 4).
- Event policies which help to optimise data traffic in hyper-scale systems (Chapter 4).
- A mediation model to integrate pull and push interaction into one coherent architecture (Chapter 5).

1.6 Dissertation Outline and Summary

This chapter discussed the purposes of the dissertation - addressing fast data processing and data mediation for large scale, distributed systems. We explained the crucial challenges of combining different interaction paradigms to address issues around services and efficient data processing through an enhanced model in combining push and pulls-based models.

In order to draw our research borderline, we listed the interested research aims, objectives with their related questions and provided focus with a clear thesis statement. The key assumptions and terminology were explained. In addition, we highlighted our research contributions in the previous section. The reminder of this dissertation is organised as follows:

- In Chapter 2, we give a more detailed description of the research background. Specifically, we are going to discuss the related work on service and service-oriented architecture in the scope of policies, event-based interaction, interval temporal logic, and event data processing.

- In Chapter 3, we focus on time, time dependent service offerings and how this can be processed with available logic over a stream of events.

- In Chapter 4, we illustrate a new approach to enable data processing over big, distributed system. Therefore, we are going to extend the actor model into a code injected actor model.

- In Chapter 5, we investigate how push and pull-based data capturing can be provided to consumers and how this can we used in junction with the architecture to mediator easily between different interaction models.

- In Chapter 6, we implement core parts of the proposed approach, measure latency and scalability. Furthermore, we discuss system evaluation results for speed and scalability and compare this with existing technologies.

- In Chapter 7, we provide a concluding discussion and identify potential future research directions.

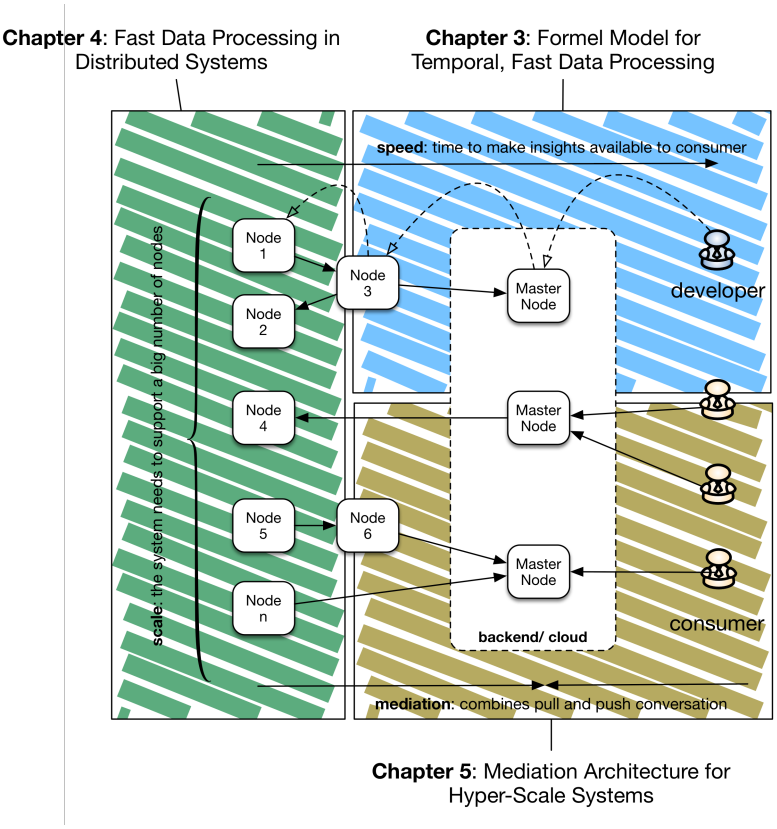


Figure 1.7: Structure of the thesis

The figure 1.7 shows how the chapters map to the three overall challenges described in 1.2:

Chapter 2

Research Background and Related Work

"There are three stages in scientific discovery. First, people deny that it is true, then they deny that it is important; finally they credit the wrong person."

– Bill Bryson, *A Short History of Nearly Everything*

This thesis covers different topics. While there is existing work on non-functional properties, service selection, complex even processing and temporal logic, there is no work as far as we are aware which combines these approaches to solve selection problems for large-scale systems.

We will start with a short overview of state-of-the art of the surrounding field without going into details that are not directly relevant for this thesis. After the overview we will dive into some more specific topics, which are relevant for this thesis and provide some more background. Thus, we will go into more detail for interval temporal logic (ITL), event data processing or complex event processing, and event-based interaction patterns. All three topics have direct impact on this

thesis and therefore we will provide a deeper introduction into these topics.

2.1 State-of-the-Art

To capture state-of-the-art for some topics is quiet challenging since research is moving forward quickly. Therefore, it is hard to give a full overview of the state-of-art. We have picked the most relevant work, which is in some aspects touching this thesis. We will look into work around (1) rules and reactive systems, (2) data processing in regards of complex event processing, (3) services, service properties and service selection, and (4) ECA rules.

The use of Event-Condition-Action (ECA) rules is well established in Data-Stream Processing applications (For90, CM94) and ECA-based policy languages (TLDS08, UBJ+03) are used to govern the behaviour of systems on the basis of these rules to control and manage distributed systems (Slo94). In our work, we are however mainly concerned about the selection and propagation of events in a P2P infrastructure. The formalisation of event policies in this work differs from traditional ECA rules in that the condition does not only describe a Boolean combination of events, but can address the history of a selected event stream that allow to specify the *distance* between propagated events using ITL (CMZ11).

Data Stream processors such as SNOOP (CM94) and successors already use event histories for detecting the order of events, making this a natural model for expressing policies that also allows for the efficient enforcement of such policies (JCSZ07). The semantics of event-policies is based on temporal projection (Mos95) as this is a natural abstraction technique for complex system specifications. Other work by Duan et.al. (DK04, TD09) on propositional projection temporal logic would provide alternative formalisations of projection, but lead to a more complex formalisation of the event policies without apparent gain in this application context. The use of policies together with a mediator has also been suggested in a different context by Edge et.al. (ESPC08) where they focus on the mining institutional transaction data for fraudulent activities. However, their use of policies is targeted to this particular application domain and is focused on

the detection of events, whereas we address the problem of event filtering and propagation as part of an infrastructure for event driven P2P systems.

As already mentioned there is a lot of work about service selection based on non-functional properties. (YRM08) provides a survey and classification of service selection based on non-functional properties. Most of the related work on using non-functional properties for service selection concentrates on defining QoS (Quality of Service) ontology languages and vocabularies and identification of various QoS metrics and their measurements with respect to semantic services. In (YZL07) QoS ontology constraints for efficient service selection are described, while (RMYT09) separates different non-functional criteria into different service categories. This is more sensible than ranking all kinds of services by using the same predefined criteria and hence not considering the different attributes that occur with specific services. All these approaches are lacking temporal aspect or NFPs.

Bonifati et al. (BCP02) describes a very interesting approach for using active rules for pushing reactive services. But it does not take into account temporal aspects or states. Roitman et al. (RGR09) presents a framework for satisfaction of complex data needs involving volatile data. But the focus is on pull-based environments. We believe that our approach is more promising for large-scale systems. With push based systems, data is pushed to the system and the research focus is mainly on aspects of efficient data processing, where load shedding techniques (TLPY06) can be applied in order to control what portions of the pushed data to process and to increase latency. Such systems include publish-subscribe (pub/sub) (DGH$^+$06), stream processing (ACÇ$^+$03), and complex event processing, however there is no consideration of bandwidth consumption.

In the following we will look into some specific topics in more details, which are relevant for the carried out research in chapters 3, 4, and 5. Interval temporal logic is key for chapter 3 where we introduce the programming model for past data processing. Event data processing or complex event processing inspires section 4. Therefore, we will look into more detail into some CEP concepts, which are relevant. Finally, interaction patterns, pull- and push-based interaction, are building the baseline for chapter 5. So, we will provide some overview here as well.

Expressions
$e ::= \quad \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc v \mid \mathsf{fin}\ v$
Formulae
$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid$
$\mathsf{skip} \mid f_1 ; f_2 \mid f_1 \Delta f_2$

Figure 2.1: Syntax of ITL

2.2 Interval Temporal Logic

The key notion of ITL (CMZ11) is an *interval*. An interval σ is considered to be a (in)finite sequence of states $\sigma_0,\ \sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables *Var* to the set of values *Val*. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, so a one state interval has length 0).

The syntax of ITL is defined in Figure 2.1 where μ is a constant value, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol. The syntax is based on (CMZ11), however uses the projection operator $f_1 \Delta f_2$ as primitive and derives the operator f^* as introduced in (Mos95).

The informal semantics of the most interesting constructs are as follows:

- skip: unit interval (length 1, i.e., an interval of two states).
- $f_1 ; f_2$: ("chop") holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval. Note the last state of the interval over which f_1 holds is shared with the interval over which f_2 holds. This is illustrated in Figure 2.2.
- $f_1 \Delta f_2$: ("projection") is defined to be true on an interval σ iff two conditions are met. First, the formula f_2 must be true on some interval σ' obtained by

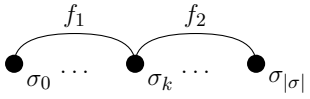


Figure 2.2: Informal Semantics of $f_1 ; f_2$

projecting some states from σ. Second, the formula f_1 must be true on each of the subintervals of σ bridging the gaps between the projected states.

An example is depicted in Figure 2.3.

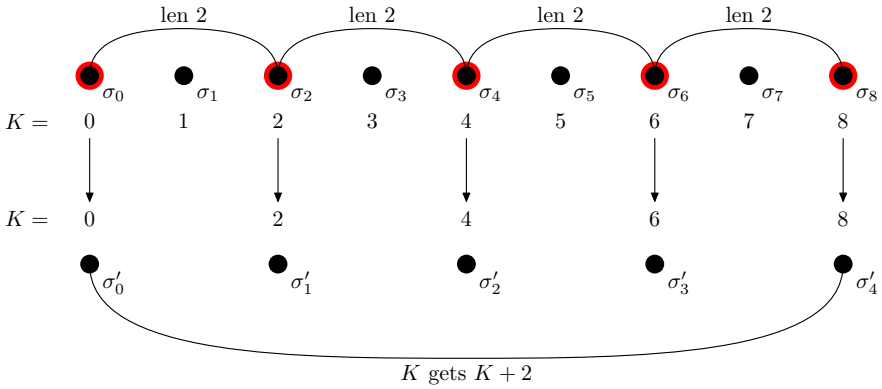


Figure 2.3: Example of Temporal Projection

In the interval σ the value of K increases from 0 to 8 in steps of one. The interval σ satisfies $(\mathsf{len}(2))\Delta(K \text{ gets } K+2)$. $(\mathsf{len}(2))$ is true if the interval is of length two and $(K \text{ gets } K+2)$ is true if the K increases by 2 from state to state. The gaps between the projected states (highlighted in red) are bridged by the formula $\mathsf{len}(2)$. The formal definition of this operator is given in (Mos95).

- $\bigcirc v$: value of v in the next state when evaluated on an interval of length at least one, otherwise an arbitrary value.
- $\mathsf{fin}\ v$: value of v in the final state when evaluated on a finite interval, otherwise an arbitrary value.

2.2.1 Derived Constructs.

The following lists some of the derived constructs used in the remainder of this paper. The binary operators $\vee$ (or) and $\supset$ (implication) are derived as usual.

- $\bigcirc f \mathrel{\widehat{=}} \mathsf{skip} \,;\, f$ (read "next f"), means that f holds from the next state. Example: $\bigcirc(X = 1)$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.
- $\mathsf{more} \mathrel{\widehat{=}} \bigcirc\mathsf{true}$ means the non-empty interval, i.e., any interval of length at least one.
- $\mathsf{empty} \mathrel{\widehat{=}} \neg\,\mathsf{more}$ means the empty interval, i.e., any interval of length zero (just one state).
- $\mathsf{inf} \mathrel{\widehat{=}} \mathsf{true} \,;\, \mathsf{false}$ means the infinite interval, i.e., any interval of infinite length.
- $\mathsf{finite} \mathrel{\widehat{=}} \neg\,\mathsf{inf}$ means the finite interval, i.e., any interval of finite length.
- $f_1 \frown f_2 \mathrel{\widehat{=}} f_1 \,;\, \mathsf{skip} \,;\, f_2$ weak chop, like chop but f_1 and f_2 don't share a state.
- $\Diamond f \mathrel{\widehat{=}} \mathsf{finite} \,;\, f$ (read "sometimes f"), i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.
- $\Box f \mathrel{\widehat{=}} \neg \Diamond \neg f$ (read "always f"), i.e., any interval such that f holds for all suffixes of that interval. Example: $\Box(X = 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.
- $\boxed{\text{i}}\, f \mathrel{\widehat{=}} \neg(\neg f \,;\, \mathsf{true})$ box-i, i.e., any interval such that f holds over all prefix sub-intervals.
- $\boxed{\text{a}}\, f \mathrel{\widehat{=}} \neg(\mathsf{finite} \,;\, \neg f \,;\, \mathsf{true})$ box-a, i.e., any interval such that f holds over all sub-intervals.
- $\mathsf{fin}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \supset f)$ defines the final state, i.e., any interval such that f holds in the final state of that interval.

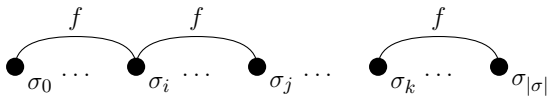


Figure 2.4: Informal Semantics of f^*

- $\mathsf{halt}\ f \mathrel{\widehat{=}} \Box(\mathsf{empty} \equiv f)$ terminate the interval when f holds.
- $\exists v \bullet f \mathrel{\widehat{=}} \neg \forall v \bullet \neg f$ existential quantification.
- $\mathsf{len}(e) \mathrel{\widehat{=}} \begin{cases} \mathsf{false} & \text{if } e < 0 \\ \mathsf{empty} & \text{if } e = 0 \\ \mathsf{skip}\ ;\ \mathsf{len}(e-1) & \text{if } e > 0 \end{cases}$ holds if the interval length is e.
- $v\ \mathsf{gets}\ e \mathrel{\widehat{=}} \Box(\mathsf{more} \supset (\bigcirc v) = e)$ gets, i.e., in every state except the initial state the variable v will be assigned the value of e evaluated in the previous state.
- $f^* \mathrel{\widehat{=}} f \Delta \mathsf{true}$ (read "f chopstar") holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds. This is illustrated in Figure 2.4.

2.3 Event Data Processing

In the last decades due to the significant increase in speed and volume of information flows, it has become vital to have technologies and automation tools allowing us to handle very high data rates with strict latency requirements. These demands are shared by many event-driven organisations in different areas of industry such as manufacturing, health care, financial services and web analytics. Complex event processing (CEP) is a technology of continuous real-time data processing, which satisfies the above requirements.

CEP can be seen as continuous and incremental processing of event streams from multiple sources based on declarative query and pattern specifications with near-zero latency. CEP enables monitoring and analysis of events happening on

different levels of organisation, helps in identifying their relationship and correlation, and triggers immediate subsequent actions.

2.3.1 Events

An *Event* is a basic concept of CEP. It defines something that happened. There are plenty of events in our every-day life, for example: a historical or social happening, a car accident or receiving of a text message. There are different formal definitions of this term in the literature ((LS08), (GPMF06)). Basically, we can define *event* within this context as any happening of interest that can be observed, recorded and reacted on in a system.

An *event* is created in two distinct steps: observation and adaptation. Firstly, a particular activity of a system should be observed without changes to the systems behaviour. Then observed activities are transformed into event objects that can be processed by the CEP system. The transformation is usually done by special entities called adapters.

As described in (Luc02) we can distinguish three major aspects of an event:

1. **Form**: The form constitutes the data associated with an event. It can contain different attributes or data components. The typical data components of an event are, for instance, a timestamp when the event happened, a source of the event and its longevity. The same event can have different forms communicating different data to interested parties. In the literature, by event/notification it is often meant a complete event object (an event together with its form) (see also (GPMF06)).

2. **Significance**: An event represents an activity. The corresponding activity of an event is called its significance.

3. **Relativity**: Events can be independent or dependent on other events. They can relate by causality, time and aggregation. The relationship between events reflects the dependencies of the corresponding activities and is called relativity. Usually, the events form encodes its relativity, so that the dependencies on other events can be derived from the given event.

In natural language or everyday life an event is defined as something that happens or is supposed to happen. This understanding is partly valid for events in CEP. Because in CEP an event is represented by an object which is triggered by an activity or happening, the significance, but it is not the activity itself.

An event is also not just a pure message. The form of such an event could be represented by a message, but it also needs to cover significance and relativity. Dependencies and relationships between events can have different form and are often the main points of interest for CEP applications.

2.3.2 Event Relationships

D. Luckham (Luc02) describes three common relationships between events:

1. **Time**: Time is used to order events, defining which of the two events A or B happened earlier. Usually, as soon as an event is created, the current time is added to the event object in a form of a timestamp. So, timestamps define time relationships between events. This type of relationship depends on the clocks in the system. Events can have different time relationships corresponding to different clocks. Event comparability in this case depends upon whether the clocks are synchronised.

2. **Cause**: Event A causes event B, if the activity signifying event A had to happen in order for activity B to happen. Accordingly, causality is a relationship of dependency between activities. If event B can happen only after event A, then event B depends on event A, or event A caused event B. When neither of the events depends on each other, we say the events are independent. The relationship of time and cause is expressed in the following axiom, which is valid for most systems: If event B is caused by event A in system S, than no clock in system S will give an earlier timestamp to event B than to event A. Consequently, if the clocks can observe two dependent events, they will always observe them in the same order.

3. **Aggregation**: If the activity corresponding to event A consists of other activities corresponding to a set of events $B_1...B_n$, then event A is an aggregation of the events B_i. In this case we say that events $B_1...B_n$ are

members of event A. Aggregation is a mean to abstract events, enabling introduction of vertical dependency. Furthermore, the activity signified by event A is more complex than the activities of member events. Because of that, its signifying event is called a complex event. It is easy to see, that aggregation leads to a causal relationship between the complex event and its members. Activities signified by complex events often happen over a period of time and instead of a timestamp a time interval is used. The time interval starts with the earliest activity and ends with the end of the latest activity from the events members.

2.3.3 Root-Cause Relations

All the above relationships support partial ordering and satisfy the mathematical properties of transitivity and asymmetry. We say that a binary relationship R defined on a set N is transitive, if for any elements A, B, C from the set N such that element A is related to element B, and element B is related to element C, follows that element A is related to element C.

For example, the ordering defined by timestamps from the same clock is transitive. Thus, if event A happens earlier than event B and event B happens earlier than event C then event A happens earlier than event C. Relationships of aggregation and causality are also transitive. A binary relationship R defined on a set N is called asymmetric, if there exist elements A and B from the set N, such that element A is related to element B and element B is not related to element A.

2.3.4 Event Processing Languages

To describe a complex event, the customer is interested in different types of event processing languages (EPL). Usually, these languages provide means to identify a sought-for event sequence using some form of a pattern, and a set of rules, which define how to react when the event sequence is found.

Because CEP as a technology emerged from different areas of computer science (such as active database systems, artificial intelligence, event simulation, etc.),

currently there are many different event query languages that use quite different paradigms and approaches and significantly differ in the level of expressiveness.

Due to the lack of common terminology in CEP, there are also different classifications of EPLs. The following classification is taken from (Eck08). The author distinguishes the following types of event processing languages:

- **Composition-operator-based**: The main idea of this group of languages is to define simple events and a set of operators that are recursively defined on them. The result of applying operators is, in its own turn, a complex event. By analogy with mathematics such languages are also called event algebras, even though not all natural algebra properties apply to composition-operator-based (COB) languages (for example, associativity does not hold). Conceptually, a typical COB language interprets events from all input sources as a single stream of events of different types (Bui09). The language defines a term of simple event query as a query that allows specifying events of a particular type, for example message received or message sent. The query takes all incoming events as input and produces a stream of events of a specified type as output. Even though some languages of this group allow specifying not only a type of the event, but also some arbitrary data fields, COB languages mostly focus on actual events as happenings rather than on the associated data. Further in the text, we will denote simple queries by capital letters (such as A, B, etc.) and the corresponding events by lowercase letters (a, b, etc.).

- **Data Stream**: Data stream languages define data streams as unbounded ordered sequences of tuples (see (Eck08)). Each tuple represents a single event. In order to be able to use SQL-like queries, data streams have to be bounded and converted to some form of relations, at least on the conceptual level. The main idea behind data stream languages can be defined as follows: At each point in time T the input data streams are converted into a set of relations, then a regular relational query is applied to them, and afterwards the result is converted back to the output streams. Consequently, data stream languages define three types of operators: stream-to-relation, relation-to relation, and relation-to-stream operators. Stream-to-relation

operators produce a relation from a stream at some point of time T. Typically, the result of such an operator is a set of tuples for a predefined period of time or of predefined volume. Basically, stream-to-relation operators define either a time or tuple based window of relevant to the query events.

- **Production Rule Languages**: Production rule languages were not specifically created for event querying, but provide a very convenient way to do that. This group includes such languages as OPS (FM77), IBM ILOG (MD05), Jess (FH04), Drools (Bro09) and others. The main idea of these languages is to use a condition-action rule of the form

  ```
  WHEN condition THEN action
  ```

  to express event queries. The semantics of this statement is that as soon as the condition becomes true the corresponding action is executed. The condition is checked against a set of facts, which is called the working memory. The facts could be inserted and removed from the working memory. Usually, representation of working memory and facts strongly depends on the general purpose programming language around which the production rule engine is build (such as Lisp, Java, etc.). There are no restrictions on the actions being called. They can be custom method calls or procedures. They also can change the working memory, inserting or removing facts. The CEP functionality can be implemented using production rule languages by inserting the corresponding fact in the working memory on each new event and writing event queries as a condition over these facts. As soon as a new event inserted, the evaluation of all production rules have to be triggered.

Interestingly, the programming model in section 3.4 will make use of production rules languages, since we believe that the

```
WHEN condition THEN action
```

is very straight forward and understandable idea for a user. The power of the window concept is otherwise very helpful for reasoning over data streams.

2.3.5 Temporal Aspects in Event Data Processing

Data stream languages define data streams as unbounded ordered sequences of tuples (see (Eck08)). Each tuple represents a single event. In order to be able to perform queries over such sequences, data streams have to be bounded, at least on the conceptual level. The main idea behind data stream languages can be defined as follows: At each point in time T the input data streams are converted into a set of relations, then regular queries can be applied to them, and afterwards the result is converted back to the output streams.

The concept of windows helps to bound the sequence of tuples. This helps to apply techniques which usually work only on bounded data sets to unbounded data streams. In general windowing is a costly way since it keeps the sequence as state to enable processing over the bounded state.

There are several types of time windows, such as (see Figure 2.5):

- **Unbounded window**: the resulting relation R at time T contains all tuples received till time T. This type of the window is very resource intensive and not often used in real life applications.
- **Sliding window**: the resulting relation R at time T contains all tuples within a window of a specified duration d. The start of the window is moved forward each single time unit or entrance of new events (K ==1).
- **Hopping window**: the resulting relation R at time T contains all tuples within a window of a specified duration d. The start of the window is moved forward each K units.
- **Tumbling window**: this is a special case of a sliding window, when the size of the window is equal to its hop size, i.e. d == K.
- **User defined window** (frame): the resulting relation R at time T contains all tuples within a time points f(T) and g(T), where f and g are user defined functions. This is also called *frame.*The user function triggers the closing of the window and opening of the successor window.

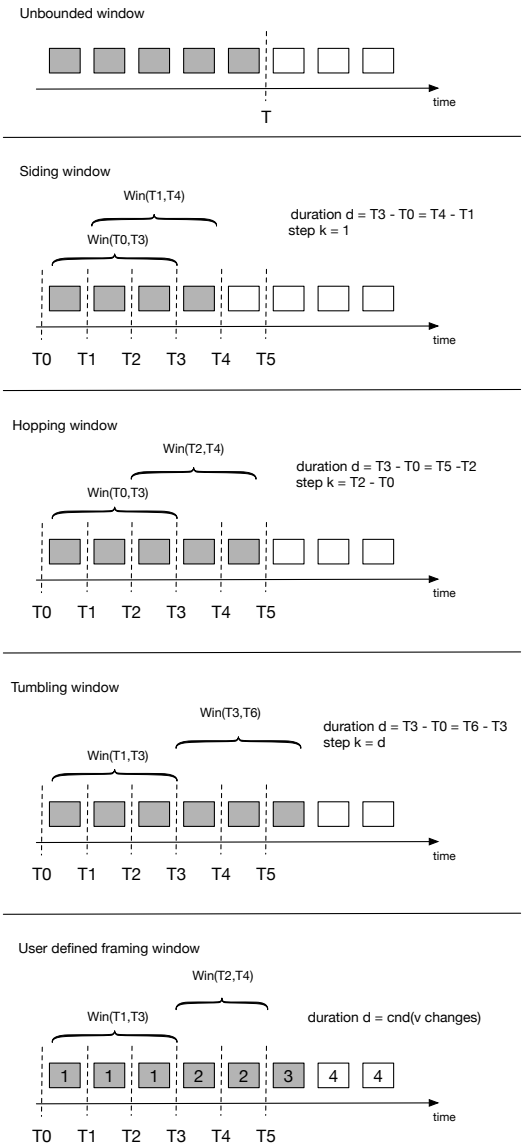


Figure 2.5: Windows for temporal data processing

2.4 CEP Tools

There are various CEP tools available on the market. An elaborated list and tool description can be found in the appendix (A). We conducted the survey within the KAP project. There are different features offered by the tools. In this section, we discuss the features offered by the different CEP tools. First, we clarify the features and then we provide a table summarising which products support which features.

We broadly devise the features into *language* features and *execution* features. The formers deal with how the tool users specify how the events should be processed. The latters are related to how the system behaves at runtime.

2.4.1 Language features

- **Type**: We can distinguish 2 broad categories of language the detecting (det) or pattern-based languages and the transforming languages. The detecting languages define detecting rules by separately specifying the firing conditions and the actions to be taken when such condition holds. Conversely, the transforming languages specify one or more operations that process the input flows of events to produce one or more output flows. They can be further refined into the declarative (decl) languages for which the processing rules are defined in term of results, and the imperative (imp) languages for which the processing rules are defined in an imperative way, letting the user choose a plan of primitive operators.

- **Time mode**: Relationship between the information items flowing into the CEP engine and the passing of time. We distinguish the following 3 modes:

  - Stream-only (stream): the only language construct dealing with time is the windowing
  - Absolute (abs): an absolute timestamp is available in language construct
  - Interval (interv): 2 timestamps, start and end

- **Single item operators**: The three operations considered are the selection, the projection (removing event fields) and the mapping (creating new fields based on the original fields).

- **Logic operators**: The logic operators apply only for detection-based languages. They are used to define rules that combine the detection of several events. The operators are conjunction (satisfied if all events are detected), disjunction (satisfied if one of the events is detected) and negation (satisfied when the event is not detected).

- **Sequence**: The sequence operator is similar to the logic operators except that the ordering of the events is considered rather than just the presence or absence of the events.

- **Iteration**: The iterations are repetitions of events for which one can specified the minimum and the maximum number of repetitions.

- **Windows**: The windows are a construct to group events into buckets that can be processed individually and for which aggregate functions are available such as COUNT. The first type is the fixed window for which the start and the end times are specified. The landmark windows specify only the start time and process continually the new incoming events. Then, we distinguish windows based on event count and on time. The count sliding windows move by one event at a time, removing from the current windows the oldest event and adding the newly arrived event. The count tumbling windows proceed by removing all the events in the current windows and then filling the new windows with new events. Finally, the count hopping windows is the generalisation of the two preceding windows, discarding an arbitrary number of events. The time windows are similar to the count windows except that they act on time rather the element counts. Therefore, the time sliding move by a unit of time, the time tumbling windows move by time step corresponding to the window, and the time hopping move by an arbitrary amount of time.

- **Flow management**: The flow management operators operate on the flows of events directly and correspond to their relational equivalents.
- **Flow creation**: The flow creation refers to the possibility to create a new flow of events.
- **User defined function (UDF)**: This refers to the possibility to define functions that can then be called by the operators.
- **User defined aggregator (UDA)**: As an extension of the UDF, the UDA refers to the possibility to create aggregate functions used with the grouping of the events into windows.
- **User defined operator (UDO)**: The latest extension considered is the possibility for the user to create its own custom operators.
- **Statistics**: This refers to the support in the language for statistical function, at least correlation and linear regression.
- **Probability**: Finally, the probability feature refers to the support by the language for uncertainty regarding event occurrences.

Table in Figure A.1 (see Appendix A) provides a summary of the language features supported by the different tools reviewed.

2.4.2 Execution Features

The execution features refer to features concerning the runtime operations of the tools. We have selected the following features.

- **Deployment model**: The deployment model refers to the way different engine instances possibly collaborate. The centralised model refer to one unique server, the network model refer to instances collaborating over the network, and finally, the cluster model refers to the collaboration between instances over high-bandwidth connections. Note that although a tool is classified as centralised, this does not prevent the user to have different

instances collaborating by building the proper input and output adapters. It only means that there is not intrinsic support for deploying queries over different machines.

- **Manageability**: Manageability refers to the possibility to start and stop queries dynamically.
- **Monitoring**: Monitoring refers to the capacity to monitor live the resource consumption of the engine instances by queries.
- **Capacity management**: Capacity management, also called load shedding, refers to the ability of the engine to select events to drop in order to avoid overloading of the engine.
- **Debugger**: The debugger feature refer to the ability to track the events as they flow through the operators.
- **High Availability**: High availability refers to the support of failure of engine instances, in which case another instance take ownership, preventing loss of analysis.

Table in Figure 2.6 shows the execution features supported by the reviewed tools. The data presented in the table is extracted from current data sheets and web pages. For some research tools the data is extracted from research papers. Thus, this information might be older then one year. The license type for the tools is also specified in the table.

2.5 Event-based Interaction Pattern

Looking at a system from a higher level, you can abstract away the details of how communication occurs and focus on what an interaction is trying to accomplish and how. Software systems can be considered societies of communicating and cooperating processes (as described in (Fai06)). Process interactions are similar to conversations between people. The expression interaction dynamics is used to describe the ways software processes interact with each other over time, with particular interest in the following areas:

Tools	License	Deployment model	manageability	monitoring	capacity management	debugger	High availability
AMIT	commercial	Centralized	X				X
Borealis	open source	Clustered	X	X	X		X
Cayuga	open source	Centralized	X			X	
Esper/Nesper	open source + commercial	Clustered	X	X			X
Infosphere Streams	commercial	Clustered	X	X	X	X	X
Oracle CEP	commercial	Clustered	X	X	X	X	X
PADRES	open source	Networked	X	X	X		X
Rulecore CEP server	commercial	Centralized	X				
SASE	open source	Centralized					
Smart Enterprise Platform	commercial	Networked	X	X			X
STREAM	open source	Centralized					
Stream Mill	open source	Centralized	X				
Streambase	commercial	Clustered	X	X		X	X
Streaminsight	commercial	Centralized	X	X		X	
Sybase Aleri Streaming Platform	commercial	Clustered	X	X		X	X
TelegraphCQ	open source	Clustered					
Websphere Business Events	commercial	Centralized	X	X		X	X

Figure 2.6: Execution features supported by CEP tools

- **Roles**: Which process is giving information to the other? The word information is used here to denote both commands and data. One process might assume the role of caller or data provider with respect to the other.

- **Control**: Which process is in charge of the interaction? Which one is responsible for initiating it? Can the interaction be aborted? If so, by which process? Is one of the processes responsible for monitoring the progress of the interaction? Which process decides when the interaction terminates?

- **Timing**: When sending messages, can the sender wait forever for a response? Can the sender continue with other work while waiting for a response? Does the sender require a response within a certain time frame? Can the receiver accept other messages while processing a prior one?

- **Flow**: Is information sent in a single exchange, or it is broken down into

an iteration of smaller ones? If an iterative flow is used, how is the end of the iteration signalled?

Looking at different kinds of software systems, it becomes apparent that there are recurring ways in which processes interact in terms of communication mechanics, assumptions made by the communicating parties, goals of the sender, overall timing, and so on. This can be called interaction patterns.

2.5.1 Message Exchange Pattern

One of the simplest ways of classifying an interaction is based on which way information flows in relationship to the party starting the interaction. If one party gives information to another party, the information is said to be pushed. If one party requests information from another party, the information is said to be pulled.

In push-pull models, one do not worry about lower-level interaction details, such as whether exchanges are synchronous or asynchronous. What matters the most is the overall direction of information flow relative to the party starting the conversation. In the next section we will look more detailed into push and pull interactions.

2.5.2 The Pull Model

The purpose of a pull interaction is for one party to obtain information from another. Normally, the caller is named client and the responder is called server. As the terms indicating this is baseline of the classical client-server architectures. In a pull interaction, the interrogator sends a request to the respondent, which replies with a response.

An example would be that a service S_1 needs to continuously monitor the status of a number of other services S_i. Status changes can only be handled at a certain rate, decided by S_1 (see figure 2.7)

In the client-server or pull model we can observe both synchronous (see figure 2.8a) and asynchronous calls (see figure 2.8b)

Pull interactions are effective when the following two things apply:

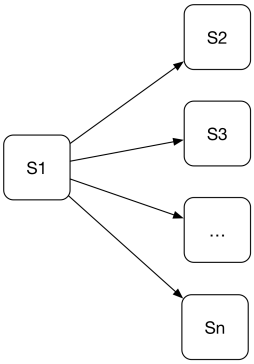


Figure 2.7: Pull model example

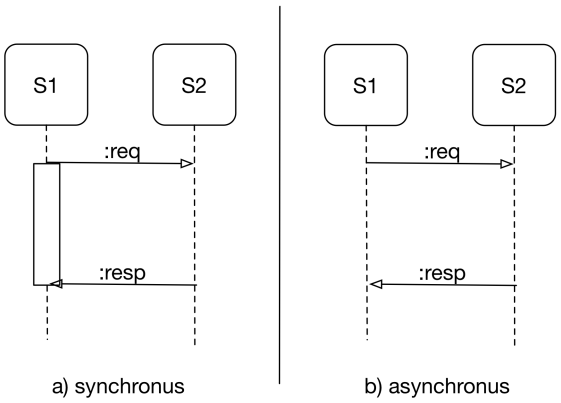


Figure 2.8: Synchronous and asynchrounous interaction

1. The interrogator (here: S_1) needs to control when information is retrieved.

2. The interrogator (here: S_1) only needs information at a given time.

The interrogators request can convey the type of information needed. The respondents response supplies the information. If the interrogator uses a method call to query for information, the method can define parameters to hold the response information. If the interrogator uses a message for the query, a separate interaction is required for the response. The response is not considered a push interaction, because the interrogator determined its timing. Pull interactions are particularly effective when the interrogator only needs information at specific and infrequent times. It would be wasteful for the respondent to push the information using change notifications, especially if changes occurred frequently.

Pull interactions are very common for two reasons:

1. They are simple to implement, because both the interrogator and the respondent can be synchronous. A simple method call is sufficient.

2. They apply to the common situation in which status changes in the respondent do not require the interrogator to react immediately: The interrogator will discover the change the next time it requests status.

2.5.3 The Push Model

The purpose of a push interaction is for one party to deliver information and/or execution control to another. There are different terms used for the one initiating the interaction. Here we can use the term publisher to designate the party starting the interaction, and the word subscriber for the party with whom the publisher interacts. In a push interaction, the publisher sends a command to the subscriber, as shown in figure 2.9.

A command can include data. A push interaction is an imperative interaction. Push interactions are effective when the following two things apply:

1. The publisher acts as a controller of the subscriber.

2. The publisher decides when to issue commands to the subscriber.

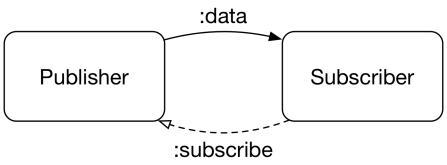


Figure 2.9: Publisher/ Subscriber

Push interactions form the basis of event-driven systems. Publishers are talkers, and subscribers are listeners. When something occurs to the publisher, notifications are sent to available subscribers to inform them of the event. Using push interactions to propagate changes through a system is particularly efficient when the talker needs to send the same command to a large number of listeners. If you have to implement the system without using push interactions, the alternative would require each listener to poll the talker periodically, waiting for changes to occur. A significant amount of processing time would be wasted in polling loops. Using a push interaction pattern frees the listeners from devoting resources to monitor the talker. While the talker has no new data, the listener can use its own resources to do more useful things than sitting in a polling loop. Due to their efficiency, push interactions are a good solution in real-time systems. When changes occur at very high speed, you might use a single notification to describe multiple changes at once. There are numerous examples of the push pattern in everyday life and in computing.

There are basically two different kinds of push interaction (see figure 2.10):

- **Fire and forget**: Calls made without the expectation of a completion, thus is interaction is not reliable.
- **Pushed Feedback**: Interaction for asynchronous blind interaction, with feedback pushed to publisher.

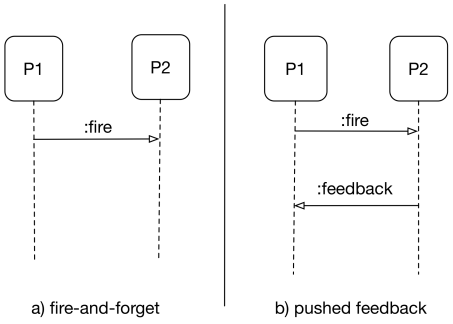


Figure 2.10: Push models

2.6 The IMC-AESOP CEP Service

The IMC-AESOP project (`http://www.imc-aesop.eu`) envisions an infrastructure that goes well beyond existing approaches. It will enable cross-layer service-oriented collaboration not only at horizontal level, e.g. among cooperating devices and systems, but also at vertical level between systems located at different levels of a Plant-Wide System (PWS) enterprise architecture. Focusing on collaboration and taking advantage of the capabilities of cooperating objects, poses a challenging but also very promising change on the way future plants will operate, as well as to the way we design software and model their interactions.

In process industry systems become nowadays larger and more complex than before. They often consist of heterogeneous, distributed components, which might themselves be complex subsystems. In order to cope with this increasing complexity hardware independent, asynchronous communication methods have to be employed. Such a flexible mechanism can be built based on the concepts of services (KCJ+10) and (CK09). Data processing in IMC-AESOP is focused on events and (complex) event processing (CEP).

Normally, complex events are created by abstracting from low-level events, which can be thought of as sensor readings. The processing of events is expressed within a specific language in terms of rules. Unfortunately, the set of features and the way to express the rules differ from platform to platform.

In IMC-AESOP some efforts were looking into concepts for aligning CEP concepts for process industries in terms of distribution and processing on devices.

As a result we were looking into the two different approaches:

- **CEP as a service**: In this work a service can either be realised as a service running locally on a server or the same concept still holds for a service running in the cloud and on top of cloud technologies. It is meant to be a general-purpose service, which offers *CEP* as a feature. CEP systems are designed and implemented in a way to be able to handle more than a million events per second. This data rate should be good enough to enable CEP as a service either on premises or in the cloud. The limiting factor here is most probably the network in terms of latency and bandwidth. This service is realised as an extended version of an event broker. While the event broker is a consumer and provider the CEP service also provides and API to deploy queries. Thus, the CEP service has an input layer excepting incoming events from publisher and an output layer sending events to registered consumers. The consumers can register for specific topics (see Figure 2.11). The queries inside the service can register to specific topics. Thus, events can be forwarded to queries for a topic. This happens via a *topicBinding*. The *topicBinding* is also used to map the output of a query to registered consumer.

- **Embedded CEP**: This is a concept of a lightweight CEP using concurrent reactive objects (CRO) model guarantying execution of CEP queries in an efficient and predictable manner on resource constrained platforms and offering a low-overhead real-time scheduling. As described in (LPM13) and (PLM12), there have been a lot of efforts on processing data on resource-constrained devices, typically on sensor nodes in wireless sensor networks, see e.g. SwissQM (MAK07) or TinyDB (MFHH12). These approaches however differ from this one in that nodes actively query for data, whereas in our approach nodes react on external events. Event-based programming is a core concept behind Contiki - a tiny operating system targeted for wireless sensor nodes (DGV04). The work on embedded CEP is providing a scheme to implement CEDR queries (BGAH06) using the CRO model. Semantic similarities between the two make it possible to design a fully automatic translation. The technique proposed in the paper is based on

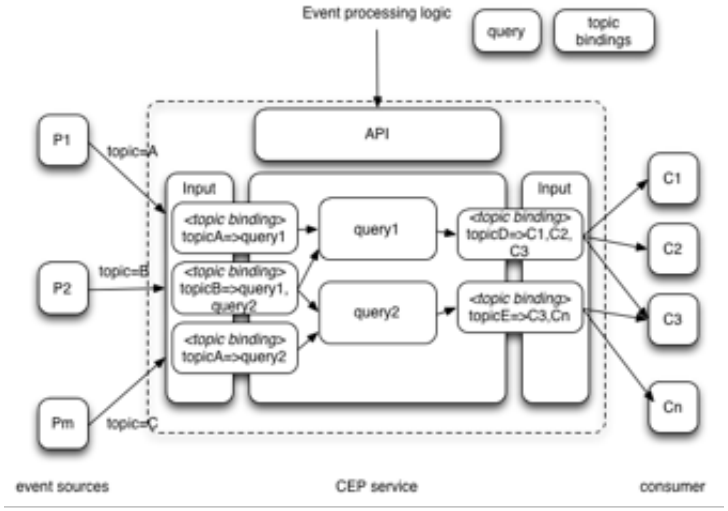


Figure 2.11: CEP as a service

compile-time translation, whose aim is to create tailored highly efficient and predictable CEP system for resource-constrained platforms.

The work in IMC-AESOP is showing some early concepts for having some data processing running as a service and put some data processing logic on a smaller device. Although the concepts do not provide generic, holistic approach end-2-end it definitely highlights the demands for such an approach.

2.7 KAP Data Processing Framework

The main objective of the KAP project (`http://www.kap-project.eu`) is to develop standards and a framework that supports sustainable manufacturing by increasing transparency and supporting informed decision making. At the shop-floor level, energy standards will ensure that energy is managed with the same level of attention as other materials in the manufacturing process. These standards will be defined to be compatible with existing production standards and will support the development of an energy management system (EMS). The inclusion of an EMS is an important step in achieving the goals of sustainable manufacturing. Other existing factory systems will continue to monitor and measure

sensor data and feed into data storage systems (DSS) and business intelligence (BI) tools at higher levels. However above these, new production performance indicators (PPIs) will be defined. By applying CEP models to past and current PPI levels, future outcomes can be predicted. This should reduce the effects of human bias in prediction while at the same time removing the need to manually revise prediction models. This method ensures continuously adjusted accuracy and creates a self-improving operational control system.

Within the KAP project an end-to-end architecture was developed to provide seamless processing of PPIs at the manufacturing environment. To enable scalability, adaptability, and availability in stable and unstable environments as given in industrial manufacturing scenarios, PPI events have to be processed and aggregated as close as possible to their data sources. Thus, one focus lies on the design and realisation of a reference infrastructure for distributed event and stream processing in heterogeneous networks that integrates existing as well as novel approaches on modelling stream networks and publish/subscribe-mechanisms for (content-based) event routing. Particularly, queries and operators have to be deployed and managed according to constraints from devices and data transfer media as well as data selectivity inside complex event processing queries. Besides mechanisms for efficient processing of raw event data streams like aggregation, compression and filtering, complex analyses will be developed including but not limited to online stream mining and (approximate) similarity matching.

The end-to-end architecture provides a concept for a generic processing engine to support distributed CEP queries based on the architecture specified within KAP (see Figure 2.12).

The data processed by the system is retrieved from various sources such as sensors, databases or other systems. The integration of this data is outside of the architecture. From the data sources at the shop-floor to the data consuming systems such as user interfaces, manufacturing execution systems (MES) or enterprise resource planning (ERP) these layers are:

1. **Data Integration**: The data integration layer is used to retrieve, transform and translate data from arbitrary systems and sensors so that it can be

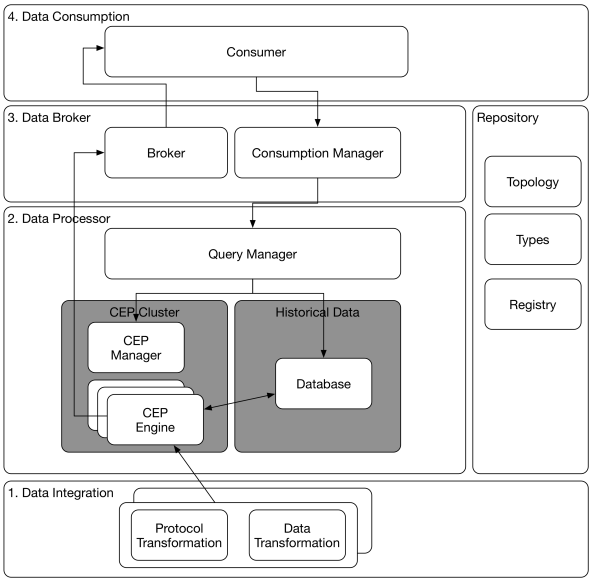


Figure 2.12: KAP architecture

processed by the KAP system. The integration can be implemented in multiple ways, e.g., using proprietary solutions or any sophisticated integration middleware.

2. **Data Processing**: All processing of shop-floor data is done within the data processing layer. The processed data can either be current real-time data of the shop-floor systems, previously recorded historical data or a combination of both.

3. **Data Broker**: Consumers can access all data in the system, configure data processing and subscribe to result- and raw-data-streams using the data broker layer. The purpose of this layer is to encapsulate the complexity of the underlying system and provide easy and fast access to it.

4. **Data Consumption**: The data consumption layer contains any software component configuring and consuming data processed by the KAP system. This can be any existing system, such as monitoring cockpits, resource planning or controlling systems.

The KAP project was focusing on shop-floor systems. The overall architecture was targeting shop-floor operators to define and consume PPIs. The CEP cluster was already trying to address the demand of distributed data processing but very limited by the underlying technology and the restrictions required from the architecture in terms of distribution concepts.

2.8 Summary

There are several approaches and technologies for SOA service selection, architectures for highly distributed architectures, and handling big data with speed. Unfortunately, most approaches are limited to only one aspect. There is at least for our knowledge no approach, which covers end-2-end SOA based scenarios, which are highly distributed and require fast processing of data. We have looked into a new family of scenarios, which are in the field of the Internet for Things. But, whenever we are looking into SmartGrid, SmartFactory or SmartHome scenarios we have exactly a huge number of data sources, combined with a different set of services (and thus with different interaction pattern), and with requirements around handling data in a smart way.

We have identified a list of requirements (1.2) for fast processing of data in a distributed fashion and the need to integrate heterogeneous data sources. By analysing current related research work against the requirements, we find they lack integrating approaches end-2-end. Moreover, most works are scoping very much on aspects, which makes it hard to cover the full spectrum of the requirements.

The involvement in the KAP and IMC-AESOP project assists to develop results for fast data processing and concepts for distributed data processing for industry scenarios. From the next chapter, we will start to illustrate the solution being contributed to the fast and distributed data processing for hyper-scale systems field.

Chapter 3

Formal Model for Temporal, Fast Data Processing

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."

– Lewis Carroll, *Alice in Wonderland*

This chapter is focusing on data processing. It describes a way to handle data and process data in an efficient and fast way. We will introduce a model based on ITL which enables processing of time triggered or time dependent data.

In general the model enables us to reason about temporal data. In particular, the model will integrate services with time dependent offerings. Normally, services offer data, either functional or non-functional data. We will introduce a way to divide these general data or properties into more specific service offerings in terms of static and time dependent data.

We are also introducing a concept (*policies*) based on event-condition-action paradigm so that we can express rules over continuous stream of data to trigger actions. Therefore, we are defining a data stream model and combine it with the notion of time.

However, the proposed model enables to express *policies* in the form of a *source-condition-action* tuple by using *operators*. This approach is not restricting us only on time dependent data. We can use it for any kind of sequential ordered data. This can either be ordered by time or any other kind of sequence.

Finally, we explain how these policies can be validated against ITL logic.

3.1 Time Dependent Service Offering

Properties of services are considered to be non-functional or functional. Such properties are used for service selection or context-based service discovery. The available approaches are based on the fact that properties are pulled from service repositories (that is from service metadata) or possibly from the services directly before the algorithm determines the most relevant service for a given context. Repositories are useful for static data and polling services directly works if a small number of properties of a small number of services is of interest. We believe that there is an emergent need to provide methods to enable the continuous evaluation of functional and non-functional properties especially in the case where the number of services is high.

We define static properties *ps* as constant over time, such as a location of a printer, the vendor of a printing machine, or the number of a taxi etc.

Dynamic properties *pd(t)* are changing over time. Using these, we define non-functional properties NFP as a tuple of static properties and dynamic properties:

$$NFP(t) = \langle ps, pd(t) \rangle$$

For the fleet management scenario the schema of the non-functional properties might look as follows:

```
<NFProperties>
  <Static>
        <TaxiId type="xs:string"/>
  </Static>
```

```
  <Dynamic>
        <GEOLocation id="x">
          <Longitude type="xs:int"/>
          <Latitude type="xs:int"/>
        </GEOLocation>
        <PassengerNumber type="xs:int"/>
  </Dynamic>
</NFProperties>
```

This presents the static data schema; like a snapshot in time. Temporal aspects are covered by events and therefore we would see different data at different points in time.

As a side note, we can define a time dependent schema for non-functional properties; for service offerings we can do the same for functional properties. In general, we are not restricted to non-functional properties here.

3.2 Event Policies

Policies refer to obligations placed on services to actively communicate *dynamic* information, with respect to a given data-schema, triggered by events and time. Informally this means that a policy defines the granularity over time at which data is pushed up the service chain to aggregating services and end-users.

In this thesis the policies are modelled similar to the well-understood Event-Condition-Action paradigm (For90, TLDS08, UBJ+03). However, the novelty of the policies used in this thesis is that they use temporal conditions that describe the *condition* between two consecutive actions that push data to aggregating services, rather than defining conditions on the system state. The advantage of this approach, compared to existing temporal conditions (JCS+06, JCSZ12), is that the condition bridges between two events, thus does not require the storage of large amounts of historical data.

Informally a policy is a set of rules of the following structure:

```
<Policy> <!-- send to Service -->
  <Rule>
    <Source>...</Source>
    <Event>...</Event>
    <Condition>...</Condition>
    <Action>...</Action>
  </Rule>
  <Rule> ... </Rule>
</Policy>
```

The `<Source>` of a rule is a continuous list of data on which the `<Action>` of the rule is invoked if the rule is triggered. The `<Event>` of a rule is an event descriptor that determines when the `<Condition>` of the rule is evaluated. The descriptor is a predicate build from primitive events (e.g. a GPS-Update) that are domain dependent and defined in the service description. Conceptually the event descriptor describes an abstraction of the event trace over which the `<Condition>` is evaluated. The `<Condition>` describes the distance between events that are communicated upstream to aggregating services as a temporal formula. The syntax that is used is an XML representation of Interval Temporal Logic formulae that is described in section 2.2. This can be represented as shown in figure 3.1.

To provide a more formal description we define the following constructs. An event policy is a set of rules $pol_s = \{r_0, \ldots, r_n\}$. Each rule $r \in pol_s$ has four components:

- **Source** (src_r): A source is defined as a list of continuous data flow. This flow of data is either finite or infinite. The continuous data flow is called a *stream* σ_s and produces data of the same type T.
- **Event** (evt_r): An event is the data construct of a stream. Events are of a specific type T. An event is defined as a tuple $\langle se, ts, p \rangle$ with se the service producing the event, ts is the time-stamp of the occurrence of the event and p the payload.

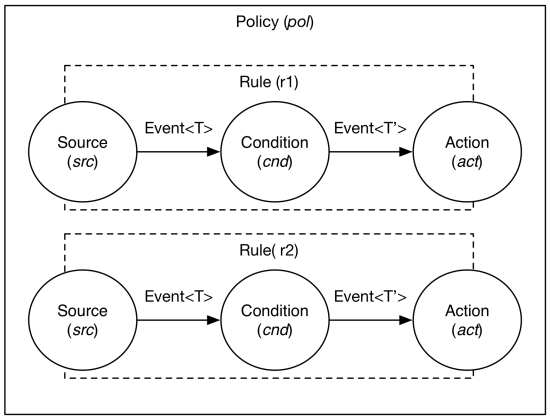


Figure 3.1: Event policy as source, condition, action

- **Condition** (cnd_r): A temporal condition bridging between two consecutive executions of the *action*. A condition of one or two stream of events and produces exactly one output stream of events. The condition can either do a simple transformation, filters, or join events from the input. The input stream has events of type *T*. The output of condition has events of type *T'*.
- **Action** (act_r): The action to be performed on the target.

3.3 Temporal Data Streams

Let each service $s \in Services$ be defined over a continuous stream σ_s of events $e_i \in Events_s$, observed by the service s. This is modelled by representing σ_s as an ITL interval and $Event_s$ as a set of propositional state variables that indicate the occurrence of events (recall that state variables can change their value from state to state). This model allows for the concurrent occurrence of events, e.g. $e_i \wedge e_j$ $(i \neq j)$, and only captures the sequence of events, rather than their absolute timing. The creation time of the event is stored explicitly as part of the event tuple and can be referred to in the conditions of policy rules.

As described an event is described as a tuple $\langle se, ts, p \rangle$, denoting the service se creating the event, the time-stamp when the event was created ts (based on the clock of s) and an optional payload p. We use the notation $e.se, e.ts$ and $e.p$

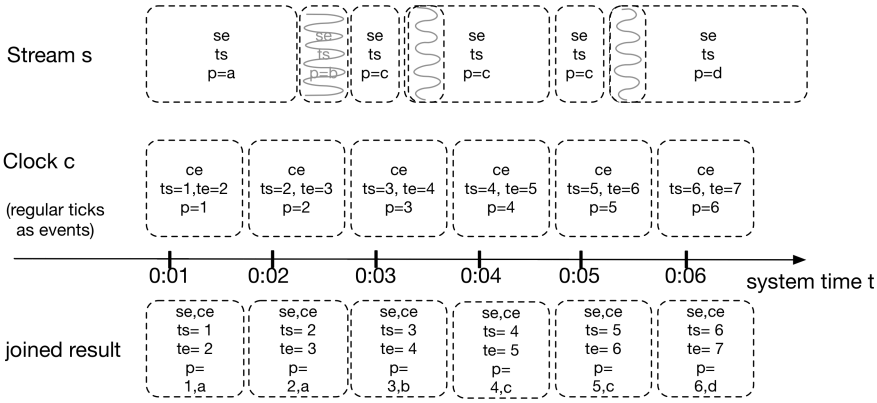


Figure 3.2: Temporal streams

when referring to a specific element of an event tuple e (see stream s in figure 3.2).

Each event e in σ_s sets the state within the stream. This means that the state holds until there is another event changing the state. The stream is in this sense an ordered sequence of states. The ordering is given by the time of occurrence of the particular event.

Sometimes it is required to have a well-defined definition of time in system. It is like a tick or a heartbeat. It is either possible to inject this heartbeat already into the event from the beginning or to have the option to specify a clock (3.2) following the same principal as the "normal" events. Thus, we do have the option by simply changing the clock stream to influence the time progress within the system. By joining a stream s_i with a clock stream c we do have the option to be very flexible with time in the system without changing the condition logic.

We are introducing another stream σ_c which we call a *clock c*. The *clock* injects events into the system. A timer given by the system clock triggers the injection of events. We are extending the event definition by an end-time te. Thus, an event is defined as a tuple $\langle ce, ts, te, p \rangle$. The end-time defines the end of the event. We define that the end-time te is the right border and does not belong to the event. The time of the event is an open interval $[ts, te[$.

The clock enables us to introduce time into the system without changing the original definition of the system. Furthermore, the timestamps ts and te

do not have to be system clock. We can set ts and te to values we want. These timestamps are following *application time* (see (BGAH06)). This is useful because we can define rules on application time and do not depend on system time alone. In this way, injecting events on the clock stream does not influence the rule.

Since the clock stream behaves like a normal stream we can join it with another stream. As a result the resulting stream is time stamped by application time so that we can have proper temporal rules over this stream. Introducing start- and end-time has some impact on the behaviour. In figure 3.2 the event with payload a in stream s correlates with the event at time 0:01 in the clock stream. Therefore, if we join both events (it is possible since the state in s has the payload a and the event in the clock stream has the payload 1) we are getting an event with a combined payload $\langle 1, a \rangle$, $ts = 1$ and $te = 2$. At time t=0:02 there is another clock event triggered with $ts = 2$ and $te = 3$. At this time the state of stream s is still having the payload a. Therefore the resulting event is $\langle 2, a \rangle$, $ts = 2$ and $te = 3$. The se value is in our example extended to a combined se, ce value to show that the event is a result of a join of two streams. Stream s is now triggering another event with payload b. Unfortunately, this event is ignored in the resulting streams since this event is triggered at a non-valid timestamp. Actually, it is below the sampling rate. In reality, it is impossible to set a proper timestamp for this event in the resulting stream since application time has the resolution of one tick in our example and we do not want to introduce values below a tick. To overcome the problem the event injection rate could be increased. Since the injection runs with system time we could easily increase the injection rate. Here, the advantage of separating application time and system time is obvious.

3.4 Programming Model for Data Streams

To program event policies we consider a functional programming model as very relevant. As discussed in section 2.3 there are certain CEP programming paradigms to enable event processing already. Unfortunately, there is no standard right now. We are providing a more general programming model, which will fit to most of available CEP engines and implementations. However, functional programming

is providing already a very promising foundation and has proven that the way in which list and sequences are handled using lazy evaluation is making a lot of sense.

In general, there is a concept of an operator. An operator o can have one or more input streams σ_i with events e_i of type T_i. An operator o is producing exactly one output stream σ_o of events e_o of type T_o. Thus, the output of one operator can be used as an input of another operator. Therefore, we can compose a graph g of operators o_i. This graph g is called an expression exp and is equal to the condition cnd.

To compose operators we want to introduce the pipe symbol $|>$ so that we can write

$$o_1|> o_2.$$

This means that the output of o_1 is used as input of o_2. Since the source src produces a stream $sigma$ as well a source can be at the beginning of a pipe. We can write

$$src|> o_1|> o_2.$$

The same is true for the end of a pipe. An action act can have an input stream without producing an output stream. Thus, the full programming model looks like

$$src|> o_i|> act$$

which is equal to

$$src|> cnd|> act.$$

As a result the condition cnd can be written a s sequence of operators

$$o_1|> o_2|> o_i.$$

Finally, the full source-condition-action rule can be expressed as

$$src|> o_1|> o_2|> o_i|> act$$

3.4.1 Operator for Stream Models

We have explained how we can compose formally operators. In the following we define operators. Figure 3.3 presents an overview over the set of required operators. The following list provides a short description (for operator description we are using a Haskell syntax and implementation):

- **Pipe** $(|>)$: The pipe is used for composing operators left to right. It feels more natural to write a pipeline from source (left) to action (right). In Haskell a pipe is the inversion of the $ operator:

```
(|>) = flip \$
```

- **Stream** : A *stream* is defined as a data type. IN combination with the *cons* operator it produces a infinite stream of elements of type a:

```
data Stream a = Cons a (Stream a)
```

- **asStream** : This is a helper function which connects a source to a condition. It creates a continuous stream of data from a source. It can be seen as a protocol or behaviour and is very source specific. From an array of numbers the *asStream* operator creates a stream of numbers. In case of an finite array the resulting stream is finite as well. The type of the *AsStream* function looks as follows:

```
asStream :: [a] -> Stream a
asStream (x:xs) = Cons x (asStream xs)
asStream []     = error "infinite list expected"
```

- **action** : The *action* operator is used at the end of a rule. It connects the results of a rule with the outside world. Thus, it can be used to connect the result with actuators of the underlying system. If this is the physical system it can trigger switches. In addition it can be used to send data over the network. Finally, a special action implementation could even hold the last result and expose an endpoint to get the state pulled from here. We encapsulate all these external interaction with the well-known Haskell IO monad.

```
action :: Stream a -> IO ()


sink :: Stream a -> [a]
sink (Cons x xs) = x : sink xs


out :: Show a => Int -> Stream a -> IO ()
out n x = x |>  sink |> take n |> print
```

In combination with *asStream* and a simple *out* action we can formulate a pass-through pipeline:

```
asStream [1..] |> out 10
```

As an input we simply create an infinite stream from a list of type *Int*. The pass-through pipeline outputs the first 10 elements. This is handled by the *out* action:

```
[1,2,3,4,5,6,7,8,9,10]
```

- **filter** : Filters events based on a given function fn and produces a new stream of events. The *filter* operator does not change the type of the incoming events. The function fn returns a *bool* based on some logic.

```
filter :: (a -> Bool) -> Stream a -> Stream a
filter p ~(Cons x xs)
  | p x       = Cons x (filter p xs)
  | otherwise = filter p xs
```

  For example

```
asStream [1..] |> filter odd |> out 10
```

  results in

```
[1,3,5,7,9,11,13,15,17,19]
```

- **map** : Is used to map a function fn to each event. Since the function fn can change type of events the resulting stream has events of type b as a result of fn.

```
map :: (a -> b) -> Stream a -> Stream b
map f ~(Cons x xs) = Cons (f x) (map f xs)
```

  For example

```
asStream [1..] |> map (*2) |> out 10
```

results in

```
[2,4,6,8,10,12,14,16,18,20]
```

- **zip** : Joins two streams into one new stream. The first stream has events of type a, the second stream has events of type b and the resulting stream has events of joined type (a, b).

```
zip :: Stream a -> Stream b -> Stream (a,b)
zip ~(Cons x xs) ~(Cons y ys) = Cons (x,y) (zip xs ys)
```

For example

```
asStream [11..] |> zip (asStream[1..]) |> out 5
```

results in

```
[(1,11),(2,12),(3,13),(4,14),(5,15)]
```

- **state** : Triggers an output event when the value of the event has changed. The state operator keeps the state of the last value and compares it with current value. Comparing happens by using equality. Therefore elements of type a must derive *Eq*. If *Eq* returns false the current value is becoming the state. The *state* operator expects an initial value to start comparison with:

```
state :: Eq a => a -> Stream a -> Stream a
state i ~(Cons x ys)
```

```
      | i==x      = state x ys
      | otherwise = Cons x (state x ys)
```

For example

```
asStream (cycle [1,1,1,2,2,3,3]) |> state 0 |> out 10
```

The *cycle* operator creates an infinite list of the list [1,1,1,2,2,3,3]. Passing this as a stream through the *state* operator results in:

```
[1,2,3,1,2,3,1,2,3,1]
```

- **len** : The *len* operator is used to chunk a set of events together. The *len* operator takes a number n as argument and collects n events together into an array of length n of events and outputs this as a new event. This is useful aggregation of values, e.g. sum, average, min, max or other statistical functions. Normally, the len operator is followed by a map operator.

```
len :: Int -> Stream a -> Stream [a]
len n (Cons x ys) = Cons (x:xs) (len n zs)
    where
        (xs, zs) = span n ys

span :: Int -> Stream a -> ([a], Stream a)
span n (Cons x xs)
      | n == 0    = ([], xs)
      | n > 0     = let (a, rest) = span (n-2) xs
        in (x:a, rest)
```

```
    | otherwise = ([], Cons x xs)
```

For example

```
asStream [1..] |> len 3 |> out 5
```

results in

```
[[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15]]
```

- **win** (window) : The *win* operator is comparable with the len operator but the window operator takes two arguments: The size s of the window and the hop size h of the window. In this way two windows can overlap each other.

```
win :: Int -> Int -> Stream a -> Stream [a]
win s h (Cons x ys) = Cons (x:xs) (win s h hys)
    where
        (xs, zs)  = span (s+2) ys
        (hxs, hys) = span (h-1) ys
```

For example

```
asStream [1..] |> win 5 2 |> out 3
```

results in

```
[[1,2,3,4,5],[3,4,5,6,7],[5,6,7,8,9]]
```

Another example in combination with map

```
asStream [1..] |> win 5 2 |> map(\y -> minimum y) |> out 5
```

results in

```
[1,3,5,7,9,11,13,15,17,19]
```

3.5 Event Policy Validation

So far we have introduced a concept to process data from source to action as an event policy. This policy is formulated as a pipeline from $src| > cnd| > act$. We are considering an event stream as an infinite sequence of states. This allows us to handle time progression as a stream of states as well, where each new timestamp is represented by a new state in the sequence.

ITL is a logic which enables us to describe properties of intervals over a sequence of states. Policies in ITL can be executed. AnaTempura is an engine which can execute ITL policies. ITL works on finite streams. But ITL can have infinite time, e.g. *true* does not restrict the length of the interval. Indeed abbreviated constructs like *finite = true;false* and *infinite = not finite* are available. In our case, we want to process infinite streams, but we do not reason about infinite intervals. We were looking at behaviours that have unfolded to the current point in time and were acting on these. This means that we always have a finite time from start of process to *now*. In particular, we benefit from the fact that we can use windows to provide a bounded sequence to ITL over a infinite stream. We are scoping the processing over a finite set of values. By moving time and windows forward we can process the stream.

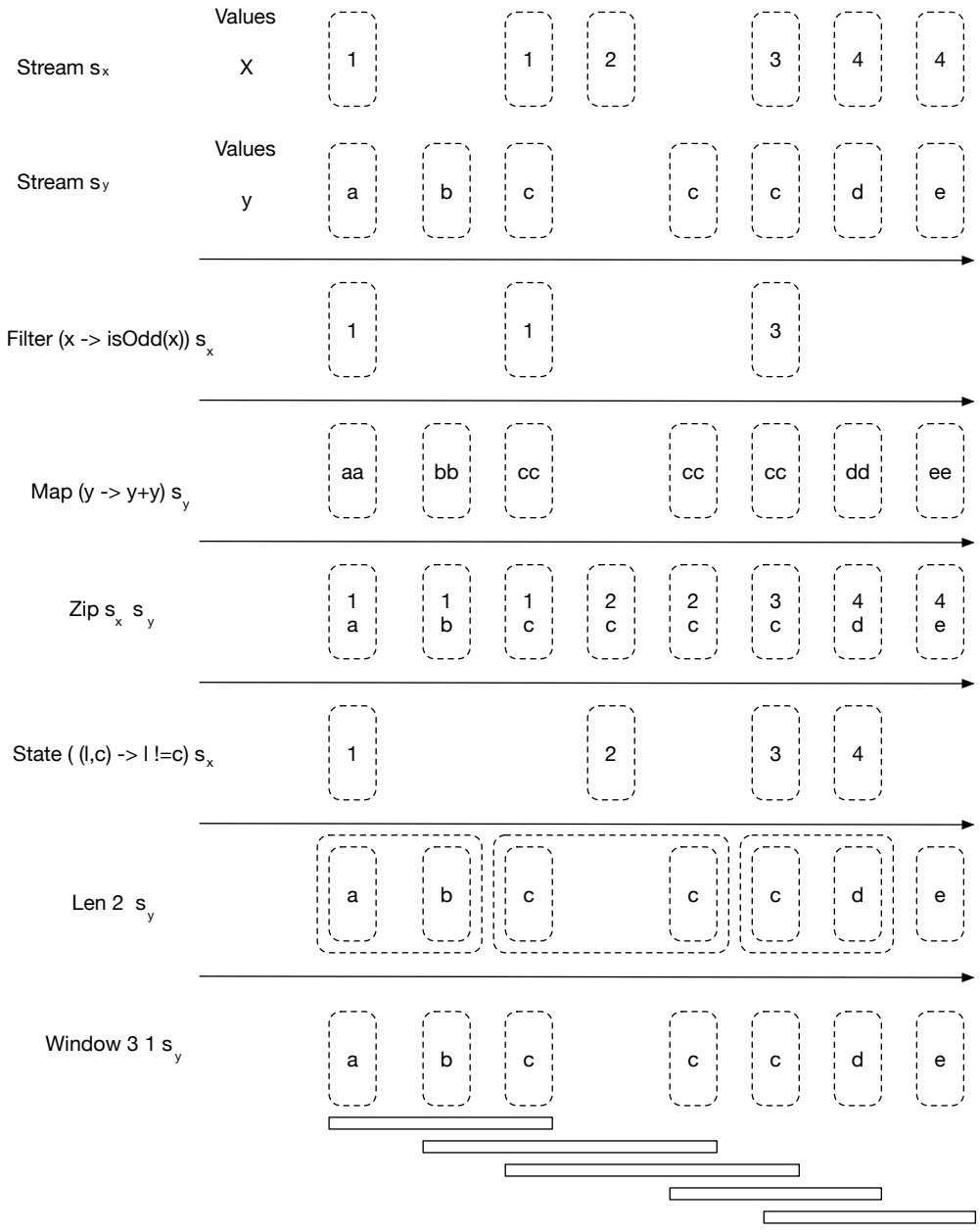


Figure 3.3: Operator overview

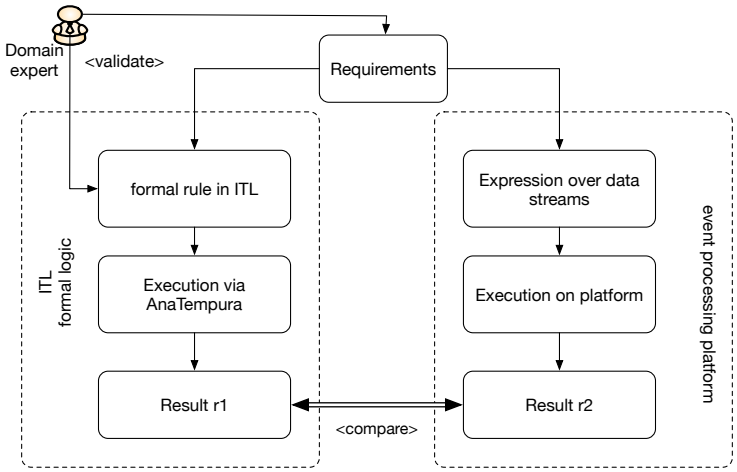


Figure 3.4: Policy validation approach

Using ITL to validate expressions over the $src| > cnd| > act$ (operator pipeline) paradigm, we have to find an ITL specification for a problem which can produce the same result as our operator pipeline. If such a problem can be formulated as an ITL policy we can execute this rule in AnaTempura. The abstracted ITL specification needs to be validated by a domain expert against the requirements. In this way , we can validate that the extracted specification reflects the requirements.

Thus, we can formulate the following assumption:

> We consider a operator pipeline expression as validated if an ITL policy can be defined such that (1) the policy yields the same outputs on all inputs as the pipeline when executed with AnaTempura and (2) the policy can be validated against the requirements for the pipeline, e.g. by a domain expert.

We use the ITL/AnaTempura path as the authoritative one. It is grounded and formally proven. The process can be described as follows (see Figure 3.4):

1. We express the rule as operator pipeline and run it.
2. We express an ITL policy and run it through AnaTempura.

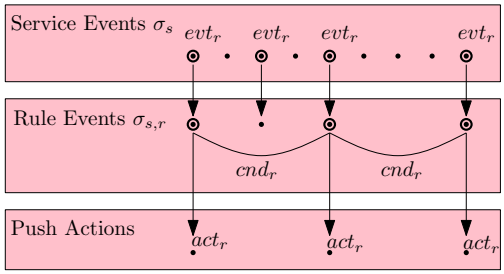


Figure 3.5: Policy Rule Evaluation

3. Domain expert reviews and validate the ITL policy against requirements
4. We compare both results.
5. If the results are equal for given input, we can consider the operator pipeline as validated.

Evaluating the policy pol_s of the service s against this interval is a two stage process in ITL.

3.5.1 Stage 1

First, for every rule $r \in pol_s$ an abstraction of the interval σ_s is generated based on the *Event* trigger evt_r of the rule r. Currently we only consider single event triggers, however the formal model is supporting combined events such as $e_i \wedge e_j$ or state formulae (i.e. ITL formulae that do not contain temporal operators). Conceptually this stage is generating an abstracted interval $\sigma_{s,r}$ of the interval σ_s that contains only those states in which evt_r is true. This is depicted in Figure 3.5.

3.5.2 Stage 2

Second, for every rule r the condition of the rule cnd_r is evaluated against the corresponding abstracted interval $\sigma_{s,r}$. The condition defines the distance between two consecutive actions triggered by the same rule. This means that the temporal formula cnd_r must hold over the subintervals of $\sigma_{s,r}$ bridging the gaps between the projected states. This is depicted in Figure 2.3.

Formally this means that the policies relate the service's event trace, viz. the interval σ_s to actions that are performed by the service as follows:

$$\sigma_s \models \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

Here $\bigcirc\mathsf{halt}\,(evt_r)\Delta f$ conceptually yields the abstracted interval $\sigma_{s,r}$ over which the policy rule is evaluated. The condition cnd_r of the rule then bridges between two consecutive actions that are performed as a consequence of the rule.

The rationale for separating the two steps is that the filtering of event streams based on simple events (evt_r) can be implemented very efficiently, whereas the complexity of the evaluation of the conditions cnd_r is more complex and can in certain cases grow linearly with the number of states that are bridged. Thus the initial reduction using the event filter reduces the complexity of the latter evaluation.

The overall service specification is then constructed from this as:

$$\sigma_s \models \bigwedge_{r\in pol_s} \bigcirc\mathsf{halt}\,(evt_r)\Delta(cnd_r\Delta\Box act_r)$$

The specification of act_r is not detailed here and we only consider that the relevant action is initiated in that state of the service interval.

The model can be expressed from its semantics using AnaTempura (Hal88, CMZ11), resulting in the following code:

```
/* run */ define example() = {
exists Evts :
{ /* create test event trace for the service */
  list(Evts,3) and stable(struct(Evts)) and evtmodel(Evts) and
  { /* example rule evaluation */
    (next halt(Evts[0]=1)) /* selecting events Evts[0] */
    proj{ /* show selected events, testing only */
      always format("Evts[0] = 1\n") and {
      len(2) /* select every second event only */
      proj{  /* show selected events, testing only */
        always format("Action on every 2nd Evts[0].\n")
}}}}}}.

set assign_ahead = false.
```

```
define evtmodel(Evts) = {
  Evts = [1,1,0] and skip ; Evts = [1,0,1] and skip ;
  Evts = [1,1,0] and skip ; Evts = [1,0,1] and skip ;
  Evts = [0,0,0] and skip ; Evts = [0,0,1] and skip ;
  Evts = [1,0,0] and skip ; Evts = [1,0,0] and empty
}.
```

Here three events are modelled for the service, and an example trace is generated by the function `evtmodel(Evts)`. AnaTempura can be run in a runtime verification mode and could receive these events from an external program. The event trigger for the encoded rule is `Evts[0]`, where a value of `1` indicates that the event occurred. This is encoded in the first projection condition `(next halt(Evts[0]=1))`, which in effect generates the more abstract interval $\sigma_{s,r}$ over which the second projection is taking place. In this example the temporal condition is selecting every second of the events (`len(2)`) on which the action of the rule is triggered. In this proof of concept only a statement is printed out to the screen, but instead a message could be easily send to another service. The above code can be readily executed in AnaTempura (available at `http://www.cse.dmu.ac.uk/STRL/ITL/`) and will produce the following output:

```
State   0: Evts[0] = 1
State   1: Evts[0] = 1
State   1: Action on every 2nd Evts[0].
State   2: Evts[0] = 1
State   3: Evts[0] = 1
State   3: Action on every 2nd Evts[0].
State   6: Evts[0] = 1
State   7: Evts[0] = 1
State   7: Action on every 2nd Evts[0].

Done!  Computation length:  7.  Total Passes:  9.
Total reductions:  297  (293 successful).  Maximum reduction depth:  11.
```

The event `Evts[0]` is raised in the states 0, 1, 2, 3, 6 and 7 as also indicated by the control outputs. The Action is triggered on every second occurrence of the event, namely in states 1, 3 and 7.

3.5.3 Policy Validation using Stream Model

The same can also be expressed using the source, stream and action metaphor and executed over data stream semantics. In this sample we are using an F# implementation of operators. Throughout the thesis we are using different implementations. This also shows that the concept is not platform dependent. In general we are using functional programming languages, such as F#, Haskell or Erlang, because expressing source, condition, and action pipelines can be expressed very easy and natural in the functional programming paradigm:

```
# create the source data
evtmodel = [{1,1,0},{1,0,1},{1,1,0},{1,0,1},{0,0,0},{0,0,1},{1,0,0},{1,0,0}]

Source.asStates(evtmodel)
|> Stream.map(fn {s,{a,x,y}} -> %{state: s, a: a , x: x, y: y} end)
#selecting events Evts[0]
|> Stream.filter(fn x -> x.a == 1 end)
# show selected events, testing only
|> Stream.print(fn x -> IO.inspect "State #{x.state}: Evts[0] = #{x.a}" end)
# select every second event only
|> Stream.len(2)
|> Stream.map(fn ([h,t]) -> t end)
# show selected events, testing only
|> Stream.print(fn x -> IO.inspect "State #{x.state}: Action on every 2nd Evts." end)
|> Pipes.Action.nul
```

Here the `evtmodel` is expressed as a list of tuples. A source stream is created of type {`state, a, x, y`} (with a as availability, x and y as position). However, the logic is a one to one mapping to the AnaTempura expression. First we filter out events which have the availability value 1 (`Stream.filter`). Secondly, we trigger only every second event (`Stream.len 2`). Thirdly, as an action we output the result. As a remark the result in action contains an array of two events. We pick the second element in the list (here: t for tail) and display its state number.

As a result we receive the following:

```
"State 0: Evts[0] = 1"
"State 1: Evts[0] = 1"
"State 1: Action on every 2nd Evts."
```

```
"State 2: Evts[0] = 1"
"State 3: Evts[0] = 1"
"State 3: Action on every 2nd Evts."
"State 6: Evts[0] = 1"
"State 7: Evts[0] = 1"
"State 7: Action on every 2nd Evts."
```

We have expressed the same policy with the operator pipeline and we received the same result. Thus, we have validated our result. We believe that this approach is superior AnaTempura since it is more powerful by offering the full functionality of functional programming in the background in addition to the logical support of ITL over temporal event streams to enable policies.

In general, the advantage of building the policy language on the basis of a formal model is that one can reason about the hierarchy of event filters throughout the service infrastructure. Without loss of generality one can reason about a general stream of events E_{sys} that contains all events that are observable in the system. Whilst the event streams E_s are generated by the individual services s, conceptually they can be seen as a filtered event stream that selects from E_{sys} only those events that originate from s. This approach makes reasoning about the interaction of the various event streams possible and does not complicate the analysis as it uses the same policy-defined event filters that are advocated in this work.

3.6 Examples revisited

So far, this all sounds very theoretical. Thus, it makes sense to revisit some scenarios introduced in 1.1.

3.6.1 Fleet Management

We can revisit the taxi management scenario to illustrate the presented theory with a simple example. In this example we have the user as one peer who is interested in finding a taxi. He is formulating his needs in form of a request providing his current location, which he forwards logically to all taxis (the other

peers). Here, it is a query (see figure 1.1) over an event stream returning the node (taxi) which is available (means number of passengers == 0) and the closest — expressed by calculating the minimum distance between the user location provided by his context and the location given by the taxi event streams. The user requests are joined with the event stream coming from the taxis:

```
/* define taxi and user tupel  */
Taxi = %{ Available :..., GEOPosition : ...}
User = %{ GEOPosition : ...}

/* run standing query */
result = taxis.asStream(Taxi)
         |> zip users.asStream(User)
         |> filter (fn (taxi, user)
             -> taxi.Available && Min(distance(user.GEOPosition, taxi.GEOPPosition)) end)
         |> Action (fn (taxi, user)
             -> select taxi)  /* create a result stream of relevant nodes */
```

Each node (taxi) is sending an event whenever it moves more than 50 meters or the number of passengers is changing indicating a change to the taxi's availability (see Example 1.1.1). There is a so-called standing query consuming the incoming event streams from all nodes (taxis) and mapping the user specific request to the stream. Since, the query is aware of all state changes of the nodes, it is aware of the state of the entire system. Thus, it can reply to the user request with almost zero latency. Here, we can make the assumption that we have four taxis (A,B,C, and D) (see Figure 3.6). At time t1 a user is requesting a taxi. At that point in time only taxi A is available so that the query will return taxi A as result. Thus, taxi A is sending a new event (A2) since it picks up the user. When a new user at time t2 is requesting a taxi the situation has changed. Taxi A is take by user1, taxi B and taxi D are available and taxi C is still taken. Thus, the query will check if taxi B or taxi C is closer to user2. The distance function will return taxi B as the closest.

When another user, user3, at time t3 is requesting a taxi, there is again a new situation and the query might return taxi D as the best fit. Already this simple scenario highlights the capabilities of this approach.

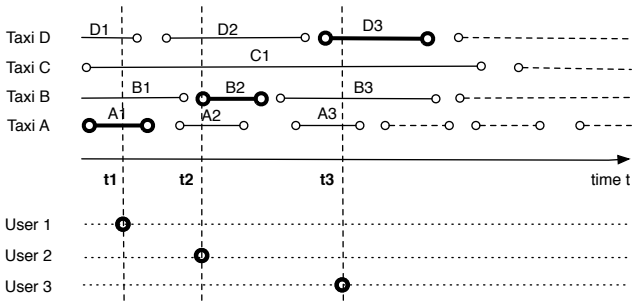


Figure 3.6: Event Stream Reasoning

3.6.2 Friend-Near-By

In the *Friend-Near-By* example we have the users who want to get a notification when one of their friends is near by. To get this notification the user has to subscribe to a query producing notifications about friends' movements. Technically the query is responsible to handle all subscriptions from users and map it to the geo location event streams coming from all user devices. The ECA rule running on the mediator is receiving two data streams: (1) is the stream of the friend relationship and (2) the stream of users' geo locations. Since, the friend relationship stream is considered static the events in this stream are having infinite live time. The geo location stream contains different events for each user and the live time of each event is defined by the policy injected in each device, such as the speed on changing the geo position.

Each node (user's phone) is sending an event whenever it moves more than 50 meters (see Example 1.1.2).

The ECA policy is collecting these notifications and correlates this geo location data with the friend relation information so that we can notify each user if one of his friends is close by, e.g.

- we want to notify Alice when one of Alice's friends is closer than 50 meters

This can be translated into the following an ECA policy:

```
Event: GeoLocationUpdate(x)
Condition: friend(Alice, X) and distance(alice,X) < 50
Action: Notify Alice of X
```

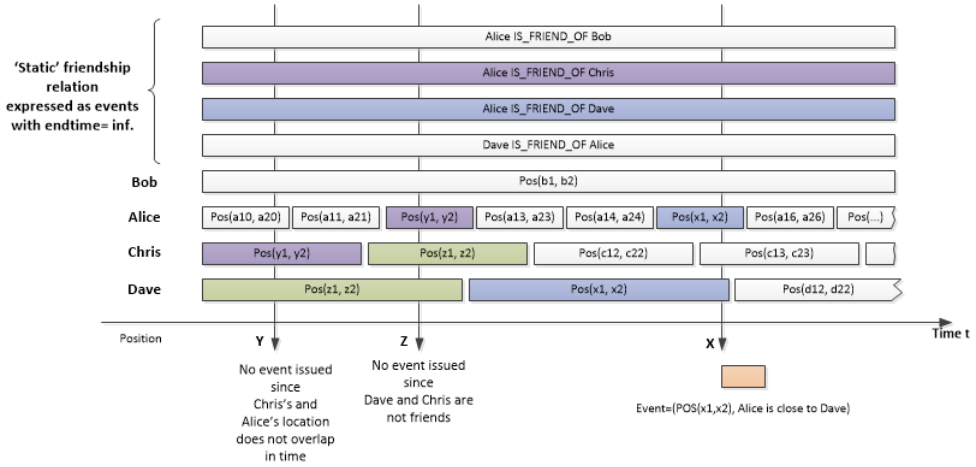


Figure 3.7: Event Stream Reasoning

And this translates directly into:

```
/* define user tuples */
User = %{ GEOPosition : ..., Friends : User[]}

/* run standing query */
result = users.AsStream(User)
            |> filter (fn (user)
                        -> friend(Alice, user) end)
            |> filter (fn (user)
                        -> distance(Alice.GEOPosition, user.GEOPPosition) < 50 end)
            /* create a result stream of relevant nodes */
            |> Action (fn (user)
                        -> notify Alice user end)
```

In our example we do have Alice, Bo, Chris, and Dave (see Figure 3.7). At time Y Chris is at position Y but Alice is not, so there is no action triggered. At time Z Chris and Dave are at the same position but they are not friends. Again no event is issued. At time Z Alice and Dave are at position X and they are friends so that both are getting a notification.

3.7 Summary

As we have seen in 1.1 (1) processing of dynamic and static data and dynamic properties and (2) timely insights through recurrent data processing are key requirements for fast data processing.

This can be achieved by using a model which enables expressing rules of data streams in general and over temporal data streams in particular.

We have developed a model based on the well-known event-condition-action paradigm to express rules. Furthermore, we have extended the rules to be able to get executed over temporal data streams. We call these rules event policies.

We have validated these policies against ITL to ground it on a solid temporal logic. Event policies can be considered as a general concept over data streams to enable filtering, aggregation, correlation and mapping. These policies form a generic concept and can be realised in different environments. We have shown how to use it with AnaTempura and functional programming, but these policies are not limited to those. It is also easy to express it with CEP rules or on top of other programming languages.

Chapter 4

Fast Data Processing in Distributed Systems

"If you thought that science was certain - well, that is just an error on your part."

– Richard P. Feynman

So far the approach was considering processing that runs on a single, physical machine; although it was not restricted to this. The goal is to describe an approach for fast data processing that can run on multiple machines or can span even across millions of machines, a hyper-scale setup. Our approach is introducing data processing units (called PIPES) on each of these machines (called NODES) and thus forming a distributed pipeline for data processing. We refer to this as distributed data processing and believe that it is a suitable foundation for fast data processing in a hyper-scale setup.

In the introduction we mentioned the Internet of Things with scenarios for Smart Cities, Smart Grid or Smart Factory. Common for these scenarios is that they require collecting data from a plethora of different data sources in a highly distributed fashion. The advantage of distributed data processing is:

1. Distributed data processing can exploit the heterogeneous environment where certain resources are available only on certain nodes and passing data around is probably less effective then passing processing logic around; there is always more data than there will be processing instructions.

2. In a highly distributed setup, communication time becomes significant. Therefore time critical information should be processed at the place where it is born. Results can be produced faster without waiting for other input.

3. Reduction of the amount of data which needs to be sent around. Coming back to the taxi example where it does not make sense to pass data, which does not provide new information around. If the taxi does not move why should it send its geo position every second?

However, there are several reasons to consider the distribution of data processing units (pipes) in systems. The architecture enabling fast, distributed data processing introduced here, is called PIPES.

4.1 Event Policy Distribution

Distribution of event policies over devices enables the processing of events as close as possible to data sources and also enables scalability, adaptability, and availability in stable and unstable environments. The deployment of event policies and operators is mainly triggered by the need to process data as close to the source as possible and to avoid sending raw data through the network.

Constraints from devices, networks and data availability for policies are key factors for distribution. In order to be able to support complex analysis of events, we need to employ general mechanisms that can provide the foundation for the efficient processing of raw event data streams.

A very concrete scenario would require adding sensors to at least one production tool to monitor all wafer processing parameters including energy consumption. This will allow us to visualise and compare the energy consumption of those tools for energy-aware tool matching. The framework needs to store process values in real time with a high sampling rate. Using a technology like

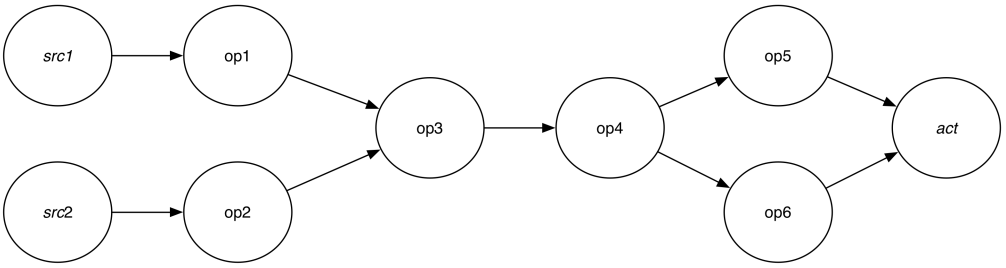


Figure 4.1: Graph of operators

event policies for data extraction and data aggregation can help to forward only specific values to other process control tools. These process control tools can use these aggregated values for correlation with other values for making production more effective.

As described in section 3.4, an event policy is a graph (see figure 4.1) of one or more operators. Each operator exposes one or more output streams. These output streams can be used as input streams for other operators. The basic data are events. Thus, the combination of operators is not restricted and only defined by the semantic of the query.

Since we have defined a policy *pol* as a set of rules r_i. Each r_i is defined as a tuple of $\langle src, cnd, act \rangle$ with *src* being a source of events, *cnd* a graph of operators and *act* the triggered action. A condition *cnd* can be split. We can take the rule r and split the rule into rule r' and rule r'' (see figure 4.2). The condition *cnd* is split into condition cnd' and condition cnd''. As an interaction between both conditions we add another action act' and source src'. The action act' is forwarding the output of cnd' to src' which acts as the input for cnd''. Thus, we can write

$$src| > cnd| > act$$

as

$$src| > cnd'| > act'..src'| > cnd''| > act$$

With this approach we keep the concept *src-cnd-act* and we can split each query for distribution.

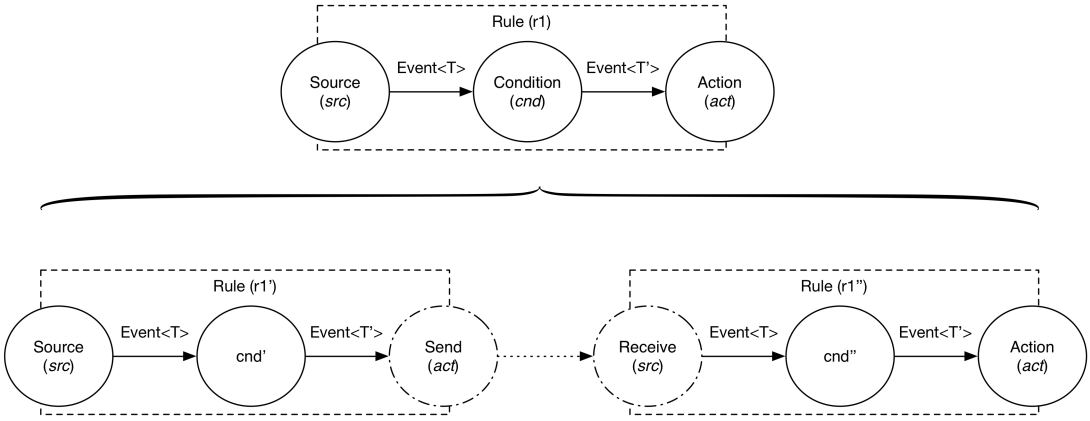


Figure 4.2: Event Policy splitting

When we talk about distribution of a policy we mean that parts of the query graph are executed on different computing nodes (e.g. a PC or PLC). Therefore, we have to understand what it means to place some parts of a policy on a separate node.

The simplest case is the linear graph. Within such a setup there are operators op_3 and op_4 so that the output of op_3 is used as input by op_4 (see figure 4.1). Here, op_3 and op_4 can be easily placed on two different compute nodes. Distribution is required when we need to sample down the data rate, pre filtering or aggregation on a node. This is useful to reduce data traffic over the network. For example, sensors are delivering data at a sampling rate of 1000 Hz. This would be too much to transport all data over the network since most of the raw data is not needed at the backend. Therefore, op_3 can do some resampling and aggregation close to the sensors before data is send across the network. By reducing the sampling rate from 1000 Hz to 10 Hz and aggregating the data (building the average over a time interval) the amount of data can be reduced by factor 100. Data traffic can also be reduced by filtering out data by op_3. In anyway, the drawback of distribution is latency caused by network latency. Usually, there is no way around distribution since insights needs to be displayed at one place. The challenge is to find the optimal way to process data and to optimise data traffic. Event policy distribution provides a flexible way to split processing logic and helps to overcome this challenge.

A more complex setup is the grouping although it is the most obvious. The operator op_4 is splitting the events into sub queries, here into operator op_5 and op_6. The grouping happens usually by some key value or group condition. But the distribution is quite simple. Each operator, op_5 and op_6 can run on its own compute node. This approach clearly makes sense for parallelisation of work (each sub query can do its own job) or to distribute for optimised capacity (e.g. memory) usage.

This leads to the most complex distribution approach, the join. If we split queries into sub queries as described above we might want to join them at the end. We also often have several data sources and they need to be joined for processing. There are operators op_1 and op_2 which feed their results into operator op_3 which should join the inputs from op_1 and op_2. In a first place both op_1 and op_2 can run on separate compute nodes. Therefore, we have to add two actions act_1 and act_2 which are forwarding results produced by op_1 and op_2. A new source src_3 needs to collect the data from act_1 and act_2 feed them into op_3. The figure 4.3 shows also the *virtual* actions and sources for a distributed setup. Since there might by events from act_1 or act_2 which might be delayed caused by network latency, src_3 has to handle them properly. This means src_3 can either wait or discard late arrivals of events. For example an additional clock stream joined with src_3 could move time forward and thus discard late arrival of events (see section 3.3). Whenever we separate two operators by distribution we have to add a new action to the sender side (for example act_3) and a new source to the receiver side (for example src_4). The semantic of the original *src-cnd-act* rule is kept but distribution is added. If only the semantic matters actions and sources can be skipped within the operator graph. If process borders matter actions and sources should be added as shown in figure 4.3.

4.2 Distributed Architecture

There are different approaches to enable distribution of processing in a system. One very promising approach is to use explicit message passing and encapsulate

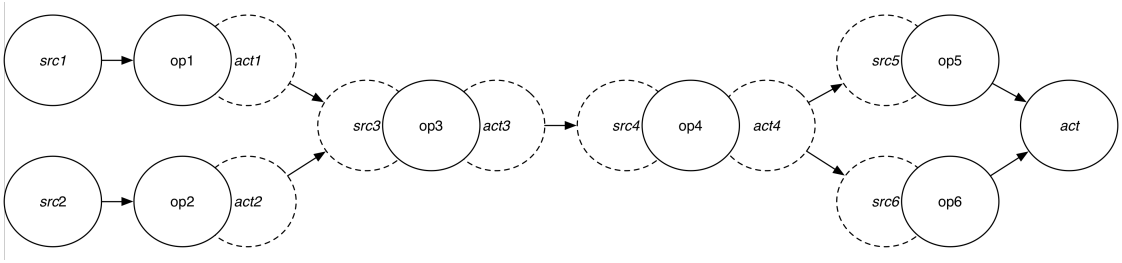


Figure 4.3: Distributed event policies

processing and state in a standalone unit. This is also called the actor model (HBS73).

The core element of the actor model is an *actor*. This actor encapsulates its state. The state can only be changed by sending explicit messages to the actor. In addition it is also possible to get the state from the actor by simply asking the actor about its state. Developed already in 1973 the actor model became very popular in distributed systems for mobile companies. The actor model was perfectly made for handling millions of mobile phone connections and their state. Interestingly, the same concepts are very useful for internet games. Millions of parallel gamers (or their avatars) need to be handled, their status needs to be maintained and their interaction needs to be tracked. In fact, Orleans (BGK+11), an actor model implementation in .NET, is used to do exactly this for games.

Therefore, the actor model seems to be a great candidate as a basis for our fast data processing architecture. Each actor can run a machine or node, explicit message passing is used to send data, results, or aggregations around. There are several actor model implementations so that the architecture can be realised on different platforms:

- In Erlang (Arm97), (Lar09) the actor model is realised as a first class citizen. It is used as a core concept and ensures scalability and reliability. Erlang is used in applications for telecommunication.

- Orleans (BGK+11) is a software framework for building reliable, scalable, and elastic cloud applications. Its programming model encourages the use of simple concurrency patterns that are easy to understand and employ correctly. It is based on distributed actor- like components called grains, which

are isolated units of state and computation that communicate through asynchronous messages.

- Akka (`http://www.akka.io/`) is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM using the actor-model.
- Haskell (`http://www.haskell.org`) is also looking into some concept similar to actors (EBPJ12) with their Cloud Haskell approach.

4.2.1 Distribution Middleware

It is reasonable to wonder whether there is even demand for an architecture for distributed data processing. Is it not possible to simply use what exists in terms of the actor model and pass messages around?

The proposed middleware is called *PIPES*. PIPES combines existing approaches and enables programming of data processing pipelines. Think about a single program that happens to spawn through your entire hyper-scale systems collecting and processing data and acting accordingly. PIPES provides a whole infrastructure suite that supports distributed data processing for any application domain. There are some important features it provides, which go beyond that readily available in actor models.

Specifically, PIPES

- extends the actor model with the capability to spawn an actor remotely and inject processing logic remotely.
- allows to change logic during runtime.
- provides a registry which enables a user to write a pipeline against registered nodes where each node provides information about data sources.
- allows automatic deploying and linking of pipes during deployment.
- allows supervision of nodes and pipes.

The PIPES middleware consists of the following core elements (see Figure 4.4). Most names correlate to the already mentioned terms, except the event policy is called *expression* and is executed within a *cell* here:

1. **Source (src)** produces or injects a continuous stream of events. A source is used as a connector to the outside world for the PIPES system. There are several implementations of a source. A simple one is the *RandomSource* which creates random values by a given interval. The *PushSource* exposes an endpoint to which data can be pushed form outside of a pipeline. A *PullSource* can be used to get data in in a request-response way. The sources enable the convergence of pull and push systems.

2. **Condition (cnd)** defines the transformation of events. A condition can contain concatenation of transformations in the form of operators. Operators are: Map, Zip, Filter. This concatenation can be expressed by using the pipe symbol (| >). Example:

   ```
   source
   |> filter(fn x -> isOdd(x) end)
   |> map (fn x -> x * 2 end)
   |> action
   ```

   In this example the filter and map functions are forming the vertex.

3. **Action (act)** triggered by the condition result. An action can either send out the result to other sources, can trigger an action or simply output the result. The action is also used to hold the final state of a pipe.

4. **Rule** is defined by source, condition, and action

   ```
   expr = fn ->  source |> vertex |> action end
   ```

5. **Cell** is a container for expressions. A cell is an extended actor since the actor behaviour is injected:

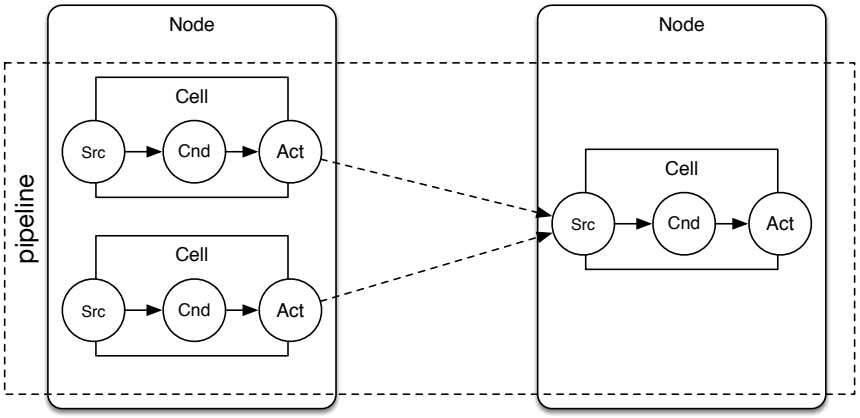


Figure 4.4: PIPES terminology

```
PIPES.send(node, cell, expr).
```

A cell instance is defined by its unique cell identifier (PID).

6. **Node** is the physical environment hosting multiple cells. A node can be identified by its unique node identifier (NID).

Different cells can be connected by joining sources and actions. By connecting cells it is possible to build highly distributed processing pipelines. Compared to the classical actor a *cell* is not only encapsulating its state and can receive events, it is possible to inject the behaviour of a cell. This is an extension to the original actor model.

We can find similar approaches in Cloud Haskell (EBPJ12) where closures can be serialised to a mailbox as expressions or functions. This is more restricted to closures and is not as complete as our approach. In Orleans (BGK+11) we can find an actor model implementation on C/.Net but the flexibility is limited to sending around behaviour in form of expressions.

The notion of sources and actions allow us to have a flexible integration of push-based and pull-based approaches. With *PullSources* classical REST services can be integrated (see 4.5).

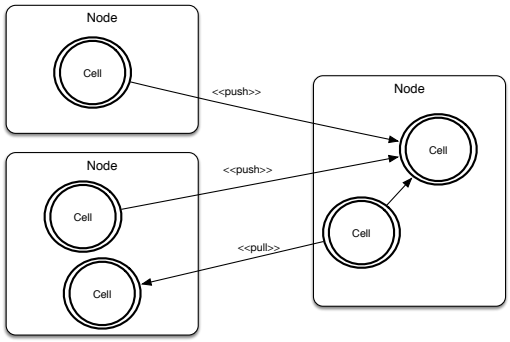


Figure 4.5: PIPES topology

Classically a system supporting actors can have thousands of actors running in parallel. The overhead and costs for running an actor is very small, consequently a *cell* also needs to be very small in terms of overheads. A *cell* can also be realised in a reliable way. The *cell* just has to persist its state. If the *cell* would fail and not react anymore at some point a supervisor of the *cell* could just restart another *cell* on the node and restore the state from disk.

4.2.2 PIPES Middleware

The PIPES middleware has four core elements (see figure 4.6). A *supervisor* is monitoring the other core elements and ensures that they are alive. If the supervisor gets notified or detects that one element is not responding anymore he triggers a restart. Since the other components can persist their state it is easy to restart if one breaks.

The *registry* is managing *nodes* and *cells*. Both register themselves with their unique *name* as soon they get created. When a *node* or *cell* register itself, the *registry* returns the *NID* or *PID* as unique identifiers for *node* or *cell*. Via the *registry* a user can lookup *nodes* and *cells* via their *name*.

There is one *node* which is called the *master node*. This *master node* runs the *registry* and *supervisor* within its own process. A *node* also runs another *supervisor*, which monitors the *cells* which run inside a *node*. *Cells* are created, started, stopped and removed via the corresponding *node*.

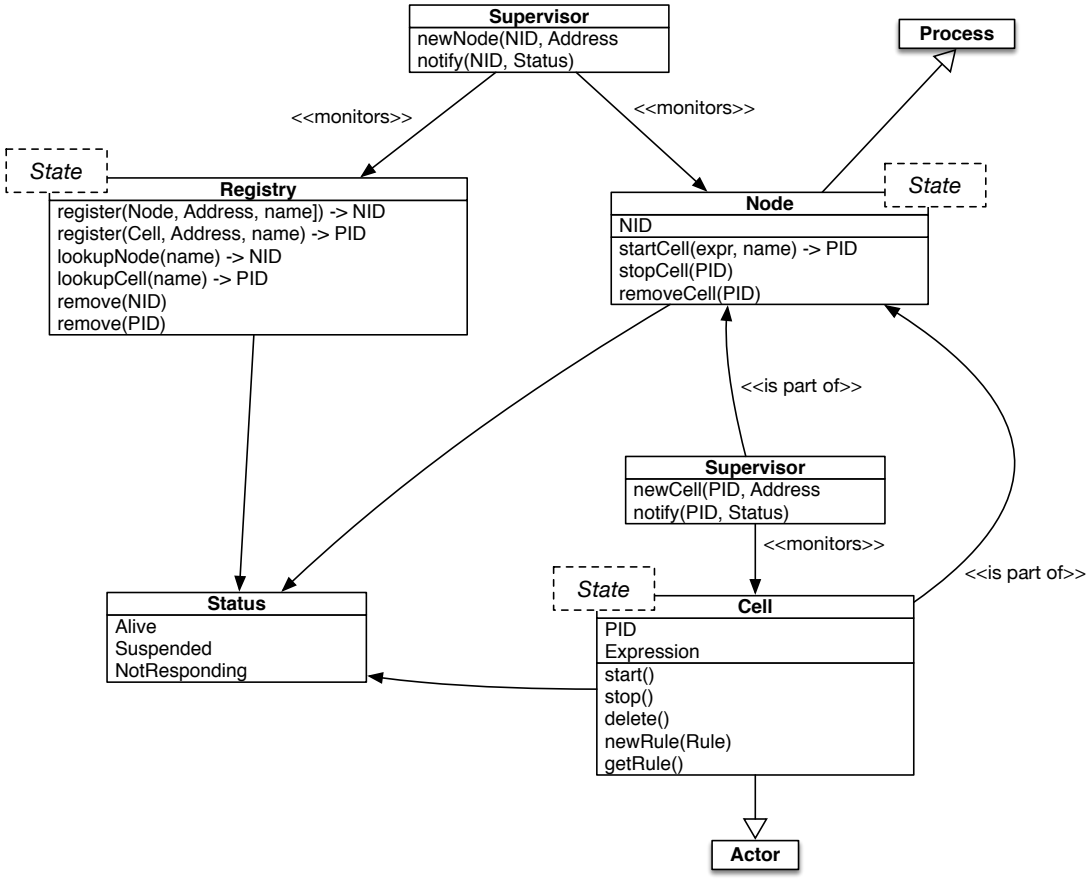


Figure 4.6: PIPES middleware

As already mentioned, a *cell* is an actor. A *cell* itself is defined by its *PID* and its contained *rule* in form of an expression. The *cell* can be started, stopped and deleted. In addition it is possible to inject a new *expression*. However, injecting a new *expression* means restaring the *cell* or at least loosing current processing results.

4.2.3 Towards PIPES Implementation

How does this all fit together? As a remainder in section 3.4 we have introduced some operators enabling data processing over temporal streams. We have validated and grounded these rules (event policies) on ITL. Within this chapter we

have explained how event policies can conceptually be distributed by using *cells*. We have provided an architecture for a middleware to enable distribution and allow management of such a setup by using *nodes*, *supervisor*, and *registry*. How would a PIPES middleware be realised?

Ideally it is realised on top of a functional programming language, such as Haskell, F# or Erlang. Haskell is the purest functional programming language and would be ideal to translate ITL logic into functions. But Haskell has limitations in terms of actor model and distribution. As mentioned, the Cloud Haskell project is trying to build an actor comparable framework, but the project is not mature enough compared to F# or Erlang regarding this aspect.

F# is not a pure functional programming language. It is a mixture of OOP concepts and functional elements. Expressing ITL logic as functions is feasible and there are several actor model implementations for F#, like AKKA, but serialisation of expressions is quite difficult.

The actor model in Erlang is providing a great foundation for the hyper-scaling setup. It can be seen a first class citizen within the Erlang framework. In fact, Elixir is used for Reactor. Elixir is an extension on top of Erlang. In Elixir everything is an expression and can be serialised easily to other instances. This build-in functionality in combination with the superior actor model implementation makes Elixir the first choice for realising the PIPES middleware.

4.3 Examples revisited

In the following section we are revisiting some scenarios introduced in section 1.1 to explain in more depth the approach.

4.3.1 Pay-as-you-Drive

Consider Figure 4.7, which shows the pay-as-you-drive scenario in a form that it fits into the PIPES system. The logic to calculate the BDI is one pipe:

```
expr2 = fn ->
    velocitySource.asStream |>
    zip (laneChange.asStream) |>
    zip (brakes.AsStream) |>
    win 60*60 60*60 (fn x -> sum(x) end) |>
    Action.send (NID#2, BDI, PID)
end
```

We consider three sources: the velocity, the notifications of lane changes, and the usage of the brakes. Then, we zip all three streams to have one combined stream. The window operator is calculating a window over an hour (60 seconds multiplied by 60 minutes) and also hops one hour. The window is used to sum the values of the three streams. The action is then forwarding the BDI every hour to the master node.

The cell on the server is collecting the BDI events and keeps the state. Whenever a customer is asking for a driver index, the cell can simply forward the result. It could also do the ADI calculation by combining the BDI with additional information, e.g. weather. The expression of the cell would look like this:

```
expr1 = fn ->
    bdi.asStream |>
    zip (weatherService.asStream) |>
    map (fn x -> ADI(x) end) |>
    Action.keepState(...)
end
```

To enable an end-2-end processing pipeline follow these steps:

1. Step 1: Author the *expression* expr1
2. Step 2: By calling *Node.start()* the expression will be send to the node with *NID2*

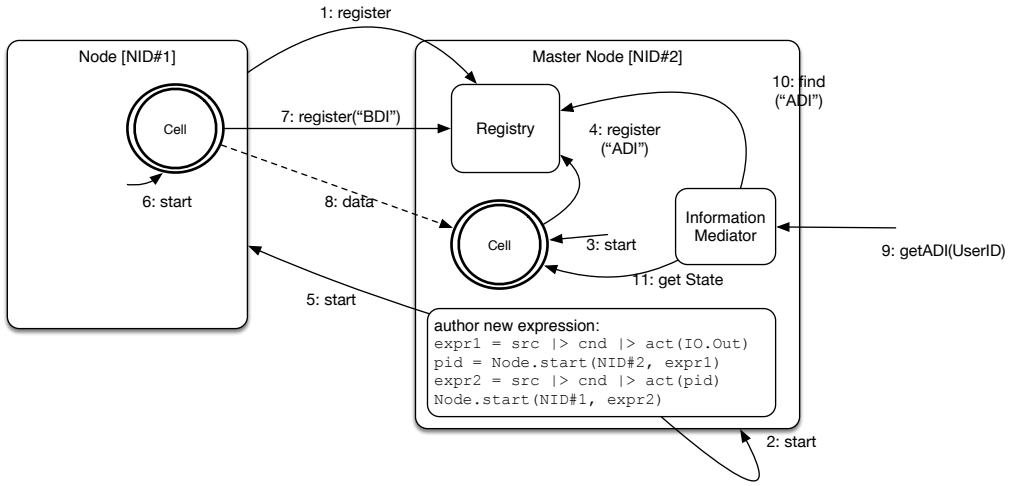


Figure 4.7: PIPES pay-as-you-drive example

3. Step 3: the node with *NID2* will start a new cell instance hosting the given expression (expr1)

4. Step 4: The cell will register itself at the *registry*. In this case with the tag BDI which needs to be defined during expression definition.

5. Step 5: The next expression will be send to node with *NID1*

6. Step 6: The node with *NID1* starts a cell with given *expression* expr2. This can be scaled out to multiple nodes if required.

7. Step 7: The new cell will register itself at the *registry* with the tag "ADI".

8. Step 8: Finally, the cell in *NID1* sends data to cell in *NID2*

9. Step 9: Now, considering a customer looking for the ADI of a driver. He has to provide the driver id and calls the information mediator endpoint.

10. Step 10: The information mediator finds the cell with tag ADI in registry

11. Step 11: Without any delay the information mediator collects state from the cell and sends this back to customer.

4.3.2 Shop Floor Monitoring

The shop floor monitoring scenario is requiring aggregation of data already at the place where the data is born. This means on the PLC, which is connected directly to the sensors. The sensor data is read from the field bus by the source on the device (here: Node#NID1). To aggregate data over 20 seconds, calculating the average of the sensor data over 20 seconds and send it, the policy could look like follows:

```
expr1 = fn ->
    sensor.asStream |>
    win 20 20 (fn x -> avg(x) end)
    Action.send (NID#2, \'devices\', PID)
end
```

The action is sending the data to the master node (here: Node#NID2). On the master node we are keeping it simple. The policy is simply receiving the data and forwarding it to display the data on a dashboard.

```
expr2 = fn ->
    devices.asStream |>
    Action.show (fn x -> display(x) end)
end
```

In cases of an alarm the operator wants to connect directly to the machine, which has triggered the alarm and wants to change the policy so that he gets a higher sampling rate. This can be done by changing the window operator parameter from 20 seconds to 2 seconds for example. So, the operator changes the policy and he also changes the send command to send the data not to the master node but to his tablet device (here: Node#NID3).

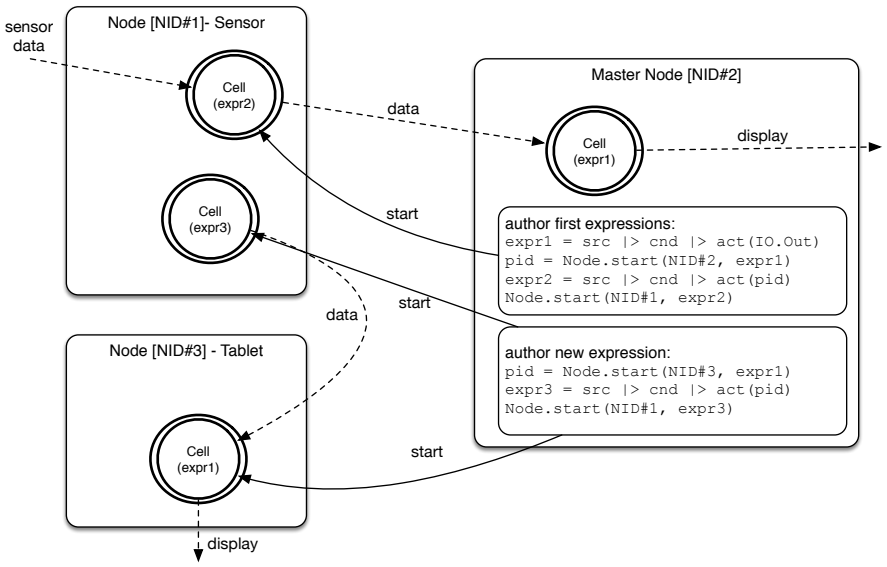


Figure 4.8: PIPES shop floor monitoring example

The policy looks as follows:

```
expr3 = fn ->
    sensor.asStream |>
    win 2 1 (fn x -> avg(x) end)
    Action.send (NID#3, \'devices\', PID)
end
```

Figure 4.8 shows the schematic setup for this scenario.

4.4 Summary

We have introduced an architecture to distribute event policies in a hyper-scale system. The proposed architecture is aligned with the basic concepts for fast data processing in section 3.4. The idea of grounding the middleware on top of the actor-model ensures that it works for systems at scale as the actor-model is already in use in highly distributed telecommunication applications.

The PIPES middleware is an easy to use and distributable concept, which can run on everything from small devices to large cloud, based systems. Thus, there are no limitations for this approach.

We have also validated that it works semantically based on the provided scenarios.

Chapter 5

Mediation Architecture for Hyper-Scale Systems

"When my information changes, I alter my conclusions. What do you do, sir?"

– John Maynard Keynes

Nowadays businesses as well as the Web require for information to be available in real-time in order to reply to requests, make decisions and generally stay competitive. This in turn requires for data to be processed in real-time. In general in service-oriented architecture (SOA) we are less concerned about latency of data processing. Clearly, there are investigations of service-level agreements (SLA) and quality of service (QoS) to guarantee service delivery. Based on this, several approaches on monitoring SLAs have emerged and solutions to find most relevant services for a given context have been developed. Most of this work is assuming that the relevant information for decision making is available and accurate.

Properties for service selection are considered to be non-functional or functional, and the available approaches are based on the fact that properties are pulled from service repositories (that is from service metadata) or possibly from

the services directly before the algorithm determine the most relevant service for a given context. Repositories are useful for static data and polling services directly works if a small number of properties of a small number of services is of interest. We believe that there is an emergent need to provide methods to enable the continuous evaluation of functional and non-functional properties especially in the case where the number of services is high (Cho08).

Lets assume there is a user who tries to locate the nearest printer with the shortest print queue because he has a deadline and needs to print out an important report. Therefore, the system needs the location of the user, typically part of a user profile, the geographical location of the printers, and information about the print queue of each printer. In service selection, an algorithm compares the location of the user with the location of the printer taking into account the number of documents in each print queue. There are several approaches which are able to identify the most relevant printer within a given context so this is not the challenge we are tackling; we are interested in obtaining the data that is used for the decision making. The geographical location is static information it does not change continuously over time. We will be using the term static property for properties whose values are static over time. The number of documents in the print queue is not static it is time dependent and changes over time as documents are printed or new documents are added to the queue. Hence the length of the print queue is a dynamic property.

5.1 Convergence of request-response and event-based Service Interaction

As we have pointed out in (TRMJ12) and in (TRM11), to achieve almost zero latency data processing, data must be available at the place where the consumer needs it, such as a data provider. So, instead of pulling data at request time from data sources, data should be pushed to such a data provider. If we apply a scan-based approach to an SOA, this would mean that a consumer is pulling data from services (see Figure 5.1a). In contrast an event-based approach would mean that

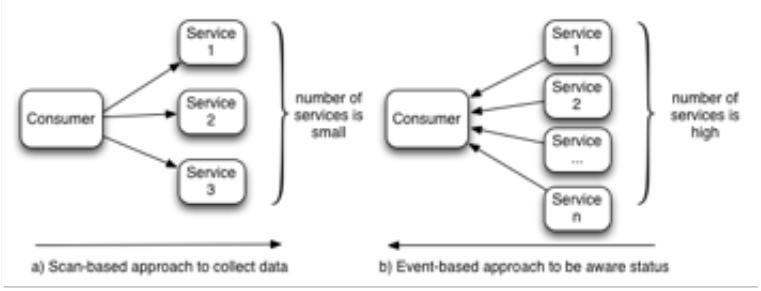


Figure 5.1: Metaphor comparison of request-response and event-based in SOA

services are pushing data to a consumer (see Figure 5.1b). The communication is still initiated by the consumer (see section 1.1.1).

In an SOA it does not matter which approach is used since both can be employed together. Usually, these interaction patterns are already supported by technologies enabling SOA, e.g. Web Services. In Web Services there are request-response paradigms as well as event paradigms used individually and in combined fashions. Thus, it is just a matter of how services are composed (see Figure 5.2). Here the service A is requesting data from services 1 and 2. While service 1 is composed of service 1.2 and 2.2 it will request data from them by scanning to its composed services. In contrast service 2 is composed of several services 2.1...2.n. Here the services are pushing their data asynchronously to service 2. This resembles the event-based approach more closely.

Using event-based models is only the first step towards a faster processing of data in terms of providing results with low-latency. If the data sources are continuously pushing data to a data provider (e.g. the selector) there is a vast amount of overhead by unnecessarily transferring data – a waste of bandwidth.

Event policies can be as smart as possible by using various sets of information, such as the prioritization of the data. Consider for example an alarm situation with cascading alarms. Such a system has to ensure that the most severe alarms are delivered and the bandwidth is not occupied with unimportant information (see section 1.1.5. Thus, event policies executed on smart data sources - intelligent objects - should enable low capacity filtering by being context-aware.

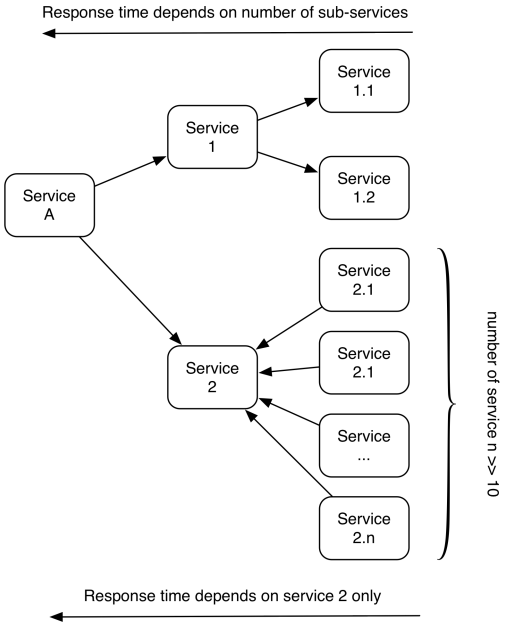


Figure 5.2: Service dependencies

5.2 The Pull/Push Model

It is quite challenging to get an accurate view of this data with classic request-response approaches, which are usually employed in SOA. Consider the number of printers within in a company, all taxis of a company within a city, or even the shuttle service on a large company campus. Here the number of possible services, namely printers, taxis, or shuttles is high. In addition the length of the print queue or the geo location of taxis or shuttles change very frequently they are highly dynamic properties.

Using a typical request-response approach every time a user asks for a taxi the system has to poll all the taxis geo locations and other properties just to be able to identify the most relevant one for the request. If we consider that this might be 50 or even 100 taxis we get a feeling for the scale. In such realistic settings it is becoming quite challenging to answer a simple question such as find the nearest shuttle to my location quickly.

We can define this more crisply as a need for a concept delivering responses with low latency based on dynamic service properties at any time to consumer

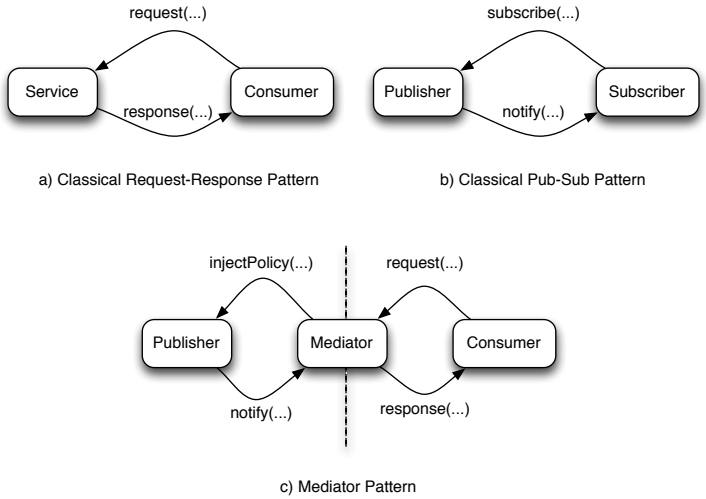


Figure 5.3: Mediation Pattern

requests from huge lists of services. Basically, we propose to combine existing request-response approaches (Figure 5.3(a)) with publish-subscribe techniques (Figure 5.3(b)).

Services offer dynamic properties to which consumer can subscribe, such as the dynamic *GeoLocation* property of a taxis service and the number of current passengers from which the system can derive if the taxi is available or not.

We envision that our approach can be adopted easily as it only requires the addition of two interfaces: (1) The publisher endpoint is exposed on the service side to which the consumer can register or subscribe to events and (2) the subscriber endpoint is exposed by the *Mediator* to enable the services to fire events in a fire and forget fashion (see Figure 5.3(c)).

The publisher interface which enables the *registry* to subscribe to a set of dynamic properties provides two operations:

$$injectPolicy(Policy) : PolId$$

with

- *Policy*: the *event policy* describes the topic to be subscribed to, the refresh interval, and the state changes which trigger event notification.

- *PolId*: Unique registration id for the policy subscription

$$unsubscribe(PolId)$$

with

- *PolId*: Unique registration id of injected policy

The subscriber interface offered by the *Mediator* provides only one operation:

$$notify(Event)$$

As defined in section 3.3, an *event* is a tuple of values $event = \langle se, ts, te, p \rangle$, containing the service endpoint address se, time information ts and te, and the payload p. The time information defines the valid start time ts and end time te of the event and the payload is defined by the type of the subscribed topic. For example the *GeoLocation* could be defined as record with *Longitude* and *Latitude*, both of the XML schema type *xs:int*.

5.3 Mediator

As described in (RL06) processing of streaming data is an important practical problem that arises in time-sensitive applications where the data must be analysed as soon as they arrive, or where the large volume of incoming data makes storing all data for future analysis impossible.

As a central instance we use a *mediator* (see figure 5.4). This *mediator* encapsulates the processing of the incoming request from the consumer side and the incoming events from the service side and maps both. The *mediator* is a service and exposed operations (methods) map internally to specific policies. Thus, during runtime the *mediator* is receiving continuous streams of events from subscribed services. Then, an incoming consumer request is handled as a query on subscribed service properties. Instead of pulling at request time all the data from all services the mediator knows at any time the status of all services. Therefore, this allows for service selection in real-time independent of the number of services.

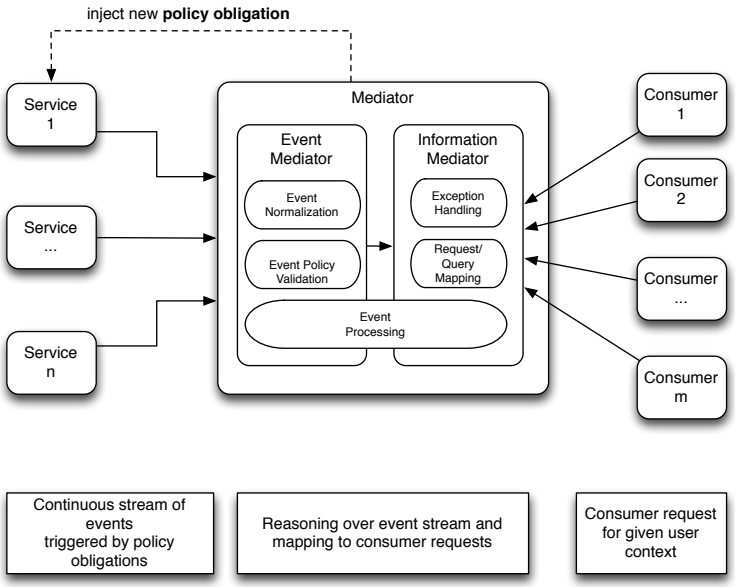


Figure 5.4: Mediation service architecture

5.3.1 Event Mediator

The *mediator* is a special *PIPES node* within the *PIPES* middleware. A specific *pipe* - called *event mediator* - is used to expose the endpoint to collect all incoming events from registered services. An event will contain metadata and payload. The metadata contains information about the time when the event was created on the publisher side. The schema of the subscribed topic, such as temperature or vibration, defines the payload. New event policies are injected via the *mediator* into the correct service (publisher). The *source* of the *event mediator* is exposing the *notifyEvent* method to be able to get called by services.

The *event mediator* is responsible to normalize the incoming data streams. For cases when not all events provide the same data structure the *request mediator* maintains a mapping table to transform incoming events from endpoints into a normalised data stream. This is realised as a rule $EventMediatorExpr$ with a policy *pol*. The policy *pol* is a set of two rules:

$\langle EventMediatorExpr, InformationMediatorExpr \rangle$

```
EventMediatorExpr = fn ->
```

```
pullPushSource.asStream |>
map (fn x -> Normalize(x) end) |>
Action.send (InformationMediator, PID)
end
```

The *service1* provides events containing temperature in Celsius while *service2* provides the temperature data in Fahrenheit. The *event mediator* normalises event streams internally before the event data is forwarded to the *information mediator* via the *pipeline*.

In addition the *event mediator* is able to detect missing events since the refresh time is set within the subscription process. Here it is possible to apply different retention policies to react to missing events, such as simply ignore missing events, use the latest event until a new event arrives, or raise an exception because the absence of an event is an exceptional case. How to handle missing events depends on the scenario and does not require a general solution.

5.3.2 Information Mediator

The *information mediator* maps consumer request to queries on continuous event streams provided by the *Request Mediator*. On the consumer side the framework still offers a normal Web Service interface, which internally needs to be transformed into a query, which is executed over the event stream. The *information mediator* also ensures the quality of the events from event streams, such as duplicated events or out-of-order events. The *information mediator* is another rule $InformationMediatorExpr$:

```
InformationMediatorExpr = fn ->
    source.AsStream |>
    filter (fn x -> InOrder(x) end) |>
    window 20 1 (fn x -> RemoveDuplicates(x) end)
    Action.keepState(EndpointAddress)
```

Here, our approach benefits from the existing work on complex event processing (CEP), such as (LPTL02) or (Luc02). The specific time information we are adding to the event helps to control the quality of events and result. While valid start and end times are generated by the service side the *Information Mediator* also added internal time information (called: System time) to the events. Within the *Information Mediator* internal clock increments are used to move time forward decoupled from external sources. Thus, the order of events is guaranteed and the quality of the results can be ensured. Basically, this is a classical CEP topic and the approach is simply benefiting from using CEP technology here which can find in for example in (BGAH06).

5.4 Filtering at the Source

To control the event flow from services to the mediator the services are accepting event policies as filtering rules. These policies are defining which state changes within a service (on the source) trigger an event (such as "temperature $> 50.2C$") and the expected interval (refresh). The expected interval would then also be used within an event so that the start time is set when the event is issued on the service and the end time is defined by the refresh interval.

In this case we have to define an event policy in a form that it has a threshold filter and in addition a clock stream as a refresh interval, such as

```
sourceFilterExpr = fn ->
    source.asStream |>
    filter (fn x -> x > Threshold end) |>
    zip (clock.AsStream( Interval)) |>
    Action.send (masterId, \'filter\', PID)
```

In general a data source, such as a device, a sensor, or a phone, can be considered as a service or a *PIPES node*. If it is a service we have to pull data from it in some cases. If it is a *PIPES node* then we can inject a policy as described above.

In (PLM12) the authors are describing an approach which is mapped to a real-time system. The approach is based on translating a policy (here a CEDR query (BGAH06)) into a concurrent reactive object (CRO) program.

Section 3.2 has explained the specification of the required filters in detail. Being able to set the event interval rate and condition helps to fine-tune the system to obtain the appropriate balance between data accuracy, response time and data traffic.

5.5 Examples revisited

How can this be used within our motivating examples? Lets revisit one of the samples from section 1.1.5.

5.5.1 Alarm Management

In the alarm monitoring example the requirement of integrating potentially many different data sources in terms of number and interaction was a strong demand. Data sources are on ones side sensors and on the other side configuration services.

We can define a system using *PIPES* concepts. On the node running on the field bus level we can inject an event policy to do the state estimation (see Figure 5.5). This is following the filtering at the source concept. Sensor data can either be pushed or pulled into the state estimation policy. The states are send as resulting events to the backend. On the backend we run a *mediator* receiving the state estimation results.

The *mediator* also injects a new state estimation policy into the field bus node.

```
StateEstimationExpr = fn ->
    sensors.asStream |>
    map (fn x -> EstimateState(x) end) |>
    Action.send (masterId, \'stateEstimation\', PID)
```

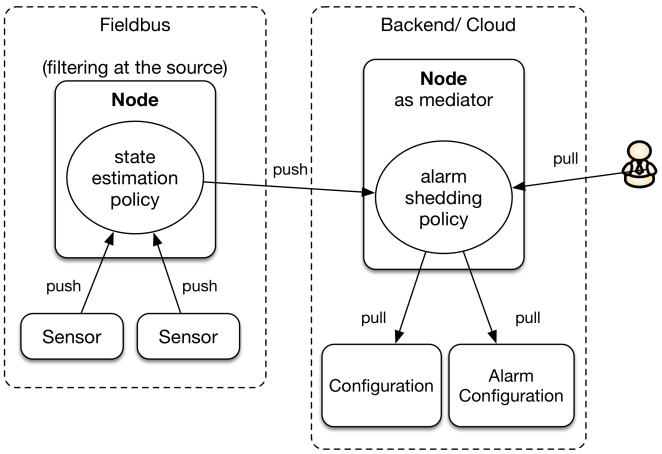


Figure 5.5: Alarm Monitoring example on PIPES

On the backend itself we run a alarm configuration policy. This policy pulls data from configuration services depending on states to provide throttled alarms to the user.

```
AlarmSheddingExpr = fn ->
    stateEstimation.asStream |>
    map (fn x -> Shedding(x, GetConfiguration(x),
        GetAlarmConfiguration(x) end)
    filter (fn x -> Prio(x) > Threshold end) |>
    Action.keepState(...)
```

5.6 Summary

This chapter has introduced the mediation concept. As we have seen from the motivating scenarios there are requirements about integrating services into such a system. These services are usually implemented in way that they are following the classical request-reesponse paradigm.

The mediation enables a grounding concept based on the fast data processing model and the PIPES middleware. The mediator is an integration point between request-response and push-based interaction paradigms.

On one side event policies can help to handle data traffic efficiently by controlling the data flow and the amount of data and on the other side it enables a system to detect data faults and missing data if required.

Chapter 6

Implementation and Evaluation

"The opposite of a correct statement is a false statement. But the opposite of a profound truth may well be another profound truth."

– Niels Bohr

This thesis is about fast data processing in hyper scale systems. The aims and objectives in section 1.3 were formulated around speed, scale and mediation. All topics are main drivers for this work.

The aims are:

1. **Speed**: Data processing model to process temporal service data with near zero latency.
2. **Scale**: Model to enable data processing in highly-distributed and large systems.
3. **Mediation**: Develop an architecture to mediate between different interaction pattern.

Therefore, the evaluation will look into all three aspects. We will evaluate the speed of data processing, the benefits of the mediation concept, and the scalability of the architecture.

So far, we have introduced a formal model for fast data processing with respect to continuous data streams and time related properties. We have provided an easy way to describe logic over continuous data with support for simple types, such as int, string, and time. In addition, this work has introduced an architecture to distribute logic which is capable to support highly-distributed and large systems. This includes concepts like filtering data at the source to manage data traffic and a way to provide insights with almost zero latency by combining push and pull conversations.

But what does *fast processing* mean? How fast is fast and is the proposed programming model and language the right approach at all? Can we verify that the insights can be provided with low latency by facilitating the mediator concept? Is the architecture able to deal with a large number of nodes?

This leads to our hypothesis for evaluation:

> The proposed architecture is capable to process data over data streams fast enough ($\sim$ 1,000,000 evt/sec), the mediator helps to reduce latency for requests compared to full data source scans, it scales linear in terms of pipeline length and event size, and provides the capabilities to realise the motivating scenarios.

Basically, we have to evaluate this work regarding to the objectives. In the following we will (and this is how the chapter is structured)

- prototypically implement the proposed architecture,
- evaluate the continuous data processing approach over data streams (see section 6.2),
- evaluate latency of push and pull based architectures (see section 6.3),
- evaluate the capabilities of the filtering at the source concept in terms of data traffic (see section 6.3),
- evaluate the distribution and scalability of the system (see section 6.4), and
- validate it by implementing a scenario end to end (see section 6.5).

6.1 Implementation

The PIPES middleware is implemented in ELIXIR/Erlang. This means it follows mainly the functional programming paradigm. It turned out that functional programming provides concepts which make writing data pipelines very straight forward and readable. Although there are quite a number of functional programming languages out there (e.g. Haskell, Closure, Scala), ELIXIR/Erlang is our choice here. It should be mentioned that concepts can be adapted to every functional language. It is not limited to ELIXIR/Erlang. But ELIXIR/Erlang provides some first-class elements, such as distribution, supervising, actors, and message and expression passing, which makes ELXIR/Erlang the language of choice for prototyping the system.

The Swedish telecom company Ericsson developed Erlang over 25 years ago. The basic motivation was to enable millions of parallel conversations at the same time, with almost zero tolerance for downtime. Scale and reliability were key requirements for designing core concepts of Erlang. Erlangs syntax derived from Prolog and was heavily influenced by Smalltalk, CSP and functional programming concepts. Fundamental concepts are lightweight processes in Erlang, which can communicate via message passing.

Elixir is a dynamic, functional language designed for building scalable and maintainable applications. Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain. Elixir can be seen as a thin layer on top of Erlang. Elixir inherits pattern matching, higher order functions and the entire process handling from Erlang. Additionally it provides a number of advantages for our work:

- The pipe operator (| >): This is a reminiscent of Prologs DCGs and Haskell monads. Can be read as unix pipe operator. x | > y means call x then take the output of x and add it as an extra argument to y in the first argument position.

- Closures: Closures really behave like closures. fns have the nice property of capturing the present value of any variables that are in their scope so that

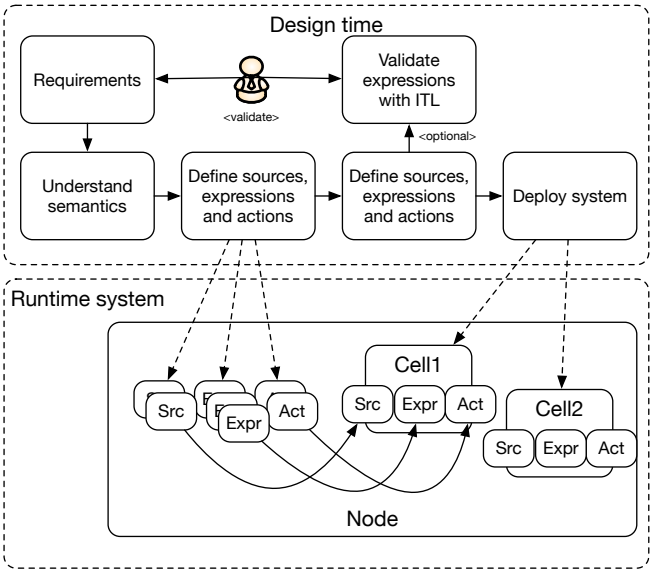


Figure 6.1: Evaluation architectural overview

immutable closures can be created.

- Metaprogramming: Metaprogramming is code that writes code. Thus, it provides an easy way to extent Elixir with new first-class elements and to extend it to data types required by scenarios.

In general, we can split the architecture for evaluation into a runtime and a design time part. For runtime we have implemented the basic functionality of (1) node, (2) source, (3) expression, (4) cell and (5) actions as processes in ELIXIR. All these entities live within the runtime layer (see Figure 6.1). Concrete expressions, sources and actions are motivated by the scenario. For these we have to understand the semantic requirements, such as window length, filter, or specific sources and actions. If expression is in place we can optionally verify it with ITL. Finally, we can deploy the system (see Section 4.3.1).

We are following this approach to evaluate the system in terms of latency of expression pipelines and the overall latency of distributed nodes. As already mentioned, a node can be seen as a system process holding multiple cells. A cell itself is a lightweight process (like an actor or agent). A source is a port to receive messages which gets forwarded to an expression; the logical unit. The result of

an expression is then forwarded to an action. In general an action is a broad concept and can deal as a gateway or forwarder to another cell or for keeping result as state.

A typical way to implement a cell is the following. First we can define an expression as a function or closure, e.g.

```
expr = fn ->
  Source.asStream(name) |>
  Stream.filter(fn v -> v >= n end) |>
  Action.log
end
```

An expression is a function defining the source as a stream, a condition, and an action. This expression can be passed to a cell which takes the expression and literally starts it, e.g.

```
# start cell
name = :node1
node = Registry.lookup(:master)
Cell.start(expr, name, node)
```

The *name* parameter is a unique name which helps to identify the cell. The *node* parameter defines the node on which the cell should run. This *node* can be on a remote computer. With expression passing it is possible to distribute expressions from one central *node* to all other *nodes* in the system. This is a core feature which makes PIPES on ELIXIR/Erlang very easy to use in a distributed setup, like in the fleet management example. There filter logic needs to be pushed to the taxis (nodes) from the backend. Here, each taxi can register at the central node and from there we can manage the rules running on the taxi node. For other systems, like Cloud Haskell, Storm, F# or .NET, it is not that easy to deploy expressions. Complex frameworks are required to serialise expression trees and deserialise it. In ELIXIR/Erlang expressions are treated like datatypes and are serialisable by nature. As a result, PIPES can distribute and manage rules, cells and nodes right out of the box.

6.2 Expression Evaluation

Expressions are rules running within cells. Expressions are handling event streams. Expression evaluation means evaluating the continuous data processing approach over data streams. This is useful to understand how stream processing behaves in general and how the pipeline concept works in particular. It is quite difficult to simply measure performance, throughput or latency without having a comparison. Therefore, we will compare the fundamental concepts of *streams* with *enumerations*.

This leads to our hypothesis for expression evaluation:

> For large data sets *stream processing* provides faster insights compared to *enumerations*.

6.2.1 Lazy Evaluation

Lazy evaluation is a great way to delay the execution of functions on larger datasets. In a typical enumeration each item is evaluated one-by-one. This is not a problem with smaller sets, but as those sets get larger the amount time to process them grows exponentially. Every function has to evaluate the entire set before the next function can execute. The benefits of *streams* is that they allow us to compose our enumerations prior to execution.

First we will take a look at using *enums* to multiply each element with 2. Starting from the following piece of code:

```
1..3
|> Enum.map(fn x -> print.(x) end )
|> Enum.map(fn x -> x *2 end)
|> Enum.map(fn x -> print.("-> "+x) end)
```

We do print before and after the mapping the value and we are receiving the following output:

```
1
2
3
-> 2
-> 4
-> 6
```

We note that each map is applied in order so first all the numbers are printed then they are doubled and printed. The *map* functions has to complete its iteration over the entire set before it can pipe the result list to the step.

The behaviour is different with *streams*. Take the following piece of code:

```
1..3
|> Stream.print
|> Stream.map(fn x -> x * 2 end)
|> Stream.print
|> Stream.run
```

Now we see that the each number is completely evaluated before moving to the next number in the enumeration:

```
1
-> 2
2
-> 4
3
-> 6
```

Compared to the *enum* behaviour this *stream* behaviour is different. For *enum* the entire list is processed completed before there is any result available. This might not be problem with 3 numbers but if we consider an infinite list then we would never get any result. With *streams* we get an result as soon a number is in the system: We start with one, it gets forwarded to the *print*, then multiplied by 2 and finally the result gets printed again. We see that data is processed instantaneously. Here in this example we have not used any parallelisation concepts but we could even consider to process numbers in parallel using the *src-cnd-action* paradigm without any limitation.

6.2.2 Pipeline Latency

Latency is a time interval between the stimulation and response, or, from a more general point of view, a time delay between the cause and the effect of some physical change in the system being observed. Within this setting it is hard to measure latency in general since it depends on the semantics of a policy or pipeline. For example if we use a window over 5 seconds and aggregate the data then this window is the limiting factor with its 5 seconds in terms of producing a result. We cannot expect any result before these 5 seconds.

In general we can compare again *enums* with *streams* in the first place. Thus, we can formulate a policy and measure the time until we receive the expected result. Lets assume we have data points entering our stream, then we can ask how long it will take to have a result at the end of the pipeline.

Definition 1 *If an event e enters the pipeline pol at time t_s and the result of the event is available at the end of the pipeline t_e we can define the latency l as the difference of the timestamps $t_e - t_s$.*

To measure the latency we define a pipeline, which passes only values, which can be, divided by 3 and 5. Finally, we take the 5 first results from the resulting list. We test this with 10,000 up to 10,000,000 events. We measure the time with the *:timer* fucntion in Erlang and injext the policy to be executed. The frame for this looks as follows:

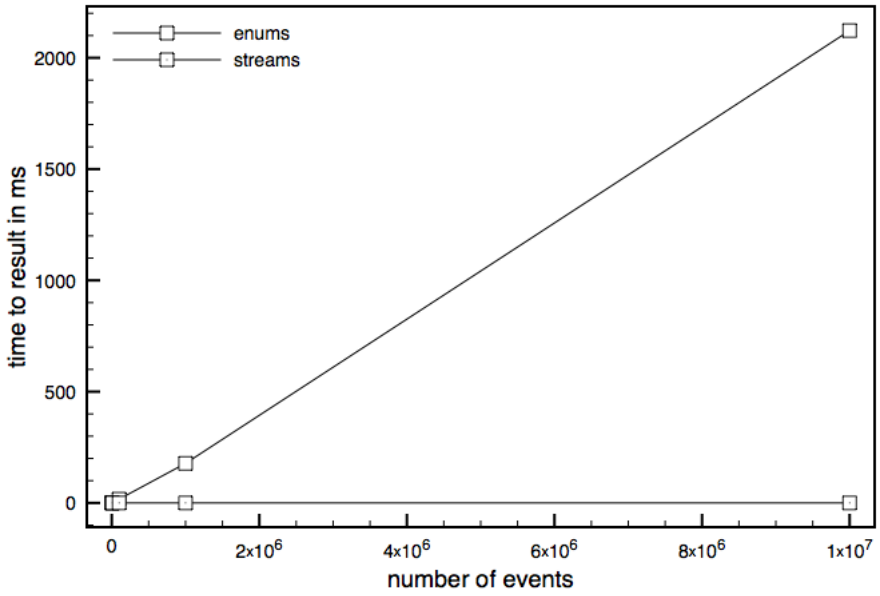


Figure 6.2: Latency comparison of enums and streams

```
result = :timer.tc( policy )
```

Measurements were executed on the following setup:

- Macbook Pro, with Intel Core i7 (3 GHz) and 8GB of Ram running Elixir1.0.1 on Erlang 17.0 (64bit) on OSX

The policy looks as follows:

```
1..n
|> Stream.filter(fn x -> rem(x,3)==0 || rem(x,5)==0 end)
|> Stream.take 5
```

The result can be found in figure 6.2.

For *enums* in every run the entire data set needs to be evaluated before we can take out the first 5 elements. For *streams* this is different. The system evaluates every element. It is as if the event flies through the pipeline and produces a

result at the end. Thus, there is no surprise that the size of the stream has no direct impact on the latency for stream processing - the time it takes to produce a result is constant. This is an important result since we are mainly interested in infinite streams it is good to know that it is possible to define policies, which have constant latency over time.

6.2.3 Pipeline Complexity

Another aspect is important to understand: Which impact does the complexity of a policy have on execution time? The complexity of a policy can be seen by the number of *conditions* (cnd) we are using between a *source* and an *action*.

Definition 2 *The complexity cx_p of an event policy p is defined as the number of conditions within a policy pol. If the policy pol is defined as pol = $src| > \sum_i cnd_i| > act$ with $\sum$ defining the pipeline of conditions as $\sum_i cnd_i = cnd_1| > ...| > cnd_i$ we define complexity as $cx_p = i$.*

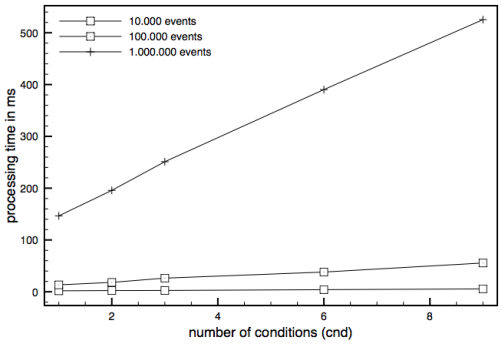
Thus, we have measured policies with different kind of operators compared again to *enum* behaviour (see figure 6.3). We are testing the *map* and the *filter* operator. We have build pipelines with complexity of 1,2,3,6, and 9.

We have varied the number of events injected into the pipeline from 10,000 to 1,000,000 events. Finally, we have measured the time it took to process all events. As a result we can see that both *enums* and *streams* behave the same way.

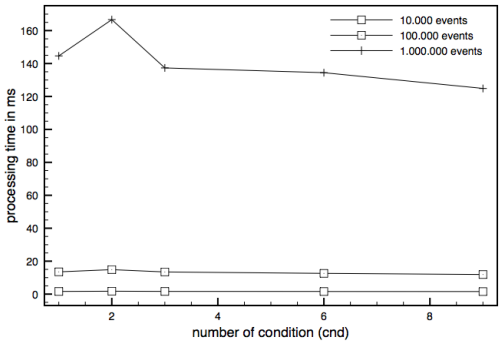
Measurements were executed on the following setup:

- Macbook Pro, with Intel Core i7 (3 GHz) and 8GB of Ram running Elixir1.0.1 on Erlang 17.0 (64bit) on OSX

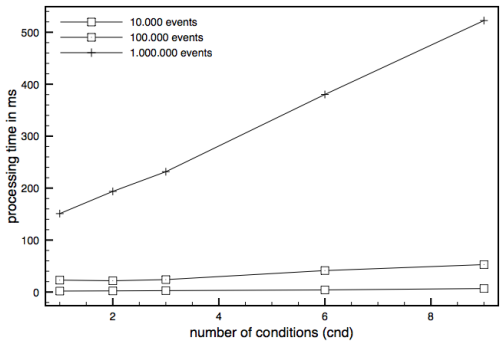
The efficiency of the *filter* operators is constant over the complexity and clearly depends on the number of events (see figure 6.3b and figure 6.3d). But the *map* operator depends linear on the complexity. This means the more *operators* we use within a pipeline the more time we might need, however this is dependent on



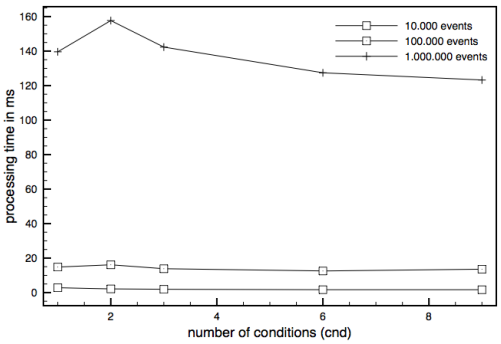
(a) Enum.Map



(b) Enum.Filter



(c) Stream.Map



(d) Stream.Filer

Figure 6.3: Pipeline comparison between Enum and Stream

the specific operators used. This behaviour is the same for *enums* and *streams* (see figure 6.3a and figure 6.3c).

As a note this means that instead of doing

```
1..3
|> Stream.map(fn x -> x * 2 end)
|> Stream.map(fn x -> x * 3 end)
```

we better should do

```
1..3
|> Stream.map(fn x -> x * 6 end)
```

Unfortunately, this is not always possible. Especially, when we consider that we are looking into distributed systems. Sometimes we want to do the first mapping on one node and the second mapping on the following node.

To summarise, streams are enabling processing over data pipelines with very low latency. In scenarios as described in the beginning of this thesis streams are superior compared ti enumerations. Using expressions over streams are promising in terms of latency and complexity in terms of expressiveness. Thus, streams and operators seem to be a promising approach for expressing rules within cells.

6.3 Mediation Architecture

The mediator approach is supposed to be beneficial for large amount of data sources and large amount of requestors.

The hypothesis for evaluation of the mediator architecture is

> Mediation with filtering of events at the source
>
> 1. provides replies with near zero latency and

2. reduces the amount of data transfered (recall that this was large because of the amount of small messages, not because the data in itself being intrinsically large).

The experimental evaluation was geared towards proving these two aspects, so we conducted two evaluations: (1) we measured the latency of finding a result using the pull model compared with a push model and (2) counted the number of messages occurring during a one second time interval in the push model and combined pull-push model. The overall setup considered settings with up to 60000 data sources. We have simulated the setup. The data sources were ELIXIR processes and the master service was simulated as another processes. Thus, all processing happened on one single machine.

This simulations were executed on

- Macbook Pro, with Intel Core i7 (3 GHz) and 8GB of Ram running Elixir1.0.1 on Erlang 17.0 (64bit) on OSX

6.3.1 Pull/Push Model Evaluation

Testing pull and push approaches is quite complex since there is a big number of data sources required to get some significant results. Nonetheless, we had to distinguish between pure latency as a result of the different concepts and latency caused by physical setup, such as network latency and latency used by semantic processing. For testing we decided to run everything on one machine and to simulate each data sources as a small object in our test setup. This object can be seen as an atom, a logical, self-contained unit. By having all simulated data sources in memory on one machine we can say that latency caused by the physical setup can be neglected. The simulated objects (*data sources*) were simply holding a random value and the query in the setup was to find the object whose value is the closest to a given number. This simple setup matches the taxi example where the value would represent the geo position of the taxi and the given number

would be the customer geo position, but also reflects the typical scenario where we see the approach apply.

In the pull approach we are iterating over all objects (*taxis*) and are trying to find $min(value - number)$. In the push approach the objects are pushing their number to the mediator and we run a query over this data, such as

```
expr = fn ->
    values.AsStream
    |> Stream.win 10 10 (fn x -> min(x) end)
    |> ...
end
```

The results are presented in Figure 6.4. While the latency is increasing linearly for the pull approach, it remains almost constant for the push approach. We also find that all values for the push approach are far lower than those for the pull approach, with for example approx. 4ms vs. 65ms for 60000 data sources. This clearly indicates that the pull approach is superior compared to the push approach in terms of latency and the trends show that for a growing number of data sources as those expected in scenarios like the Internet of Things is delivering performance close to no latency. We also want to point out that this is the pure measured latency ignoring network and processing delays – once these are added as additional factors to the evaluation, latency will become rapidly worse for the pull approach as much more communication and more processing is needed compared to push approach which remains constant for the full spectrum (albeit a little slower in real terms than measured in the isolated setting).

6.3.2 Filtering at the Source

The drawback of the push approach is the number of data items sends from data sources to the mediator. Therefore, we have introduced policies (rules) in our approach to avoid that messages are polluting the network unnecessarily. The next evaluation is comparing the number of messages send in a one second interval

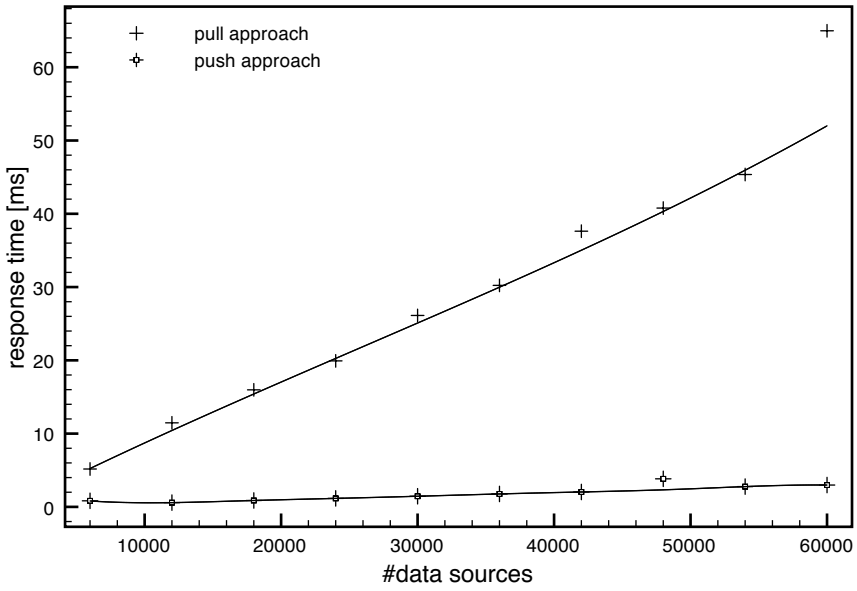


Figure 6.4: Latency of pull approach vs. pull approach

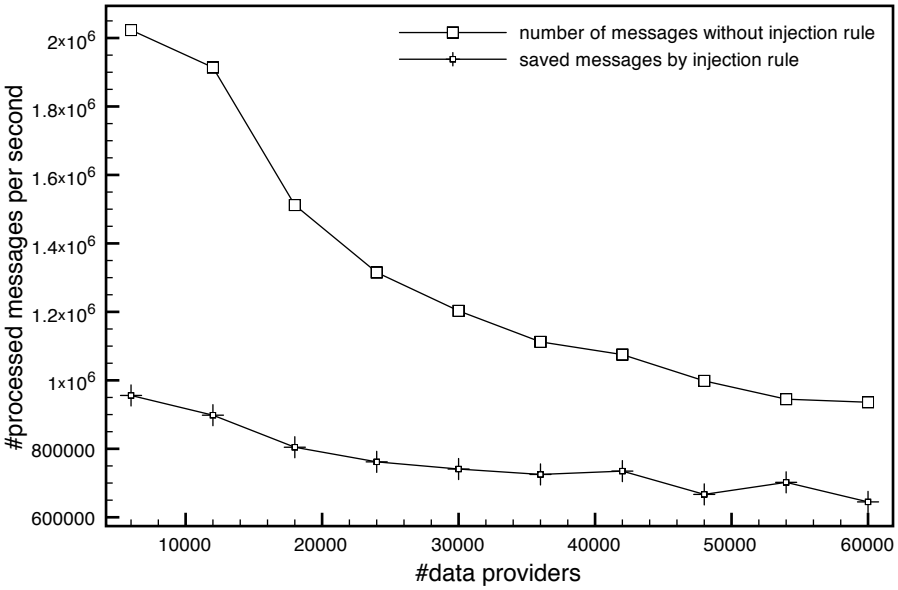


Figure 6.5: Push approach with injected policy and without

in pure push approach compared with the number of messages which sent in a push approach with filtering at the source.

The latter is expected send fewer messages because of the injected policy. For testing we extended the objects with another random value representing the availability. The availability here is a random number, either 0 or 1. The policy is saying that the object should push only messages when the availability is 1 (means the taxi is available).

Figure 6.5 shows the results as we expected. With a simple policy we can roughly save 50% of exchanged messages (based on the randomly changing values). It is clear that with even more specific policies (*such as taxi should have moved more then 50 meters before sending an update*) and real data (a specific taxi being available 50% of the time does not make for a profitable business model!) the number messages can be further reduced.

The number of processed messages per second goes down for both graphs since the receiver node can handle only a limited number of events. It means that the receiving node is dropping events. We can compute the maximum number of events, which are in the system by multiplying the number of processed messages per second with the number of data providers. We can see that with a number of 60000 sources the numbers of messages, which can be processed in sum, are hitting a limit here.

The number of messages is proportional to the size of data exchanged (data traffic) and this relates to bandwidth usage. With the pure push approach we see that we will hit at some point the bandwidth limit. This approach would not support hyper-scale systems.

The usage of event policies and use them to filter already at the source shows a superior approach in terms of bandwidth usage and the flexibility to support distributed systems with up to several thousands of sources.

6.3.3 Integration by Mediation

We have discussed already in section 4.2.1 that the approach can be easily integrated into existing solutions and systems. With cells we can compose complex, distributed graphs for data processing. It is possible to split logic (1) vertically by

simply putting different *src-cnd-act* cells on different nodes and (2) horizontally by simply splitting the processing and running cells in parallel.

This can be handled with specific *source* and *action* implementations:

- **PullSource**: A *PullSource* is used to recurrently read data from an endpoint. The interval i is defining the polling frequency and a defines the type of the expected data offered by the service endpoint e. The syntax looks as follows (again Haskell inspired):

```
PullSource  :: Int -> Endpoint e -> a -> Stream a
```

- **PushSource**: A *PushSource* exposes an endpoint and expects data of type a. The interval i is optional and is used to verify the frequency of incoming data, e.g. checking if data arrives late and can be discarded.

```
PushSource :: Endpoint e -> a -> Int -> Stream a
```

- **SendJSONTo**: The *Send* is a specific action, which forwards data to the given endpoint. The endpoint is defined as a REST endpoint. The *send* action sets a *topic* to enable the receiving source to identify received events. The topic is added to the payload information and the *token* could be a security token. In this example the *send* action is sending the event as JSON:

```
SendJSONTo :: Endpoint e -> Topic -> Token
```

Already with this small set of *sources* and *actions* it is possible to integrate easily into existing systems. The condition itself is decoupled from the integration and works on top of *streams* and *temporal* data information.

First we do have to calculate the ADI as described in section 4.3.1. This would run as a cloud service. On one side data in from of a BDI is pushed every 10 seconds from cars to this policy and on the other side data needs to be pulled every 1 hour from weather services. These two data streams need to be joined to create the ADI (see Figure 6.6). Finally, we are forwarding the result to an insurance company. Such a policy looks pretty simple:

```
ADICalcualtionExpr = fn ->
    PushSource("ADI", driver, 10 seconds) |>
    zip (PullSource(1 hour,
            "http://weather.com/region...",
            weatherData)) |>
    map (fn x -> calculateADI(x) end) |>
    SendJSONTO ("http://pay-as-you-drive.com",
      x.driver_id, secToken)
end
```

In reality we would have to deal with some other details, such as security. But most of this can be handled in specific implementations for sources and actions. Thus, as a result the final event policy can be reduced to something simple as shown.

6.4 Distribution and Scalability

Up to here, we have investigated latency of expressions, and behaviour of mediation architecture.

It is worthwhile to spend a deeper look into the overall scalability and distribution of the PIPES middleware. This means that we have to understand the performance of the connected cells and how this scales over multiple hops.

As a working hypothesis we start from the following:

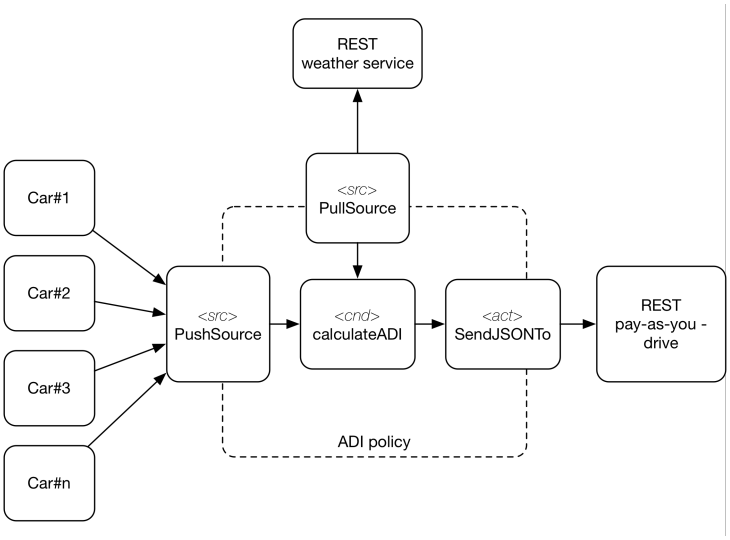


Figure 6.6: Latency of pull approach vs. pull approach

The PIPES middleware scales linear in terms of pipeline length (multiple cells and nodes) and message size. (This requires communication of messages (serialisation/deserialisation) between cells)

For the test evaluation we have implemented sources, cells and actions. A cell can take a cell defined by a source, condition and action, e.g.

```
expr = fn ->
  Source.asStream(name) |>
  Stream.filter(&(&1) >= n) |>
  Action.log
end
```

This expression can be passed to a cell which takes the expression and literally starts it, e.g.

```
# start cell
Cell.start(expr, name, node)
```

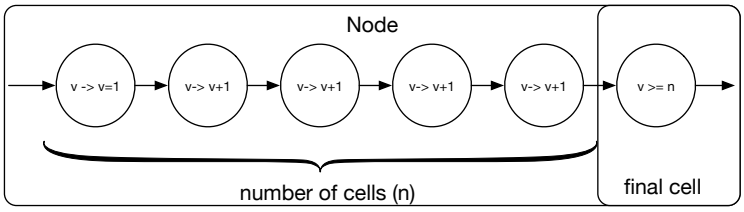


Figure 6.7: Approach to evaluate scale over cells

The name parameter links the source stream to the running expression.

We have varied the number of messages we have ingested into the pipeline from 10,000 up to 3,000,000 and measured the time to process to all incoming messages. To verify that all messages were processed the filter condition was holding back until the final data point (10,000, 100,000,) was reached and printed this to the standard output via the log action. We think it is fair to make use of the filter since it is a valid operator and by holding back the output we could avoid pollution of the result by printing on standard output.

The code to make the measurement looks pretty simple:

```
# measure time
result = :timer.tc(fn ->
    1..n |>
    Enum.map(fn x -> Action.send(pid, x)
    end)
end)
```

We can use the *:timer* module from Erlang. We can pass an expression to it. In our case it is an enumeration from *1..n* (with n means the max number of messages per measurement) (see Figure 6.7). For each element in the enumeration a data tuple with value x is sent to the cell with the process id *pid*.

In the same way we can use the approach for scaling-out measurements. A set of connected cells are defining a pipeline. The question we want to answer is: How does the PIPES middleware scale over the number of connected cells?

We measure this by building a pipeline and vary the number of cells within a pipeline. The number starts with 3 going up to 100. Each cell is running the following expression

```
expr = fn ->
  Source.asStream(name) |>
  Stream.map(fn x -> x+1 end) |>
  Action.forward(toPid)
end
```

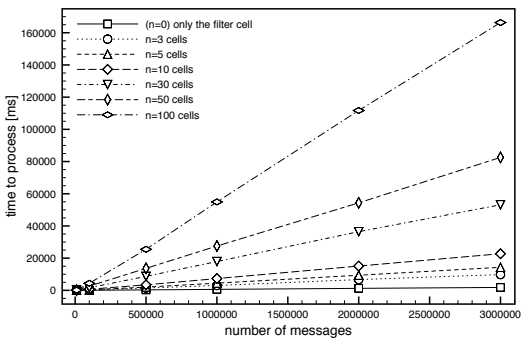
We measure the performance over a different number of messages (10,000 up to 3,000,000). The last cell is using the same *expr* as above. It filters until the maximum number is reached before it prints the result on standard output to verify the full pipeline was processed. In case we have a pipeline with 3 cells with 10000 messages we expect 10003 for example as output.

The evaluation setup is executed on four different machines:

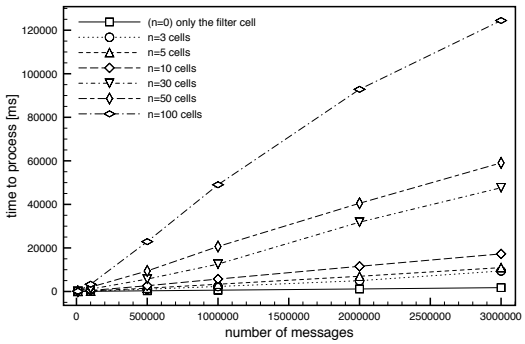
- Mac mini, with Intel Core i5 (2.3 GHz) and 16GB of Ram running Elixir1.0.4 on Erlang 17.0 (64bit) on OSX
- Macbook pro, with Intel Core i7 (3 GHz) and 8GB of Ram running Elixir1.0.4 on Erlang 17.0 (64bit) on OSX
- Intel Xeon 2.8GHz and 12GB of Ram running Elixir1.0.5 on Erlang 18.0 on Windows 10 with 64bit
- Raspberry Pi 1 ModelB with 512MB of Ram running Elixir1.0.3 on Erlang on Linux

The results are shown in figure 6.8.

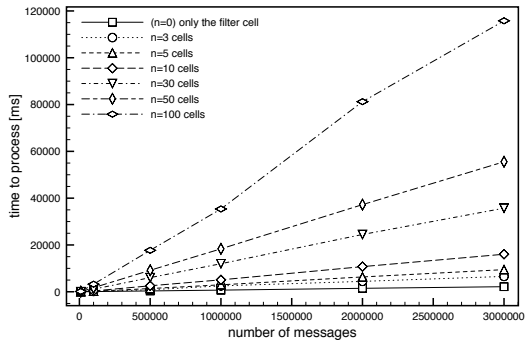
On all computers the systems scales linear over the number of cells and over the number of messages. We can calculate an average throughput of $\tilde{8}$00,000 messages/sec. based on one single cell. This result is close to what we wanted to reach (remember: in the hypothesis we wanted to reach $\tilde{1}$,000,000 messages/sec.). This result shows that we can talk about *fast processing* of data streams.



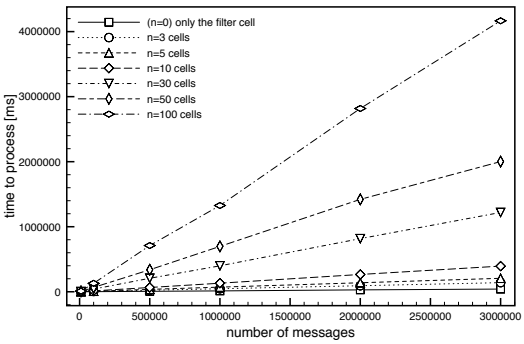
(a) Mac mini



(b) Macbook Pro



(c) Xeon



(d) Raspberry Pi

Figure 6.8: Measurement of cell scalability over different computer

So far we have measured the processing time in terms of scale. Another interesting parameter is the size of a message. We have repeated the measurement on two computers

- Mac mini, with Intel Core i5 (2.3 GHz) and 16GB of Ram running Elixir1.0.4 on Erlang 17.0 (64bit) on OSX
- Macbook pro, with Intel Core i7 (3 GHz) and 8GB of Ram running Elixir1.0.4 on Erlang 17.0 (64bit) on OSX

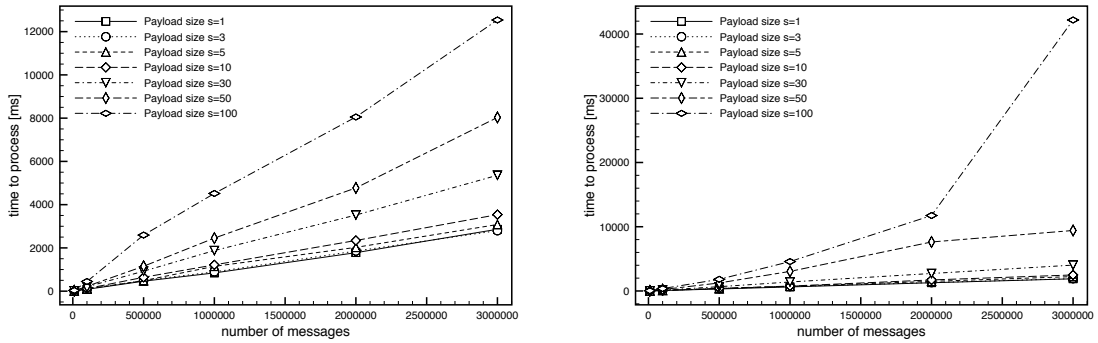
We have repeated each measurement 10 times and calculated the average and standard deviation of 10 measurements. We measured with payload sizes of 1,3,5,10,30,50, and 100 values. This means a message was basically a tuple of the form {*1: 1, n: n*} where the first value *1: _* is naming the property and the value behind the colon represents the value. This tuple can be created with a single line of code where m defines the size of the tuple:

```
message =
    1..m |>
    Enum.reduce(%{},fn (x,acc) -> Map.put(acc,x,x) end)
```

The message was sent to the cell using the forward action and the time was measured:

```
result = :timer.tc(fn ->
    1..n |>
    Enum.map(fn x ->
        Action.send(pid, Map.put(message,1, x))
    end)
end)
```

The number n of messages varied from 10,000 to 3,000,000 messages. The resulting time was measured in milliseconds (ms) (see Figure 6.9).



(a) Mac mini **(b)** Macbook Pro

Figure 6.9: Scalability depending on message size over different computer

The results show that the system scales linear over message size and number of messages. There is a trend that with larger messages the time will increase more then linear. But the number of more then 100 properties per message is not a very common use case. Thus, considering an average number of 10 properties per message the scalability looks very promising over the entire spectrum.

6.5 Example revisited

In the introduction we have mentioned a few scenarios to motivate the work. To validate the approach we will implement one example end to end. We pick the fleet management. Since we do not have an entire taxi fleet available we will simulate taxis as single processes within an Elixir/Erlang runtime on a Raspberry Pi.

In section 6.4, we haven proven that the PIPES concept does linear scalable in terms of length of a pipeline and in terms of message size. In general we have also proven that the expressions over streams can produce results with very low latency. The same applies to the general concepts of having a mediator and introduce filtering at the source. Therefore, the implementation of the fleet management is bringing all pieces together.

The setup of the fleet management simulation is based on three Raspberry Pis

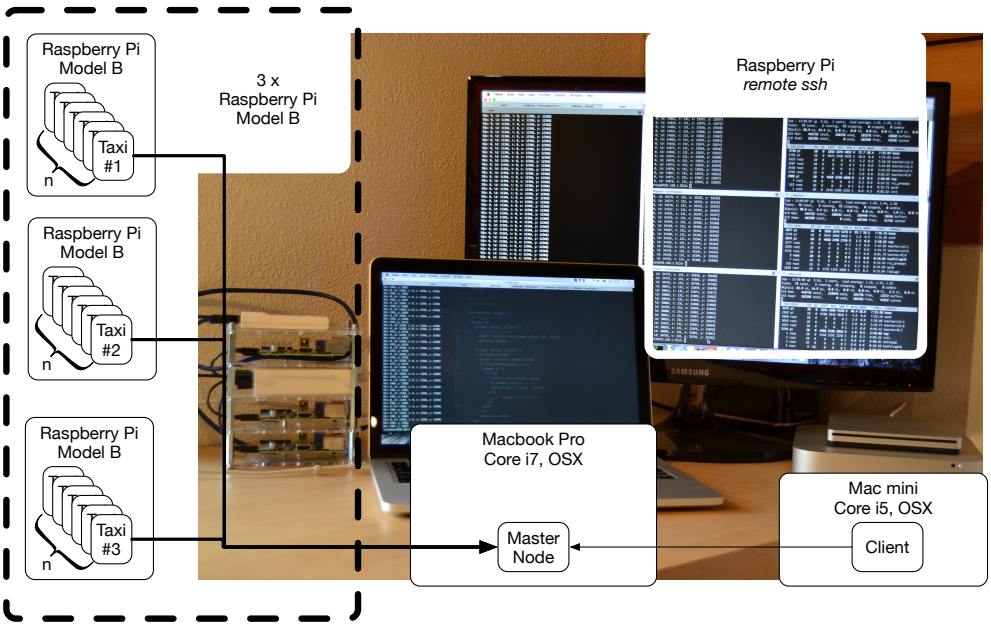


Figure 6.10: Fleet management simulation

Model B (each ARM v6, 512 MB Ram), a Macbook Pro with a Core i7 (Duo Core, 3.0 GHz, 8GB Ram) as a server backend, and a Mac mini with Core i5 (Duo Core) used a requesting client. The Core i7 runs the master node (*mediator*) receiving the messages from the taxi processes. These processes are running on the Pis (see Figure 6.10). Each Pi is running n taxi processes (n: 100, 1000, 5000, 10000) within one single node.

Using the PIPES API it is quite straightforward to implement the relevant parts. A taxi can be simulated like follows:

```
def positionDataAsStream(id, x, y, interval) do
  S.repeat( %{id: id, a: 0, x: x, y: y},
      &(%{a: avail(id),x: move(&1[:x]),y: move(&1[:y])})) |>
  S.zip(S.interval(interval)) |>
  S.map(fn {t,v} -> %{id: id, a: v.a , x: v.x, y: v.y, t: t} end)
end
```

The parameter *id* is used to identify the taxi, x and y representing the start

positions on a 2-dimensional grid and the interval is defining when a new event gets triggered. The first line uses the repeat method and simply creates an infinite stream and outputs event tuples of form {id, x, y, a} (with a representing the availability of the taxi; a=0 taxi is available and a=1 taxi is not available). The *move* method changes the values x or y by a random value between -1 and 3. This stream is zipped with a clock stream producing an output every *interval* (in milliseconds). The results of the zip method is a joined stream which outputs a time stamped position data tuple {id, x, y, a, t}. Now, we can inject a rule on each taxi, which forwards the new position only if the distance is greater then 20. Here is the number 20 an arbitrary number and is usually defined by the use case. The rule looks as follows:

```
def rule(stream) do
  stream |>
  S.state(%{x: -1, y: -1, a: 1},
      fn (v,acc) -> distance(v,acc) > 20 end)
end
```

The method expects a stream and detects if there is a new state. A new state is defined by the closure used within the operator. If there is a new state available the action forwards to the master.

Alternatively, we can define a rule which forwards the average position over a time window. In this example the time window has the size of 5 seconds and starts accumulating every 5 seconds (hop size):

```
def rule(stream) do
  stream |>
  S.win(5, 5, fn (elems) -> average(elems) end)
end
```

The *average* methods takes a list of tuple (*elems*) and calculates the average position of x and y. As a timestamp t we output the last timestamp of the list.

Same applies for availability *a*. The structure of the output tuple is the same as above {id, x, y, a, t}.

In both cases, with state method or time window, the rule outputs a results, which can be forwarded to the master. Therefore, a taxi cell can be implemented as follows (with id is a taxi identifier and master defines the master cell id):

```
def startTaxi(id, master) do
  Cell.start (fn ->
    positionDataAsStream(id, rnd(1000), rnd(1000), 1000) |>
    rule |>
    Action.forward(master)
  end)
end
```

The master node is implemented as a single node. First we create an *ActionState*, which can keep changes in a dictionary and exposes an endpoint for clients to request the available taxi for a given taxi. The closure within the *ActionState* expects a stream of taxi updates and the client position v. First the closure filters for available taxis and secondly returns the taxi which has the minimal distance to the given position.

```
def startMaster do
  # create action state endpoint
  # provide standing query
  endpoint = ActionState.start(fn (stream,v) ->
    stream |>
    S.filter(fn x -> x.a !=1) end) |>
    S.min_by(fn x -> distance(x,v) end)
  end)
  # cell expression
  expr = fn ->
```

```
      Source.asStream(:master) |>
      Action.keepState(endpoint)
    end
    # start master cell
    master = Cell.start(expr, :master, 1000)
    Registry.register_name(:endpoint, endpoint)
    Registry.register_name(:master, master)
    {:ok, endpoint, master}
  end
```

The expression *expr* running in the master cell is quite simple. On the source side it expects incoming data and forwards to the Action, which can keep the state. Finally, the cell gets started with the given expression expr and name *:master*. The *startMasterNode* method has to return the master PID and the endpoint PID so that taxis can forward to the master and clients can request results from the endpoint.

Finally, the client is simply implemented as an Elixir process sends a message to the master endpoint. The endpoint sends back the result as a message to the client. This can also implemented as a request-response over http (if required) without changing the core logic.

We have measured the memory consumption on the RaspberryPis to validate scalability on smaller devices. Furthermore, this proofs the cell processes can scale up millions without impacting the memory too much, This can be considered as *lightweight* processes. The result is shown in figure 6.11.

As a summary we can say that it is straightforward to implement the fleet management with the PIPES middleware even in a distributed setup. It is also very impressive to see how lightweight a cell process is. We can run 10,000 cells on a Raspberry Pi and we can run more than 1,000,000 cells on a Core i7 with enough memory. Since it scales very linear this middleware can easily be used to hyper-scale systems from smaller devices to the cloud.

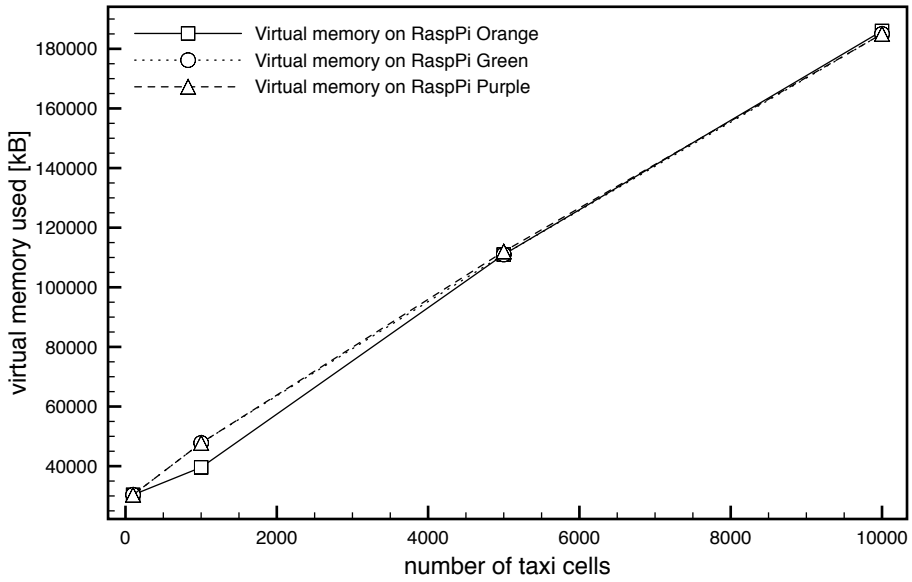


Figure 6.11: Used virtual memory on RaspberryPis

6.6 Comparison with other Technologies

So far, we have seen how PIPES behaves in data processing speed, scale over multiple nodes and mediation. There are some other technologies which are comparable to the PIPES middleware. For comparison we have picked the most common: (1) *Storm* is definitely the best know technology for distributed data processing, (2) *Akka* is well-known as distributed actor platform , and (3) *Cloud Haskell* as the functional programming platform which is close to PIPES concepts.

Storm (`https://storm.apache.org`) is a distributed, real-time computation system. On a *Storm* cluster, you execute topologies, which process streams of tuples (data). Each topology is a graph consisting of spouts (which produce tuples) and bolts (which transform tuples). *Storm* takes care of cluster communication, fail-over and distributing topologies across cluster nodes. *Storm* is created by Twitter and open sourced in 2011. It is written in Clojure and Java, but it works well with Scala. It is well suited for doing statistics and analytics on massive streams of data. *Storm* can describe streaming computation very simply: You make a graph of computation with some input data source called spouts at the

top, below that computation nodes called bolts that can depend on any spout or bolt that has been computed above it, but you cannot have cycles. The graph is called a topology.

Akka (`http://www.akka.io/`) is a toolkit for building distributed, concurrent, fault-tolerant applications. In an *Akka* application, the basic construct is an actor; actors process messages asynchronously, and each actor instance is guaranteed to be run using at most one thread at a time, making concurrency much easier. Actors can also be deployed remotely. Theres a clustering module coming, which will handle automatic fail-over and distribution of actors across cluster nodes.

As described in (EBPJ12), *Cloud Haskell* (`http://haskell-distributed.github.io/`) is a domain-specific language for cloud computing, implements as a shallow embedding in Haskell. A programmer can make use of massage-passing model, as we can find it in *Akka* or Erlang, but with some Haskell specific benefits, e.g. purity, types, and monads. The term cloud in *Cloud Haskell* refers to a large number of processors, with separate memories that are connected by a network and have independent failure nodes. In *Cloud Haskell* the core concept are lightweight processes (like actors) which communicate through typed channels.

The PIPES system can be seen as bringing the best of all worlds. In PIPES the core concepts are sources, conditions and actions which builds together an expression. The expression represents the logic and can be executed by a cell. Compared to *Storm* a cell could be a spout or a bolt or both and provides a more flexible to create a topology. A cell itself spawns lightweight processes, like in *Akka* or *Cloud Haskell*. Therefore, it is natural to create many cells if needed to process data. Since a cell can hold source, condition and action, PIPES provides a very flexible way to distribute data processing expressions within a topology. Thus, parallelising and scaling becomes very natural in PIPES. As shown, PIPES is able to handle large amounts of data, scales very nicely and distribution works out-of-the box.

6.6.1 Core concepts

Firstly, the basic unit of data in *Storm* is a tuple. A tuple can have any number of elements, and each tuple element can be any object, as long as there is a serialiser

available for it. In *Akka*, the basic unit is a message, which can be any object, but it should be serialisable as well (for sending it to remote actors). In *Cloud Haskell* the unit of data is a type (deriving from Typeable) and must be serialisable via a channel. The type safety of Cloud Haskell makes it very robust for developing system while concepts in Storm and Akka are more flexible.

In PIPES messages are tuples of values. A tuple is serialisable by the systems and makes distribution and message passing easy and out of the box. The complexity of a tuple is not limited and it is transparent if the message is processed locally or forwarded to a remote cell. The missing type information is limiting the robustness of a pipeline. This is work for future research.

In *Storm*, we have components: bolts and sprouts. A bolt can be any piece of code, which does arbitrary processing on the incoming tuples. It can also store some mutable data, e.g. to accumulate results. Moreover, bolts run in a single thread, so unless you start additional threads in your bolts, you dont have to worry about concurrent access to the bolts data. In *Akka* the core concept are actors. Each actor is a lightweight process (not a system process). Each actor can receive messages and maintains its state. This is comparable to *Cloud Haskell* where we can find lightweight processes which communicate through type channels.

A PIPES cell is comparable to an actor in *Akka* or a process in *Cloud Haskell* although it can behave like a bolt since a cell runs a piece of code (expression) to process incoming messages (tuples) and can store mutable data if required.

The major difference between actors and bolts is how they communicate. An actor can receive messages and can send messages to any other actor, as long as it has the reference to this actor (by looking it up in a registry). It can also send back a reply to the sender of the message that is being handled. *Storm*, on the other hand is one-way. You cannot send back messages; you also cannot send messages to arbitrary bolts. But you can send a tuple to a named channel (stream), which will cause the tuple (message) to be broadcast to all listeners, defined in the topology. As a result, the topology in *Storm* is well defined and stable while the topology in *Akka* and *Cloud Haskell* is more fragile and ad-hoc.

PIPES uses messages to communicate between cells and can even send messages back, although this can lead to some strange behaviour. By using message

passing the topology in PIPES is also very flexible and can be changed quite easily. The only dependency is the cell id to which a message should be sent.

In *Storm*, multiple copies of a bolts/sprouts code can be run in parallel (depending on the parallelism setting). So this corresponds to a set of (potentially remote) actors, with a load-balancer actor in front of them.

PIPES is not limiting any parallel execution of cells. The routing or load-balancer would be a cell with specific logic taking care for parallelising the message passing to other cells. To union results another cell is required (see fleet management example).

6.6.2 Size and Scalability

There is also a difference in the weight of a bolt, an actor, and lightweight process. In *Akka*, it is normal to have lots of actors (up to millions). The same applies to *Cloud Haskell*. In *Storm*, the expected number of bolts is significantly smaller; this is not in any case a downside of *Storm*, but rather a design decision. Also, *Akka* actors typically share threads, while each bolt instance tends to have a dedicated thread. Dedicated threads usually share memory to exchange data. In *Cloud Haskell* both message passing and shared memory is possible.

PIPES is comparable to *Akka*. PIPES is expecting to have a big number of cells where each cell is able to process a huge number of messages.

Akka is better for actors that talk back and forth, but you have to keep track the actors, and make strategies for setting up different actor systems on different servers and make asynchronous request to those actor systems. *Akka* is more flexible than *Storm* but there is also more to keep track of. *Storm* is for computations that move from upstream sources to different downstream sinks. It is very simple to set this up in *Storm* so it run computation over many distributed servers.

The following table summarises the comparison and provides an overview:

	Storm	**Akka**	**Cloud Haskell**	**PIPES**
Platform	Java/ Scala	Java/Scala	Haskell	Elixir/Erlang
Programming model	OO	OO	functional	functional
Data	tuple	message	typeable	typed tuple
Communication	streams	message passing	typed channel	message passing
Distribution	stouts / bolts	actor	process	cells
Hot deployment	–	–	–	Code redeployment during runtime
Data stream processing	Streams (filter, map, join,)	–	–	conditions (filter, map, zip,)
Temporal concepts	windows	–	–	clock stream (with len and win)

To summarise, PIPES provides a complete set of features for distributed data processing. It brings together the best of all worlds: It provides an easy way to describe and combine cells, enables the user to describe the pipeline for event stream processing, and is fast enough to process data in a scalable fashion.

6.6.3 Event Processing Comparison

From a platform perspective PIPES can offer already a sufficient set of features. Processing of expressions within a cell can be compared with complex event processing. Therefore it makes sense to revisit engine and language features of section A.

The following table provides an overview of PIPES compared to CEP engine features:

Engine Feature	
Deployment model	clustered
Manageability	limited
Monitoring	via Erlang observer
Capacity management	–
Debugger	–

Most products provided a more complete set of engine features. Most of them are manageable and offers capacity management. Some of them monitoring and debugging features. Here, PIPES can be improved is not on the level of available products.

The following table provides an overview of PIPES compared to CEP language features:

Language Feature	
Type	functional
Time mode	world clock stream
Single Item Operators	selection, projection, mapping
Logic Operators	not yet
Windows	sliding, hopping, and tumbling, count, user defined
Flow Management	join
Others	UDF, UDA, UDO

Main language features are available in PIPES. Although PIPES is lacking some operators, we were focused so were on providing operators which were required by motivating scenarios. Furthermore, it was not in focus to build a platform with a complete operator list. The focus was more on providing a foundation which is strong enough to fulfil requirements and to provide a coherent foundation which can be validated via ITL to gain a more or less formal proof. In this sense, PIPES offers a foundation based on formal logic which is not available with any other technology (as far as we are aware).

However, PIPES is a complex concept combining event processing with highly scalable and distributed architecture which is new. We believes this is a very useful combination for scenarios introduced in the introduction of the this thesis.

6.7 Summary

In this chapter we have evaluated the approach presented within this thesis regarding (1) the overall approach for continuous data processing, (2) the latency of push and pull based architectures towards a hyper-scale setup, (3) the capabilities of the filtering at the source concept in terms of data traffic, and (4) building distributed, fast processing data pipelines.

We have shown that using streams for data processing is a useful concept in terms of latency and processing speed. We have demonstrated that complexity of pipelines only matters for specific operators.

Overall, both evaluation tests regarding mediation architectures highlight how (1) using the push approach with mediator and (2) policy injection on the data source can be combined to form a promising architecture supporting low-latency for large systems.

We have validated the applicability of the approach for real-world scenarios by proving that building a pipeline end-2-end is (1) possible and scales in terms of throughput and number of cells and (2) it provides a valid foundation for writing distributed applications in an easy and natural way.

Chapter 7

Conclusion

"Success is not final, failure is not fatal: it is the courage to continue that counts."

– Winston S. Churchill

SOA and big data are two of the mega-trends of the last decade. When talking about big data it is usually linked to the 3 Vs as stated by Gartner (Gar11): Volume, Variety, and Velocity. But the common understanding of big data was mainly associated with volume and variety only. Velocity represents the speed data at which needs to be processed. Nowadays, this is getting more and more important driven on one the side by the pure speed or latency in which insights can be produced and on the other side by the pure amount of data. The expected data produced by sensors, phones, devices, etc. is expected to be so big that all raw data cannot or may not be stored. Thus, it needs to be processed on the fly with almost no latency. Google and others have started to use the term *fast data* for this.

In addition SOA is still there and still important. There are services everywhere, providing data and offering functionality, e.g. weather, climate, or traffic insights. Combining different data sources and services to produce better

and more specific insights and predicting behaviour or avoiding unplanned downtimes are becoming key in the next years. Reading machine data in real-time, correlating it with prediction models and triggering actions is one aspect in IoT scenarios.

In this thesis we have presented how to combine fast data processing with SOA. The overall aims of our research are (1) to develop a model to process temporal service data with near zero latency, (2) to enable data processing in highly-distributed and large systems, and (3) to develop an architecture to mediate between different interaction patterns.

In this chapter, we will discuss our research contributions by reflecting on the research aims in Section 7.1 and considering future research directions in Section 7.2. Finally the concluding remarks will be made in Section 7.3.

7.1 Research Contribution

We divided our research aims into three aspects and each aspect comes with several sub goals and objectives. We will now discuss our research contributions against each of them.

7.1.1 Model for Fast Data Processing of Temporal Service Data

A language to describe data processing pipelines from sources to actions. The main focus of the syntax is on expressing transformations of input data and trigger actions on the output side. We have introduced a concept called *event policies* to enable rules over data streams. These data streams were considered to follow the ITL logic. The *event policy* is a kind of ECA rule or a *source-condition-action* rule (see section 3.2). By defining a source-condition-action paradigm and verfiying this against ITL we have provided a foundation for fast data processing over continuous data streams.

Support of time dependent service offerings. We have extended the current logic with time information by introducing the concept of a clock stream.

This additional time information is building the grounding to process time dependent service offerings. With SCA over time rules it is possible to formulate time related rules over data streams.

7.1.2 Data Processing in highly-distributed Systems

.

Defined **an architecture to enable a distributed pipeline setup over a heterogeneous set of nodes**.We have extended the *event policy* paradigm towards support of distributed systems. Polices can be distributed to nodes and executed. The distribution concept is aligned with the policy concept. A policy consists of a source-condition-action pattern. Thus, this can be easily distributed through a complex distributed setup. Sending policies around as extended actors enables us to support highly-distributed and large setups. The footprint by using an actor model is very small so that we can easily scale to a big number of policies and pipelines.

7.1.3 Mediation Architecture for Distributed Systems

A mediation model to integrate pull and push interaction into one coherent architecture. We can find a lot of SOA based systems. Most of them are using the classical request-response interaction paradigm. In contrast event-based systems are getting popular as well; especially in some distributed systems. We have developed an interaction paradigm which easily enables us to integrate heterogeneous interaction pattern. Starting by combining push and pull based interaction helps us to integrate request-response interaction with event-based systems.

The mediation pattern enables us to control event-flow very elegantly by using event policies and filtering at the source. This pattern extends the existing pull- and push-model to have insights at your fingertip. There is no need to scan through the list of services. Data is there when required.

7.2 Future Research Directions

The programming model for fast data processing, the PIPES middleware, and the mediation architecture are supporting the convergence of SOA and event-based systems in large-scale. Although we have successfully achieved our research aims, there are some remaining and arising research issue and directions in fast data research.

Internet of Things: In the *Internet of Things* we are expecting a lot of new requirements and challenges in the next years. The integration of different sources and the extraction of insights will create new scenarios and services. The demand for expressing rules over on-the-fly data will increase dramatically. The pure amount of data will require a very flexible model to change rules on-the-fly and for a highly distributed setup. Managing the amount of data, the optimal usage of bandwidth or simply expressing rules depending on context will become a key challenge for IoT. The expressiveness of rules needs to be increased and extended. In addition, the requirement of placing rules as close to the source as possible requires us to think about how to build engines which are able to execute event policies on really small devices, e.g. sensors. More operators which makes the definition of rules easier are desirable. Ideally a user should be able to read rules as plain English.

Factory of the Future: In factory of the future (or Industry 4.0) each work piece will be connected through its entire life time and deliver telemetry data. Each work piece is becoming a smart object. These object are carrying unique information to be identifiable, to communicate with equipment and machines to tell them how a piece needs to be processed and in production it will tell about its status, maintenance requirements, etc. Future research could investigate how presented work can be used for smart objects. Which data can be processed on really small processors and which needs to be send as aggregated data to the cloud? This work provides some grounding, but there are challenges about connectivity, about data in general, and about device capabilities at the horizon which are not covered.

Predicting behaviour. So far, we defined operators only for processing actual data to provide insights which are very timely. We have not looked into

models which can predict certain behaviour. It would be nice to detect patterns on a data stream and predict problems early. Fro example, we would know there is a characteristic spike in the data stream which is caused by a problem which will lead to a problem. It would be nice to learn this correlation and express it as an *event policy*. The concept of event policies which we have presented here would basically support it. There is currently a missing link between the learning phase and expressing the policy. Future research could investigate how event policies could be generated based on historical data in a first step and in a second step deriving event policies dynamically while processing data streams.

Enabling event policy evolution. The current PIPES system supports the ability to update event policies during runtime. This does not mean that states from one version of the policy migrated to the new version of a policy. So far, we have not looked into any kind of event policy versioning and evolution. Future research could investigate into evolution concepts of event policies. There might be changes to policies which do not break policy logic so that they could be considered as a next evolution step of a policy. This work would also link to the automatic adaption of prediction policies during runtime.

7.3 Concluding Remarks

It is not foreseeable when the dramatic increase of data sources producing trillions of data points a second will stop. The amount of available data is directly linked to the demand of processing this data and getting use out of it. This requires concepts on how to handle this data (1) in terms of expressing rules, (2) in terms of deciding which part of the rule can run where and (3) to control the data flow without reducing the amount of information extracted from data. This thesis has looked into all three aspects and has delivered a coherent concept for processing data, distributing logic and integrating it into existing systems. This is clearly only the beginning for a whole new set of application and research for hyper-scale systems. We hope this thesis can make some contribution for the future to help overcome the data flow without losing too much privacy.

Appendix A

Platforms for CEP

In order to conduct this review of the state of the art, we have based our analysis on different sources. First, we have identified 2 existing surveys from which some of our results are based ((CM12), (FTR+10)). Second, we identified the market leaders in the field of CEP based on a Forrester report (GR09). Finally, we completed the analysis by gathering additional information more recent than the previously mentioned documents.

A.0.1 Selection Criteria

The projects and products discussed in this section had to satisfy the following criteria in order to appear.

- Availability of sources or binaries: in particular, research paper without an available platform were not considered
- Availability of documentation: some commercial products provide documentation and support only after buying the product. Those products are mentioned and references are given, but they are not discussed.

We list all the CEP products/projects satisfying the criteria:

AMIT

IBM Active Middleware Technology (AMIT) is a lightweight complex event processing engine that has been developed in IBM Research - Haifa. AMIT is now commercially available as a WebSphere family product extension and should be considered part of the WebSphere Message Broker. The latest version has been released in 2008 and it seems that IBM is now promoting WebSphere Business Events as its WebSphere CEP processing.

Borealis

Borealis is a research project developed at Brandeis University, Brown University, and MIT. It is a distributed stream processing engine building on previous efforts of the team in the area of stream processing: Aurora and Medusa. The Borealis project is no longer an active research project and its latest release dates back from 2008 and is available under an ad-hoc public license. The Borealis software runs on Linux x86-based computers.

Cayuga

Cayuga is an open-source expressive and scalable Complex Event Processing (CEP) system developed at the Cornell Database Group. The system is implemented in C++, and runs on Windows, Linux, and Mac OS X. The code is available on SourceForge under a BSD license.

Continuous Analytics

The Continuous Analytics software system (9) from Truviso is a hybrid of real-time stream processing and relational dabase management system (RDBMS). It resulted from the commercialization of the TelegraphCQ tool. A data stream query processor has been integrated as a component of the SQL query processor inside PostgreSQL.

Esper/Nesper

Esper is an open source event stream processing (ESP) and event correlation engine (CEP). A commercial version is also available with additional features such as high availability and which includes support. Esper is a Java-based whereas Nesper is the equivalent CLR-based engine.

Event Processing Network

EventZero (12) is Brisbane-based Independent Software Vendor (ISV) founded in 2005. The Event Zero product suite is a full-featured commercial Event Processing Network (EPN) platform used to implement many different types of event processing solutions. Tehir EPN is a fully integrated suite of products that provides functionality for all of the stages in the event processing lifecycle: capture, process, and respond.

Infosphere Streams

InfoSphere Streams is a CEP platform commercialized by IBM. It consists of a programming language and an integrated development environment (IDE) for Streams applications, and a runtime system that can execute the applications on a single or distributed set of hosts. The Streams Studio IDE includes tools for authoring and creating visual representations of the Streams applications.

Oracle CEP

Oracle launched its commercial event-driven architecture suite in 2006 and added BEA's WebLogic Event Server, itself based on Esper core engine, to it in 2008, building what is now called "Oracle CEP", a system that provides real time information flow processing. Oracle CEP is a lightweight Java application container based on Equinox OSGi. Oracle CEP uses CQL as its rule definition language which allows the manipulation of Streams and Relations.

PADRES

Publish/Subscribe Applied to Distributed Resource Scheduling (PADRES) is an open-source project. It is a distributed enterprise-grade event management infrastructure developed in the Middleware Systems Research Group at University of Toronto. PADRES is based on the publish/subscribe event notification model.

Progress Apama

Progress Apama is a commercial product resulting from the acquisition of Apama inc by Progress software corporation in 2005. It is one of the market leaders in the field of CEP and provides a set of tools for creating, deploying and monitoring CEP applications.

RTM Analyzer

RTM Analyzer (17) is commercialized by RTM Realtime Monitoring gmbh which is a full subsidiary of Software AG. It is modular CEP platform originating from the results obtained in the PIPES research project headed by B. Seeger.

Rulecore CEP Server

Rulecore CEP server is a commercial tool provided by Rulecore. It is entirely event-driven and its execution engine works by continuously evaluating reaction rules in response to inbound events. The rules are defined using a declarative XML based rule language.

RulePoint

RulePoint is a commercial CEP software, initially developed by Agent Logic, which was then bought by Informatica in 2009.

SASE

Stream-based And Shared Event processing (SASE) is a research project developed at the University of Massachusetts. Their focus lies in pattern matching in

event streams with a formal approach based on the query evaluation model relying on non-deterministic automata with buffer. The initial query model SASE has been extended in SASE+ to incorporate Kleen closure.

Smart Enterprise Platform

Starview Smart Enterprise Platform is a commercial system for developing and running event-driven applications. It provides a comprehensive environment of event servers, remote agents (lighter version of event servers), monitoring and simulation servers.

STREAM

The Stanford Stream Data Manager (STREAM) is a research project that has been actively developed in Stanford University until 2004. A distinguishing feature of the language is the use of two elements, streams and relations, having operators on each types as well as operators transforming on type into the other. The project served as testbed for different researches in the context of stream processing including the query language, the query processing, and the system as a whole.

Stream Mill

Stream Mill is a research project conducted at the Web Information Systems Laboratory of UCLA. Research efforts have been targeted at the query model and language, optimization techniques, efficient support for XML as well as mining a continuous stream of data.

Streambase

StreamBase Systems was founded in 2003 by Dr Mike Stonebraker, building on the efforts from M.I.T, Brown University and Brandeis University in the Aurora project. Streambase's Event Processing Platform is a commercial product combining a graphical event-flow development environment, a high troughput event

server and a broad connectivity to real-time and historical data. Client libraries are available for Java, C++, .Net and Python.

StreamInsight

Microsoft StreamInsight is a commercial platform for stream query processing based on the Microsoft Research project called CEDR. Streaminsight embraces a temporal stream model to unify and further enrich query language features, handle imperfections in event delivery and define consistency guarantees on the output.

Sybase Aleri Streaming Platform

The Sybase Aleri Streaming platform is a commercial tool resulting of the merger of Coral8, Aleri and Sybase, which was then later bought by SAP. The platform provides an IDE called Aleri Studio based on the Eclipse framework for application development. It also provides multiple programming interfaces to communicate with the platform as well a a large set of utilities.

TelegraphCQ

Telegraph is a research project in UC Berkeley's Computer Science Division. They core research evolves around technologies for adaptive dataflow. In particular, they have introduced techniques to reshape dataflow graphs to maximize performance and to do load balancing across multiple machines on a network.

Tibco BusinessEvents

Tibco Business Events is a commercial product using a model-driven approach to collect, filter, and correlate events. Its execution engine is a RETE-Based rules engine and can be distributed across a multiple computers.

UC4 Decision

UC4 Decision is a commercial product resulting from the acquisition of Senactive by UC4 in 2009. UC4 Decision is a complex-event processing tool that processes events based on pre-defined process models.

Vantify

Built on CEP technology, WestGlobal's Vantify Experience Centre is a commercial real-time Business Activity Monitoring solution.

Websphere Business Events

Websphere Business Events is a commercial product part of the Websphere line of business offered by IBM. The technology originates from the acquisition of the Aptsoft company in 2008. The tool allows to define complex event processing more from a workflow perspective, and a distinguishing feature is the aim to target directly the business analysts rather than deep technologists and programmers, and therefore a set of easy to use tools with a clear view of the business processes is part of the Websphere Business Events offering.

	Tools	Type	Time mode	Single Item Operators			Logic Operators					Windows										Flow Management														
				Selection	Projection	Mapping	Conjunxtion	Disjunction	Negation	Sequence	Iteration	Fixed	Landmark	Count Sliding	Count tumbling	Count Hopping	Time Sliding	Time Tumbling	Time Hopping	User Defined	Join	Union	Except	Intersect	Group By	Order By	Flow Creation	UDF	UDA	UDO	Statistics	Probability	Predictability	
	AMIT	Det	Abs	X			X	X	X	X	X	X	X		X		X			X														
	Borealis	Imp	Stream	X	X								X	X	X		X	X																
	Cayuga	Decl	Interv	X	X	X				X	X										X					X								
	Esper/Nesper	Decl + Det	Stream	X	X	X	X	X	X	X	X			X	X		X	X			X	X		X	X	X	X	X		X				
	Infosphere Streams	Decl	Stream	X	X	X								X	X	X	X	X	X		X	X	X	X	X	X	X	X						
	Oracle CEP	Decl	Stream	X	X	X						X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X		X				
	PADRES	Decl + Det	Abs	X			X	X		X	X	X	X	X	X																			
	Rulecore CEP server	Det	Abs	X	X	X	X	X	X	X	X	X	X		X			X		X							X							
	SASE	Det	Abs	X	X		X	X	X	X							X																	
	Smart Enterprise Platform	Imp	Abs	X	X	X				X				X			X				X													
	STREAM	Decl	Stream	X	X	X								X			X				X	X	X	X		X								
	Stream Mill	Decl	Stream	X	X	X						X	X	X	X	X	X	X	X		X	X	X	X		X	X							
	Streambase	Decl	Stream	X	X	X								X	X	X	X	X	X		X	X			X	X	X	X	X					
	Streaminsight	Decl	Interv	X	X	X								X				X	X		X	X		X	X	X	X	X					
	Sybase Aleri Streaming Platform	Decl	Stream	X	X	X				X				X			X				X	X		X	X	X		X					
	TelegraphCQ	Decl	Abs	X	X	X															X			X	X								
	Websphere Business Events	Det		X	X	X	X	X	X	X		X		X			X				X												

Figure A.1: Language features supported by the tools

References

[ACÇ+03] D Abadi, D Carney, U Çetintemel, M Cherniack, C Convey, C Erwin, E Galvez, M Hatoun, A Maskey, A Rasin, A Singer, M Stonebraker, N Tatbul, Y Xing, R Yan, S Zdonik, U Cetintemel, and M. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, page 666, New York, NY, USA, 2003. ACM. 27

[Arm97] Joe Armstrong. The development of Erlang, 1997. 85

[BCP02] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39:645–660, 2002. 27

[BGAH06] R.S. Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *Arxiv preprint cs/0612115*, pages 363–374, 2006. 49, 60, 105, 106

[BGK+11] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans : Cloud Computing for Everyone. *Proceedings of the 2nd ACM Symposium on Cloud Computing SOCC 11*, pages 1–14, 2011. 85, 88

[Bro09] Paul Browne. JBoss Drools Business Rules. *Birmingham UK Packt Publishing*, 2009. 36

[Bui09] Hai-Lam Bui. Survey and Comparison of Event Query Languages Using Practical Examples. *Citeseer*, (November 2008), 2009. 35

[CC12] Badrish Chandramouli and Joris Claessens. RACE: real-time applications over cloud-edge. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 625–628, 2012. 7

[Cho08] David Chou. Using events in highly distributed architectures. *The Architecture Journal*, (17):29–33, 2008. 98

[CK09] A Colombo and S Karnouskos. *Towards the factory of the future: A service-oriented cross-layer infrastructure*, page ch. 6. John Wiley and Sons Ed., 2009. 48

[CM94] S Chakravarthy and D Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, nov 1994. 26

[CM12] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: from data stream to complex event processing. *ACM Computing Surveys*, 44(3):1–62, 2012. 149

[CMZ11] A Cau, B Moszkowski, and H Zedan. The ITL homepage: http://www.cse.dmu.ac.uk/STRL/ITL. Technical report, Software Technology Research Laboratory, De Montfort University, 2011. 26, 28, 72

[DGH+06] A. Demers, J Gehrke, M Hong, M Riedewald, and W. White. Towards expressive publish/subscribe systems. *Advances in Database Technology-EDBT 2006*, pages 627–644, 2006. 27

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. pages 455–462, Washington, DC, 2004. IEEE Computer Society. 49

[DK04] Zhen-Hua Duan and Maciej Koutny. A framed temporal logic programming language. *J. Comput. Sci. Technol.*, 19(3):341–351, may 2004. 26

[EBPJ12] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud, 2012. 86, 88, 138

[Eck08] Michael Eckert. Complex event processing with XChangeEQ: Language design, formal semantics, and incremental evaluation for querying events. *Dissertation, Ludwig-Maximilians-University of Munich*, 2008. 35, 37

[ESPC08] Michael Edge, Pedro Sampaio, Oliver Philpott, and Mohammed Choudhary. A Policy Distribution Service for Proactive Fraud Management over Financial Data Streams. *Services Computing, IEEE International Conference on*, 2:31–38, 2008. 26

[Fai06] T Faison. *Event-Based Programming: Taking Events to the Limit*. APress, 2006. 42

[FH04] Ernest Friedman-Hill. Jess: The Rule Engine for the Java Platform, 2004. 36

[FM77] C Forgy and J McDermott. OPS, A Domain-Independent Production System Language. In *IJCAI*, volume 1, pages 933–939. Morgan Kaufmann Publishers Inc., 1977. 36

[For90] Charles L Forgy. Expert systems. In Peter G Raeth, editor, *Expert systems*, chapter Rete: a fa, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. 26, 56

[FTR+10] L.J. Fülöp, G. Tóth, R. Rácz, J. Pánczél, T. Gergely, A. Beszédes, and L. Farkas. Survey on Complex Event Processing and Predictive Analytics. *Networks*, 2010. 149

[Gar11] Gartner. Gartner Says Solving ' Big Data ' Challenge Involves More Than Just Managing Volumes of Data, 2011. 144

[GPMF06] G. Muehl L. Fiege, Peter Pietzuch, Gero Mühl, and Ludger Fiege. *Distributed Event-Based Systems*. Springer, New York, 2006. 32

[GR09] Mike Gualtieri and John R Rymer. The Forrester Wave: Complex Event Processing (CEP) Platforms, Q3 2009. Technical report, 2009. 149

[Hal88] Roger William Stephen Hale. *Programming in Temporal Logic*. PhD thesis, Trinity College, University of Cambridge, oct 1988. 72

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973. 85

[HMT+14] Robert Harrison, Stuart McLeod, Giacomo Tavola, Marco Taisch, Armando Walter Colombo, Stamatis Karnouskos, Marcel Tilly, Petr Stluka, François Jammes, Roberto Camp, Jerker Delsing, Jens Eliasson, and Joao Marco Mendes. Next Generation of Engineering Methods and Tools for SOA-Based Large-Scale and Distributed Process Applications. In Armando Walter Colombo, Thomas Bangemann, Stamatis Karnouskos, Jerker Delsing, Petr Stluka, Robert Harrison, François Jammes, and José Luis Martinez Lastra, editors, *Industrial Cloud-based Cyber-Physical Systems: The IMC-AESOP Approach*. Springer, may 2014. 22

[JCS+06] Helge Janicke, Antonio Cau, Fran\ccois Siewe, Hussein Zedan, and Kevin Jones. A Compositional Event & Time-based Policy Model. In *Proceedings of POLICY2006, London, Ontario, Canada*, pages 173–182, London, Ontario Canada, jun 2006. IEEE Computer Society. 56

[JCSZ07] Helge Janicke, Antonio Cau, Francois Siewe, and Hussein Zedan. Deriving Enforcement Mechanisms from Policies. In *Proceedings of the 8th IEEE international Workshop on Policies for Distributed Systems (POLICY2007)*, pages 161–170, jun 2007. 26

[JCSZ12] H Janicke, A Cau, F Siewe, and H Zedan. Dynamic Access Control Policies: Specification and Verification. *The Computer Journal*, 2012. 56

[JKB+14] François Jammes, Stamatis Karnouskos, Bernard Bony, Philippe Nappey, Armando Walter Colombo, Jerker Delsing, Jens Eliasson, Rumen Kyusakov, Petr Stluka, Marcel Tilly, and Thomas Bangemann. Promising Technologies for SOA-based Industrial Automation Systems. In Armando Walter Colombo, Thomas Bangemann, Stamatis Karnouskos, Jerker Delsing, Petr Stluka, Robert Harrison, François Jammes, and José Luis Martinez Lastra, editors, *Industrial Cloud-based Cyber-Physical Systems: The IMC-AESOP Approach*. Springer, may 2014. 22

[KCJ+10] S. Karnouskos, A. Colombo, F. Jammes, J. Delsing, and T. Bangemann. Towards an architecture for service-oriented process monitoring and control. 2010. 48

[KHJ+14] Stamatis Karnouskos, Vladimir Havlena, Eva Jerhotova, Petr Kodet, Marek Sikora, Petr Stluka, Pavel Trnka, and Marcel Tilly. Plant Energy Management. In Armando Walter Colombo, Thomas Bangemann, Stamatis Karnouskos, Jerker Delsing, Petr Stluka, Robert Harrison, François Jammes, and José Luis Martinez Lastra, editors, *Industrial Cloud-based Cyber-Physical Systems: The IMC-AESOP Approach*. Springer, may 2014. iii, 11, 12, 22

[Lar09] Jim Larson. Erlang for concurrent programming, 2009. 85

[LPM13] P. Lindgren, P. Pietrzak, and H. Muekitaavola. Real-time complex event processing using concurrent reactive objects. IEEE International Conference on Industrial Technology, 2013. 49

[LPTL02] Ling Liu, Calton Pu, Wei Tang, and Pu C Tang W Liu L. Continual queries for internet scale event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610–628, 2002. 105

[LS08] D. Luckham and Roy Schulte. Event processing glossary-version 1.1. *Event Processing Technical Society, Tech. Rep*, (July):1–19, 2008. 32

[Luc02] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman, Amsterdam, 2002. 32, 33, 105

[MAK07] R. Mueller, G. Alonso, and D. Kossmann. SwissQM: Next generation data processing in sensor networks. 2007. 49

[MD05] Colleen McClintock and Christian De Sainte Marie. ILOGs position on Rule Languages for Interoperability, 2005. 36

[MFHH12] S. R. Madden, M. J. Franklin, M. J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. pages vol. 20, no. 1, pp 122–173, 2012. 49

[Mos95] Ben Moszkowski. Compositional Reasoning about Projected and Infinite Time. In *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 238–245, Fort Lauderdale, Florida, 1995. IEEE Computer Society Press. 26, 28, 29

[MTJ14] Stephan Reiff Marganiec, Marcel Tilly, and Helge Janicke. Low-Latency Service Data Aggregation Using Policy Obligations. *2014 IEEE International Conference on Web Services*, pages 526–533, jun 2014. iii, 4

[PLM12] P. Pietrzak, P. Lindgren, and H. Muekitaavola. Towards a lightweight CEP engine for embedded systems. 38th Annual Conference of the IEEE Industrial Electronics Society, 2012. 49, 106

[RGR09] Haggai Roitman, Avigdor Gal, and Louiqa Raschid. Web Monitoring 2.0: Crossing Streams to Satisfy Complex Data Needs. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1215–1218. IEEE Computer Society, 2009. 27

[RL06] A Riabov and Z Liu. Scalable planning for distributed stream processing systems. In *Proceedings of ICAPS*, volume 06, 2006. 102

[RMYT09] Stephan Reiff-Marganiec, H. Yu, and Marcel Tilly. Service selection based on non-functional properties. In *Service-Oriented Computing-ICSOC 2007 Workshops*, volume 4907, pages 128–138. Springer, 2009. 27

[Slo94] M Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994. 26

[ST13] Ivo Santos and Marcel Tilly. DiAl: distributed streaming analytics anywhere, anytime. *Proceedings of the VLDB Endowment*, 6(12):1386–1389, 2013. iii, 10

[TD09] Cong Tian and Zhenhua Duan. Complexity of propositional projection temporal logic with star. *Mathematical. Structures in Comp. Sci.*, 19(1):73–100, feb 2009. 26

[TLDS08] K Twidle, E Lupu, N Dulay, and M Sloman. Ponder2 - A Policy Environment for Autonomous Pervasive Systems. In *Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on*, pages 245–246, jun 2008. 26, 56

[TLPY06] Y.C. C Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006. 27

[TRM11] Marcel Tilly and Stephan Reiff-Marganiec. Matching customer requests to service offerings in real-time. In *Proceedings of the 2011 ACM Symposium on Applied Computing - SAC '11*, page 456, New York, New York, USA, 2011. ACM Press. iii, 4, 98

[TRMJ12] Marcel Tilly, Stephan Reiff-Marganiec, and Helge Janicke. Efficient Data Processing for Large-Scale Cloud Services. In *2012 IEEE Eighth*

World Congress on Services, pages 242–250. IEEE, jun 2012. iii, 6, 98

[UBJ+03] A Uszok, J Bradshaw, R Jeffers, N Suri, P Hayes, M Breedy, L Bunch, M Johnson, S Kulkarni, and J Lott. KAoS: policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In *Proceedings POLICY 2003 Policies for Distributed Systems and Networks*, pages 93–96, jun 2003. 26, 56

[YRM08] H.Q. Q Yu and Stephan Reiff-Marganiec. Non-functional property based service selection: A survey and classification of approaches. In *Proc. of 2nd Non Functional Properties and Service Level Agreements in SOC Workshop (NFPSLASOC08)*. Citeseer, 2008. 27

[YZL07] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6—-es, 2007. 27