Antti Aalto

Scalability of Complex Event Processing as a part of a distributed Enterprise Service Bus

Degree Program of Computer Science and Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 19.11.2012

Thesis supervisor:

Prof. Heikki Saikkonen

Thesis instructor:

M.Sc. Atso Haapaniemi

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Antti Aalto

Title: Scalability of Complex Event Processing as a part of a distributed Enterprise Service Bus

Date: 19.11.2012 Language: English Number of pages: 9+66+7

School of Science

Degree Program of Computer Science and Engineering

Professorship: Software Technology Code: T-106

Supervisor: Prof. Heikki Saikkonen

Instructor: M.Sc. Atso Haapaniemi

Complex event processing (CEP) is an emerging technology to facilitate analysis and pattern matching in data streams and historical data. CEP can combine various types of data from multiple streams, databases and other sources. Several CEP engines are available but only few are both able to integrate seamlessly with multiple data sources and to scale out to support high volumes of events.

In this thesis I discuss the latest progress in research and applications of CEP. I review the requirements of an event-driven service oriented architecture. I designed and implemented a scalable, distributed architecture integrating a complex event processing service with an enterprise service bus (ESB).

The key architectural insight in the system is to separate the integration functionalities of the ESB and the complex event facilities. This results to a stateless ESB, which can be scaled out by adding more processing nodes. A dedicated CEP cluster can then be tuned to handle high throughput and scaled out separately.

The results of the performance tests show that the system can be scaled out by adding more compute instances to a cluster. The ESB achieved a throughput of 1 750 messages/s per instance in my test setup and could be scaled out linearly. The throughput of the CEP cluster depends heavily on the required computations and data dependencies. I demonstrate an example case, where the cluster handles 28 000 events per second on eight processing nodes. The median latency for receiving an event at the ESB, sending it to CEP and receiving the derived events for further processing is under 10 ms.

Keywords: complex event processing, enterprise service bus, event-driven service oriented architecture, environmental data

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Antti Aalto

Työn nimi: Monimutkaisten tapahtumien käsittelyn skaalautuminen hajautetun palveluväylän osana

Päivämäärä: 19.11.2012 Kieli: Englanti Sivumäärä: 9+66+7

Perustieteiden korkeakoulu

Tietotekniikan tutkinto-ohjelma

Professuuri: Ohjelmistotekniikka Koodi: T-106

Valvoja: Prof. Heikki Saikkonen

Ohjaaja: FM Atso Haapaniemi

Monimutkaisten tapahtumien käsittely (CEP) on uusi teknologia, joka helpottaa virtamuotoisen ja historiallisen datan analyysia ja säännönmukaisuuksien löytämistä. CEP voi yhdistää tapahtumia useista virroista, tietokannoista ja muista lähteistä. Markkinoilla on useita CEP-moottoreita, mutta harva niistä pystyy sekä käsittelemään saumattomasti dataa useista lähteistä että skaalautumaan suurille tietomäärille.

Käsittelen tässä diplomityössä viimeisimpiä tutkimuksia CEP:ssä ja teknologian sovelluksia. Selostan tapahtumavetoisen palvelusuuntautuneen arkkitehtuurin vaatimukset. Suunnittelin ja toteutin skaalautuvan, hajautetun arkkitehtuurin, joka mahdollistaa monimutkaisten tapahtumien käsittelyn palveluväylässä.

Arkkitehtuurin suurin oivallus on erottaa palveluväylän integraatio-ominaisuudet ja monimutkaisten tapahtumien käsittely erillisiksi kokonaisuuksiksi. Seurauksena on tilaton palveluväylä, jota voidaan skaalata lisäämällä rinnakkaisia virtuaalikoneita. Pelkästään monimutkaisten tapahtumien käsittelyyn tarkoitettua CEP-klusteria voidaan silloin skaalata erikseen ja samalla ottaa datariippuvuudet paremmin huomioon.

Suorituskykytestit osoittavat, että järjestelmää voidaan skaalata lisäämällä virtuaalikoneita. Palveluväylä pystyy käsittelemään 1 750 viestiä sekunnissa yhdellä koneella ja skaalautui lineaarisesti koneiden määrän kasvaessa. CEP-klusterin suorituskyky riippuu voimakkaasti laskennan vaativuudesta, tyypistä ja datariippuvuuksista. Esittelen esimerkkitapauksen, jossa klusteri käsittelee 28 000 viestiä sekunnissa kahdeksalla koneella. Mediaanilatenssi tapahtuman vastaanotosta palveluväylässä, sen lähettämisestä CEP-klusterille ja käsittelyssä syntyneiden vastineviestien vastaanottoon on alle 10 ms.

Avainsanat: monimutkaisten tapahtumien hallinta, palveluväylä, tapahtumavetoinen palvelukeskeinen arkkitehtuuri, ympäristödata

Acknowledgements

This thesis was born from the hype of some of the hottest technologies in 2012. This unlimited enthusiasm and inspiration slowly developed to a much more pragmatic view on complex event processing, real-time stream processing, distributed systems and big data.

The research was done in MMEA project at HiQ Finland. I kindly thank Atso Haapaniemi for the guidance I received in various situations. I'm also grateful for the flexibility to study and work I have enjoyed under the supervision of Jukka-Petri Sahlberg. Special thanks go to the MMEA team members Juhani Jaakkola, Janne Juhola, Jussi Kolehmainen, Juha-Matti Lehtinen, who have played a significant role in developing the components relevant to this thesis.

From Aalto University I want to thank professor Heikki Saikkonen and D.Sc. Seppo Törmä. They both expressed much interest in my work and provided rapid and to the point feedback and comments.

And last but not least, I'm thankful for all the support I received from my family and friends during this process.

Espoo, 19.11.2012

Antti Aalto

Contents

Abbreviations

API	Application programming interface
ASR	Architecturally Significant Requirement
CEP	Complex Event Processing
CSV	Comma separated values
DS	Data source
EBS	Elastic Block Store
ED-SOA	Event-Driven Service-Oriented Architecture
EPA	Event Processing Agent
EPL	Event Processing Language
EPN	Event Processing Network
ESB	Enterprise Service Bus
JAXB	Java Architecture for XML Binding
JAR	Java ARchive
JIT	Just-In-Time
JVM	Java Virtual Machine
MOM	Message-Oriented Middleware
SOAP	(formerly) Simple Object Access Protocol
SUT	System under test
WSDL	Web Service Description Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

List of Figures

List of Tables

Chapter 1

Introduction

Even though environmental data is produced in massive amounts, it is rarely exploited to its full potential. The data is usually only used and kept within certain organisation boundaries, partly because of the lack of interoperable formats and services to share the data. The problem is boosted by the ever increasing number of new measurement formats and the tremendous amounts of data produced by an increasing number of sensor networks and computer models. (Kotovirta 2012)

Processing environmental measurements and making real, wide-spread use of it, is a big data problem. More importantly, the processing must be nearly real-time, because few are interested in yesterday's weather forecast or a even a couple of minutes late tsunami alarm. (Wächter et al. 2012) Because of the timeliness aspect of the data, efficient stream processing technologies must be used to reap most benefit of it. However, to have more general use, the measurements must be given a context and linked with other up-to-date data, including other data streams, older measurements and computational models (Kotovirta 2012).

One emerging technology for handling such complicated patterns is complex event processing (CEP (Leavitt 2009)). CEP simplifies expressing relations between data and finding patterns from "a cloud of events" (Luckham 2001; p. 28) It is a mix of old and new technologies and fits well in many event-driven applications. The multitude of applications, as surveyed in Chapter 2, draws a very promising picture of the possibilities of CEP.

Another, a bit more established concept investigated in this thesis is enterprise service bus (ESB). ESB is an architectural pattern which aims to ease integration of various otherwise incompatible applications. It has also been implemented as many different products carrying the name ESB. Event-driven SOA is a widely used design pattern in ESBs and is a very natural fit to CEP. (Leavitt 2009)

In this thesis I describe an integration and processing platform for the environmental data, called MMEA Bus. I explain several key technologies suitable for the application, present an architecture for the platform and evaluate a prototype implementation. The goal of this thesis is a prototype of an interoperable and performant complex event processing platform for environmental data, which application developers can build on.

1.1 Motivation for complex event processing

In contemporary enterprise organizations there are huge amounts of transactions happening that manifest themselves as events (Ghalsasi 2009). For instance, trades in a stock market or natural disasters are easy to describe as events happening in some point of time (Adi et al. 2006). In this section I will give two brief examples for applications of complex event processing.

Events can be abstracted on many different levels (Luckham and Frasca 1998). In the stock market a trade might consist of several bids, offers, payments and financial transfers. On the lowest level a stock trader is responsible for executing them. At the end of the day the trader is probably interested in how well he or she did and wants to see a list of all transactions. The boss of the trader doesn't care about every single trade that has been made but is looking for something more complex. He might perhaps care of a complex event of how much profit (if at all) our trader has made during the last month. (Luckham 2001; pp. 294-327)

The data may have even more users. An auditor might define some constraint (e.g. trader may not buy stocks to his or her personal account, if he has just before received an buying order and then proceeds to sell those stocks), which indicates breaking some rule (in this case trading to an personal account), and then receive a notification, if this constraint is broken. (Luckham 2001; pp. 294-327)

In this thesis project, complex event processing was used to process environmental data, e.g. temperature, CO_2 levels, light, humidity, chemical concentrations or vibrations. For example, we might have vibration sensors installed in buildings sending their readings to our CEP server. When the server receives the readings, it might detect that the sensor is vibrating fast. CEP engine can then check if most of the other sensors in the same area are reporting heavy vibrations to detect an earthquake. In a CEP system we might also be able to leverage additional sensors. A pressure sensor under a road might allow us to filter out cases where a large truck is driving through the district.

Complex event processing offers the users a way to automate the detection of anomalies or other interesting phenomena. It is way too tedious for the auditor to correlate all the trades made by all the traders to detect all the various blunders they might have done. In the earthquake example we can use CEP to process the data real time and detect the danger quickly. CEP system might be able to send alert to the inhabitants and authorities critically faster than a human in its place. (Luckham and Frasca 1998)

1.2 CEP as a part of ESB

Enterprise service bus is an integration pattern and product. According to (Menge 2007) there was no consensus of the definition, but a common thing in all products and solutions

marketed as ESBs was that it provides message-oriented middleware (MOM) for enterprise application integration (EAI) use cases. As noted by (Luckham 2001), the business messages are very interesting input for CEP. Thus, CEP is widely presented as a natural part of an ESB (Menge 2007, Bo et al. 2008, Leavitt 2009, Wishnie and Saiedian 2009).

From the implementation point of view the combination of CEP and ESB is not so straightforward. (Bo et al. 2008) recognized that most ESB products are weak in CEP. In their paper they present an implementation of an ESB that is specifically intended for complex event processing. Nevertheless, still their implementation lacked the capability to scale out, by which I mean, to run CEP parallel on multiple machines.

(Wishnie and Saiedian 2009) recognized the problem of connecting CEP with the current MOM products and their drastically different models for scaling out. In their paper they describe an architecture and implementation of a complex event routing infrastructure based on an unstructured peer-to-peer network. Their aim was to introduce CEP as a first class citizen in MOM. In their P2P network, more computing nodes could be added to increase the resources of the CEP engine. Still, this novel approach did not lead to a breakthrough in performance and the maximum throughput was limited to less than 1500 events per second.

1.3 Research questions and the scope of the thesis

The main research question of this thesis is how does complex event processing fit to an enterprise service bus. To answer this question, I must first understand the complications posed by CEP in an ESB.

If an enterprise service bus can be implemented with completely stateless instances in a cloud computing environment, the scalability should be only a minor concern. By stateless it is meant that the instances don't contain any volatile data about the current clients, connections or messages they are serving. In completely stateless service a successful strategy for achieving scalability would be to just add more server instances and distribute the load evenly to those.

ESB is mostly based on the principles of event-driven service-oriented architecture, which is by design stateless (Schmidt et al. 2005). However, the complex event processing breaks this harmony by being inherently stateful. To match patterns of multiple events the engine must have the information about the previous events, which in our beautiful stateless case could have gone to any instance in the cluster.

In this thesis I investigate the current state of the complex event processing in enterprise service buses. I review the previous research on implementing CEP and using it in different business cases. I define the requirements for MMEA Bus, which is an ESB aimed for serving environmental data. Following these requirements, I describe an architecture for MMEA Bus and its implementation. Lastly, the implementation is evaluated qualitatively and with performance tests focusing in throughput and latency.

1.4 Definitions and naming conventions

Because complex event processing is such a new field, the terminology has only recently begun to stabilise. For example, EPL has been used to mean "Event Pattern Language" (Luckham 2001) before the currently most widely accepted "Event Processing Language." Other terms with essentially the same meaning are "Continuous Query Language" (CQL) (Arasu et al. 2006), "Event Query Language" (EQL) (Eckert et al. 2011) and "StreamSQL" (Luckham and Schulte 2008). In this thesis I will follow the Event Processing Glossary - Version 1.1 (Luckham and Schulte 2008) whenever possible with the disambiguations defined here.

Even the field of complex event processing has many names. The next most popular names are stream processing (Stonebraker et al. 2005), data stream management (Babu and Widom 2001), event stream processing (Arasu et al. 2006) and just plain event processing (Luckham and Schulte 2008). All these terms mean essentially the same and the goals of the academics using these terms have been very similar. Maybe the only difference is that "complex event processing", as described by (Luckham 2001; pp. 28-37), don't inherently think of events manifesting themselves in a stream but in a less structured manner. In this thesis, I use always the term CEP, even if the original author of some cited system called it something else.

In the seminal book "The Power of Events" written by David Luckham events are described to reside in an event cloud (Luckham 2001; pp. 28-37). This vague notion gives impression of static events that can be accessed repeatedly, independently of the time. Furthermore, (Stonebraker et al. 2005) presents as a requirement for event processing systems that they must seamlessly integrate stored and streaming data, although it is acknowledged that it may be hard to fulfil. As there are little ready solutions for this problem and in MMEA Bus the amount of sensor data is way too big to be stored completely in memory, accessing events outside a defined window (e.g. last 24 hours) is not considered in this thesis in detail. Nevertheless, combining historic data with the current data streams is an interesting question in CEP (Stonebraker et al. 2005).

For word *event* there are two definitions referring to anything that happens (e.g. earthquake) and to the representation (e.g. high vibration sensor reading). Because of the nature of the application I'm using CEP, I will use *event* always for the representation. This also makes it sensible to refer to an earthquake by the term *complex event* as it is something that is deduced from several strongly vibrating sensors in a certain geographical locations.

1.5 Structure of the thesis

This thesis consists of four parts: a literature review, the architectural design of MMEA Bus, its implementation and the performance test results. In detail, the chapter outlay is the following.

I begin in chapter 2 by introducing the technical background for service-oriented and event-driven architectures. I will also explore the benefits offered by enterprise service bus and take a detailed look in complex event processing. I will also introduce the fundamentals of scalability, high-availability and other general requirements often associated with enterprise event processing. Before proceeding to my own architecture, I describe the previous major research on complex event processing.

In chapter 3 I will explain the architecturally significant requirements of MMEA Bus. An architecture fulfilling most most of the requirements is presented and its implementation is described in chapter 4.

I evaluate the implementation with comprehensive performance tests. The results are given in chapter 5. The main results are summarised in chapter 6.

Chapter 2

Technical background

2.1 Enterprise application integration

Enterprise application integration (EAI) means the act of creating new business solutions by sharing data and business processes between different systems in a unified way (Ruh 2001; p. 2). EAI makes use of middleware that provides application-independent services for communicating over a network. During the last 20 years there has been huge advances in middleware solutions. Currently many state of the art systems utilize messaging based asynchronous communication mechanism and are built on the principles of Service Oriented Architecture (SOA). In this section I will describe the most relevant technologies and concepts the MMEA Bus is built on.

2.1.1 Service oriented architecture

Service oriented architecture (SOA) models were created to facilitate the design of enterprise software. SOA addresses the following concerns. First, many systems need to be integrated to a single interoperable entity. Second, the existing components may not talk the same language. Third, businesses implement new products rapidly. Another source of integration requirements are mergers, which bring new, incompatible systems to the ecosystem. (IBM 2004; pp. 34-52)

These concerns elicit the following requirements to simplify the architectures: The existing assets must be reused. The architecture must be implemented and adopted incrementally. There must be a middleware, which provides transactions and multiple communication models. The middleware must also provide security features, support multiple platforms and programming languages. The enterprise platforms must scale to support high volumes of events. (IBM 2004; pp. 34-52)

These requirements can be fulfilled by encapsulating functionality in large grained services. Services are defined by explicit, implementation-independent interfaces. They are loosely coupled and provide reusable business functions. The services can often be composed to create more powerful functionality. (IBM 2004; p. 37)

In many systems the aim is to define stateless services. In many cases this is not possible, because there is much interdependent business data. A somewhat weaker goal is to design connectionless services, where all the sent messages are self-contained. This allows efficient implementations and fault-tolerance features. (IBM 2004; pp. 45-47)

SOA can be implemented in an event-driven way. Event-driven SOA decouples interactions. The event publishers need not be aware of the receivers of their messages. It also makes many-to-many communications easier. The communication is asynchronous, which minimises delays on the implementation level. (Maréchaux 2006)

2.1.2 Web services

Web services are software providing an interface that can be accessed over a network. They use XML-based technologies for data representation and interface definitions. Web services exchange SOAP messages, typically over HTTP, although SOAP can be bound to any transfer protocol. WSDL is a machine-processable interface definition language that can be used to automatically generate code for client and server ends. (Haas 2004)

Web services are well fit to implement SOA. The key technologies are open and standardised. SOAP offers an XML-based format for exchanging documents and passing messages. WSDL is an open standard and machine readable interface definition language for web services. The standards include also UDDI service registry, which was originally meant for locating services over the Internet. Although that goal failed, it has found new uses in the internal registries of enterprises. (IBM 2004; pp. 53-56)

Web services are independent from communication mechanisms and they have a wide industry support. There are also many standardised extensions. Most important ones are probably WS-Addressing, WS-Security, which move the respective functionality to the middleware layer instead of a developer having to reimplement them in each application. (IBM 2004; p. 54)

2.1.3 Enterprise service bus

Enterprise Service Bus is relatively new integration component that has gained a lot of traction in industry. There are several commercial and open source products that call themselves ESB, which have slightly different sets of features. Anyway, the term is most commonly used for a message-oriented middleware component, which facilitates interaction

of distributed services by providing routing and mediation infrastructure. ESB is built on open standards, e.g. web services and other technologies exploiting XML. Common features included in an ESB are invocation, routing, adaptation, mediation, security and complex event processing. (Menge 2007)

An ESB is able to send requests to other services and combine the results. Because it is an integration component, it is expected to be able to handle several protocols, for example web services (SOAP), message queues (JMS), remote method invocation (RMI) or email (SMTP). If there are new services behind previously unsupported protocols, the ESB is expected to be the point of integration implementing the protocol and providing other systems a standard interface most commonly over a web service. (Menge 2007)

Routing can be handled in an ESB by supporting WS-Addressing. WS-Addressing adds new attributes to the SOAP header, which can be processed independently of the message contents. (Box 2004) The ESB may not always obey exactly the routing information in the headers but it might apply some changing rules too. One application is distributing the load for several identical services on separate servers. Routing can also be content-aware. (Menge 2007)

An ESB may be able to adapt different message formats to a unified one. At the very least, it should be able to transform the messages from one format to another so that different services are able to communicate with each others. In the simplest form this can be a straight-forward XSL transformation between different XML schemas. However, the ESB must be able to make adaptations between all the endpoints it integrates to. (Menge 2007)

There are some differences in the ways the ESB products define mediation. In some cases the mediators are able to do powerful computation (Godage 2007) but in some cases they are limited to the very basic and lightweight forwarding of messages with simple rules (Wheeler 2011).

Security is an obvious requirement in an enterprise environment. An ESB is a good place to support it as it is a very central component and is able to interact with many different security providers. WS-Security defines multiple features, including authentication, identification and encryption. (OASIS 2006)

2.2 Complex event processing

Complex event processing is an emerging technology which operates on event streams and historical data. It can be used to detect patterns consisting of multiple events in near real-time. CEP performs operations on events and their compositions while they occur (Eckert and Bry 2009). It enables various use cases and extracting meanings and detecting phenomena in event patterns.

In this section I give a brief introduction to the fundamentals of CEP from the viewpoint of our application in environmental sensor data processing. I introduce the concepts of an event, an event processing network and an event processing agent.

2.2.1 Events in CEP

There are two parallel definitions for the word *event*. First, it can mean "anything that happens, or is contemplated as happening", e.g. a financial trade is made. Second, it may also be understood as the object acting as the manifestation of something that happens, a purchase order is sent. (Luckham and Schulte 2008) In MMEA Bus we mostly consider the messages under mediation to represent events, which implies the second definition, although in our usage there is not much possibility of confusion.

According to Luckham (2001; p. 88), an event always has three aspects: form, significance and relativity. *Form* is the "real" physical or electronic representation of an event. In our sensor data processing system the form is a SOAP message or some other message in our internal format. Every event signifies an activity and thus the *significance* is the relation to the real world phenomenon. Events are also often related to each other. *Relativity* includes time, causality and aggregation of complex events.

In an ideal world we would like all the events to have a timestamp that exactly tells us the time of the generation of the event. This timestamp would tell us the full ordering of the events in the time and let us easily do reliable interval calculus and elicit causal relations. (Luckham 2001; pp. 94-100) However, time in distributed systems is a complicated issue. All nodes in the network have their own internal clocks, which are not perfect and may have different skews and drift rates. If timestamps for two events are issued by different machines, we cannot rely solely on them when defining an order for them. The only thing that we can trust in clocks is that a single machine always gives a later timestamp to a event that arrives later. (Coulouris et al. 2011; pp. 612-615)

One interesting question with timing information is, where should the timestamps be issued. If we had some single event processing engine located on one physical machine through which all the events would flow, we could use it to issue reliable timestamps. This would not be a perfect solution, because there is network latency when the message is in transmission from the source to the engine. Nevertheless, the latency can be tolerated, if there is no jitter (the latency doesn't vary), and if we are more interested in the relative than absolute timing of the events. (Coulouris et al. 2011; pp. 615-617) If we ease the requirement that the time must be exact, we can even use multiple machines for assigning the timestamps. If we assume network latency of 20 ms, we can safely say that cloud virtual machine instances using Network Time Protocol to set clock do not add measurable inaccuracy to the time. (Windl et al. 2009)

2.2.2 Event processing networks

An event processing network (EPN) is a conceptual model describing the elements of complex event processing. There are four types of components in an EPN: event producers,

consumers, processing agents and channels. The EPN was first described by Luckham (2001; p. 207) and later defined more formally by Sharon and Etzion (2008).

An EPN can be represented as a directed graph, where the nodes are event processing agents, producers or consumers and the edges are event channels. The purpose of a channel is to deliver events between the nodes of the graph, which implement data input, output and the intended operations. As noted by Etzion and Niblett (2010; p. 117), the networks can also have loops to allow feeding events back upstream. The EPNs can also be nested, which means that EPAs can be implemented as EPNs (Etzion and Niblett 2010; p. 118).

Event processing network can be used for distributing the load in a complex event processing system. Because the EPN graph shows explicitly the causality of the events, it also shows the dependencies required by the EPAs. This is further explained in Section 3.4. (Lakshmanan et al. 2009)

2.2.3 Event processing agents

In a complex event processing system multiple rules are applied to the events that flow through. These rules are applied in Event Processing Agents (EPA), which are the fundamental building blocks of CEP. EPAs monitor the patterns in event flows and react according to their defined function. There are atleast two classifications of EPA types, first one by Luckham (2001) and a later one by Etzion and Niblett (2010). On higher level their differences are small: both take events as input and produce new events as output according to some reaction rule

Luckham (2001; p. 177) classifies agents as in *filters*, *maps* and *constraints*. Filters are event patterns that remove uninteresting events from the streams. Only relevant events are passed further to maps and constraints. Maps are used to create higher level complex events by aggregating multiple lower level events. These aggregations specify event hierarchies. Constraints can detect the presence or absence of an event or a complex event in a stream. They create notification events, when the constraint is broken.

The classification by Etzion and Niblett (2010; pp. 121-122) is more fine-grained and reflects more closely the contemporary event processing systems. It begins by defining the functions that a single EPA can execute. One filter can include one or more of the functions *filtering*, *matching* and *derivation*. Filtering is defined as previously. Matching finds patterns in the events and creates new events according to that pattern. Derivation corresponds to Luckham's aggregation with the output of matching as its input.

Etzion and Niblett (2010; p. 123) define nine different EPA types: *filter*, *pattern detect*, *transform*, *aggregate*, *split*, *compose*, *translate*, *enrich* and *project*. The most important types are filter, transformation and pattern detect. Filter can be included in any of the

other EPA types as function described above, but it can be useful also as a standalone agent. Pattern detection is discussed further in section 2.2.5.

Transformation is an abstract supertype of translate, aggregate, split and compose and never used by such alone. Translation means directly mapping one event to another event. In an enterprise software environment this can be done with for example XSLT. Two more advanced translation agents are enrich, which bundles an event with data from some global source (e.g. database), and project, which acts like a project operator in relation algebra and can be used to select certain attributes from an event. (Etzion and Niblett 2010; pp. 125-126)

Aggregate works like explained above. Split creates new events by copying a subset of the attributes of an event to new ones. Compose takes groups of events from multiple inputs, matches them and creates derived events.

2.2.4 Event producers and consumers

Event producers, also known as sources and emitters, are nodes of the event processing network that originate events and supply them for processing. It must be noted that also event processing agents create new complex event. Thus defining the producers is not always trivial. However, defining some sources and sinks for the events might help to build abstractions and to understand the problem at hand. Etzion and Niblett (2010; p. 87) define the event producers as those nodes of the event processing network, which don't take any inputs.

There are several ways for creating events. In many business systems the CEP module observes traffic in the middleware layer (Luckham 2001; p. 129). In MMEA system the environmental sensors emit readings and act as the source for events. Furthermore, in our CEP engine we create complex events. These events can be observed again in other parts of the system and are thus in every way regular events.

The definition of event producers depends always on the abstraction level and the boundaries of the event processing network (section 2.2.2). In MMEA system it is reasonable to consider the environmental sensors as the producers, as they are the furthest observable entity in the system. Furthermore, we cannot influence their inputs. Anyway it is healthy to remember that even the sensor readings are, at least on some level, abstractions and approximations of the real world events (e.g. "the sun is shining" or "gas is leaking").

Event consumers, also known as sinks, are agents that don't emit any events back to the system. Their definition is in the same way quite arbitrary, as was the case with the producers, but still possibly helpful. (Sharon and Etzion 2008)

2.2.5 Pattern detection

Pattern detection is the crown-jewel of complex event processing. I'll discuss here some the basics of pattern detection as defined by Etzion and Niblett (2010; pp. 214-242). Contemporary complex event processing engines using special event processing languages (EPL) most of the EPAs are very powerful pattern detection agents (EsperTech Inc. 2012). Later in section 4.2.7 I will show some example EPLs.

Pattern detection works always in some context. The context defines the relevant events supplied for pattern matching. It can be temporally or spatially bounded. Context can also be based on semantics of mutually referenced objects or entities. This context is often called a window.

The definition of a pattern starts with pattern signature. The signature includes the pattern type, parameters, relevant event types and policies. The events are selected to detection according to the pattern type. (Etzion and Niblett 2010; p. 216)

Pattern detection is executed by filtering out the obviously irrelevant events. The stream of filtered events is then forwarded to an event matcher, where the events are grouped into sets of participant events. The matcher is run on a set and it chooses the eventual groups of events that fill the conditions of the EPA and sends them to derivation. The derivation step creates new complex events. (Sharon and Etzion 2008), (Etzion and Niblett 2010; p. 216)

In (Etzion and Niblett 2010; pp. 214-228) the patterns are divided in two categories, basic patterns and dimensional patterns. They consist of logical operations (conjunction, disjunction, negation), threshold patterns, subset selection patterns and modal patterns. For example, there are patterns to detect if one instance of each types of the participant set (or none of those) has been seen (Sharon and Etzion 2008). A threshold pattern might trigger when three events of some type has been processed. Subset selection patterns can select, for instance, the relative n highest values of a set. Modal patterns can check if some assertion is true always or sometimes. (Sharon and Etzion 2007)

As the name implies, dimensional relate to some dimension: time, location or both. This enables comparisons and orderings of events. Events can be processed as sequences in time. For instance, dimensional patterns can be used to find trends or spatially close events. (Etzion and Niblett 2010; pp. 228-242) Good examples of event algebra system for an industrial application are described in (Paschke et al. 2010) and (Zang et al. 2008).

These patterns as such without clearly defined rules, e.g. for precedence and associativity, can be very ambiguous. For instance, if we want to match a pattern that consists of one instance of event type A and one of type B and we receive two As and one B, how do we know, which A to select? Pattern policies (Etzion and Niblett 2010; pp. 237-238) and directives (Sharon and Etzion 2007) allow us to express evaluation, cardinality,

repeated type, consumption and order policies. Evaluation policy defines whether we want to evaluate the pattern every time a new event is observed. Another option would be to defer evaluation and run them as a batch. Cardinality policy determines how many times one event can be part of a pattern matched. In the example, if we can either select only the first A and classify the second as unmatched. Or we could match B twice and derive two matches. (Sharon and Etzion 2007)

Repeated type policy defines which instances of a repeated type are kept in the set of relevant events. Possible policies are the first, the last, all of them, or some more complex criteria, e.g. one with the maximal value. By setting the consumption policy one can dictate whether an event is removed from the set of participant events when included in a matching set. Lastly, order policies let us define the attribute the events are ordered by. (Etzion and Niblett 2010; pp. 239-242)

2.2.6 Previous applications of CEP

Several applications of CEP are described in academic literature. In this section I highlight some practical applications of complex event processing in financial services, warehouse management, manufacturing and healthcare.

Financial applications

Adi et al. (2006) present two other cases for exploiting CEP in financial services are discussed. The first case is an alert system for banking, which is very similar to the scenario later developed by Mukherjee et al. (2010) (discussed below). In the second case an insurance underwriting process was automated. The latter case focused in decoupling the rules of underwriting insurance from the business process the application goes through.

The scenarios described by Adi et al. (2006) elicit many requirements that are very generally applicable to CEP systems. First, changing the rules of the system must be feasible. Second, it can be useful supply the newly produced events back to the system and apply the rules again to them. Third, it is not always the best option to run the whole CEP engine (AMiT (Magid et al. 2010) in their case) to match all patterns. Single events can sometimes matched more easily with less powerful but faster techniques. These ideas are also applied in this thesis.

Mukherjee et al. (2010) used IBM InfoSphere Streams to monitor capital markets. Their aim was to timely detect fraudulent activities on stock market, for which purpose they implemented a proof of concept system. For trade surveillance they created continuous queries, which tested, for example, for suspicious long gaps, anomalies in trade prices and quantities and variations in price.

Traditional approach for detecting anomalies on stock market has been statistical analysis after the trades have been completed. This method also has the benefit that new analyses can be run when needed, as new ways to cheat are always possible. However, for fast reactions for frauds, real-time monitoring is a necessity and can be as a complementary method. In the tests ran by Mukherjee et al. (2010) the approach based on streaming data proved to be orders of magnitude faster than a traditional approach based on relational database and single issue queries. Their system was measured to handle 50 000–200 000 events per second with each single rule.

Monitoring material flows and business processes

Perhaps the most studied use case for CEP is its applications on RFID-based systems, most prominently in warehouses and factories dealing with huge and moving inventories. The specification of the second generation of RFID specification dictates that a reader must handle 1 800 reads per second (EPCglobal Board 2008). In (Dong et al. 2006) the core principles for applying CEP in RFID middleware are analysed and proposes a solution based on Application Level Events defined in (EPCglobal Board 2009). The purpose of the solution is to screen meaningful data from irrelevant one by matching event patterns.

Other examples of using RFID-generated data include research by Zang et al. (2008) and Baarah et al. (2011). Zang et al. (2008) designed a system for monitoring a Chinese refrigerator factory is described. The system is based on SOA and uses elaborate XML representation for events. The system is fairly performant achieving 80 000 generated new events per second in the test case they describe.

Integrating RFID-based data and business processes have also been studied. According to Baarah et al. (2011) the possibilities of monitoring and controlling cardiac patient flow in a large Canadian hospital was analysed. The proposed architecture includes four data sources: medical equipment, physiological sensors, RFID tag readers and the BPM system. This data was collected to a central repository and processed by a CEP engine.

It is clear that the raw RFID data is not useful but must be connected with BPM to give the messages a meaning. This resulted in real-time dashboard that could show the most interesting timely information of the healthcare processes in a matter of seconds, in contrast to the conventional systems, in which similar analysis could take hours or days. Events processed by CEP can be very fine grained and could be used to enhance care delivery instead of only for administrative purposes. For example, the system could alert if some patient had stood in a queue too long, or it could just be used to optimise resource utilisation.(Baarah et al. 2011)

While the CEP systems are often developed bottom-up by first identifying the event information available, Kellner and Fiege (2009) describe a top-down approach. This allows businesses first to define key performance indicators and other abstract measures, and then

hierarchically proceed down to find the correct low level events in a changing environment to calculate them.

One of the most influential academic CEP projects that been undertaken, Borealis, was demonstrated in (Ahmad et al. 2005). In the demonstration Borealis was used to create real-time gameplay information for team management in an open source first-person shooter game Cube. The analytics could provide notifications, for instance, when there are more than 20 enemy players near the own home base. The system was shown to handle an increasing and variable load while still maintaining low latency. Borealis is further described in Section 2.4.1.

2.3 Scalability and high availability

Scalability and high availability are timeless, hot topics in distributed enterprise systems. Services provided over the Internet are required to handle millions of requests per second and even a short period of downtime can be costly for a business. In this section I will explore the theoretical backgrounds of scalability and high availability.

2.3.1 Scalability attributes for CEP

Complex event processing systems have a very wide variety of scalability requirements. The best collection of different variables is in the book written by Etzion and Niblett (2010; pp. 264-266) and I'll mostly follow it in this section while complementing with some details from other sources.

Volume of events: The most straightforward variable is the number of input events processed per second. It is also the most benchmarked variable in literature (Mendes et al. 2008; 2009, Kleiminger et al. 2011). Another dimension of the volume of events is the message size. The larger message size can affect the performance of some components quite a bit, as seen in chapter 5.

Event processing agents: A useful CEP system must also scale to support a large collection of event processing agents. Large computations are often most easily presented as simple steps of a complex event processing network. This requires that data can be passed fast from EPA to next EPA or that there is some other optimisation to remove this step.

Producers and consumers: In our application the system must accommodate many producers and consumers of the data. A substantial number of individual users might want to make their data available to others. The system must also be able to send the processed data and identified complex events to the correct subscribers.

Window size: Window size is a substantial source of complexity. Window size affects on how many events a computation is applied at a time. For example, a pattern detection might be applied for all events for the last five minutes. The window size can have drastic effects on EPAs if the computational complexity grows faster than linearly in respect to the input size.

Computational complexity: Even though the focus in CEP and in this thesis is in Big Data, the traditionally much researched challenge of computational complexity still plays a big role (Etzion and Niblett 2010; p. 266). Much of the depends on how well the computation can be partitioned, parallelised and distributed (Etzion and Niblett 2010; p. 271) (Heinze 2011).

Environment: The developer of a CEP system must take into account the specifics of the system. Different environments offer variable amounts of memory and CPU cycles. Some environment have limitations in power consumption. In a distributed environment message passing adds limits to latency. Sometimes the bottlenecks can even reside outside the CEP part of the system, for example in input or output channels like message queues or web services transmitting the events. (Etzion and Niblett 2010; p. 266)

Constants: There are also some factors that influence heavily the constants of the computational requirements. In addition to the already noted variable message size, also its encoding matters much. While SOAP-enveloped XML-based messages might offer a good support for enterprise integration patterns, their serialisation and deserialisation are computationally expensive compared to lighter and flatter representations.

2.3.2 Availability

When developing a distributed version of Aurora stream processing system, Cherniack et al. (2003) recognised three sources of failures in distributed stream processing: 1) server and communication failures 2) sustained congestion levels and 3) software failures.

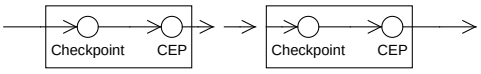


Figure 2.1: Persisting checkpoints

The CEP engines depend heavily on the context information of the events, which includes summaries of the past events. If an engine fails, this context can be lost. As an example of a fault tolerance scheme, the figure 2.1 shows checkpoints which persist the data. If a CEP engine fails, the data can be retrieved from the queue held by the checkpoint and used to refill the window. (Stonebraker et al. 2005)

2.3.3 Two dimensions of hardware scaling

Scaling a computing system can be achieved by scaling *up* or scaling *out*. By scaling up I mean using a bigger, faster computer with more CPU cores and more memory. By scaling out I refer to adding more computers (or cloud virtual instances) to a cluster of computers.

Scaling up offers some clear benefits, because it allows the software to run on a single machine with a shared memory. This reduces the architectural constraints imposed on the software run on the hardware. Furthermore, management of a single big machine is easier than management of a cluster. Vertical scaling is not always the perfect solution. Michael et al. (2007) showed that scaling up can be very expensive. It was also observed that to efficiently use all the power in one box, it might be necessary to use similar techniques as in a distributed system. This was referred as "scaling-out-in-a-box."

When considering only hardware costs, scaling out can often be cheaper. Horizontal scaling can utilize affordable commodity hardware and the buyer doesn't have to pay hardware vendor a premium for highly specialized business hardware. However, to leverage distributed hardware the software must be carefully designed to distribute the processing. Execution running on one of the machine instances cannot refer to memory located on an another instance. (Michael et al. 2007)

Sometimes the best results can be achieved by combining both vertical and horizontal scaling. Sometimes the biggest possible single machine is not fast enough and the system builder must resort to a distributed, scaled-out architecture. In section 2.4 I will describe a solution, which will utilize multiple instances of the biggest EC2 instance type available.

2.4 Complex event processing implementations

MMEA Bus is required to be able to process a large amount of messages per second. Because we want to allow non-expert application developers to define their own EPL statements, we cannot rely solely on fast code, minimalistic implementations and quick processing, when addressing the performance. Thus we must make sure that our hardware offers enough processing power and that our system is able to exploit it.

In this section I explain the basic principles in scaling complex event processing. I review the literature describing the different approaches taken before. During the review I will comment on how the different ideas were considered while designing MMEA Bus and which were eventually incorporated in it. A more detailed view on the implementation is given in Chapter 4.

As already explained in section 1.3, we cannot just scale out by adding more nodes and then forward events to them randomly. CEP must be able to correlate and aggregate

arbitrary events in an event window and thus the events in processing are inherently interdependent.

There are different granularities on which we can distribute computation in CEP. Heinze (2011) calls this elasticity and identifies three levels of it in CEP: engine, query and operator level. In engine level elasticity the smallest unit of processing is a window, and every engine runs on one machine. More machines are employed by adding more engines to the EPN. In query level elasticity, queries of a single engine are deployed on multiple machines and the input data is split between them. The operator level elasticity is the most fine-grained level and allows every operator to run parallel on different machines. However, this increases the communication overhead inside a single query.

2.4.1 Aurora, Medusa and Borealis

Carefully crafting finer grained event processing networks rather than defining too powerful and complicated event processing agents may allow breaking the processing down to a series of steps. The steps can then be pipelined on multiple machines. The most important project taking this approach is Borealis, a distributed complex event processing engine developed at Brandeis University, Brown University and MIT. It is a successor to the Aurora and Medusa projects. (Ahmad et al. 2005)

Aurora

Aurora is a CEP engine, although the term used by the research group is "stream processing engine." Aurora system is expressed by a simple boxes and arrows flowchart as shown in Figure 2.2. The boxes are EPAs and arrows event channels. The EPAs offered by Aurora include filter, binary merge (union), sorted window, map, join, an extrapolation operation and several aggregation operations and they are further described by Carney et al. (2002). The queries in Aurora are defined with a graphical user interface (Carney et al. 2002) or an XML-based query language (Borealis team 2006).

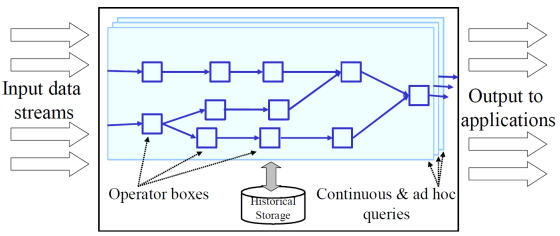


Figure 2.2: Aurora system model (Carney et al. 2002)

There are three query models: continuous queries, views and ad hoc queries. Continuous queries operate on real-time data. They process events as they arrive and store data only

for the time required by any given window. Views execute continuous queries but store the results so that they are available for user initiated retrieval. Ad hoc queries create new paths to the network and run their operations on all the events that are still available in the queue supplying data for the EPN. (Carney et al. 2002)

Some channels between boxes act as connection points. In these points input events are stored in Aurora storage manager (ASM) for some predefined time. Aurora supports adding new boxes and workflows downstream from the connection points, which can be used to create new queries during runtime. The ASM is also responsible for managing the queues needed by the windows in EPAs and buffering messages when they arrive faster than they can be processed. (Carney et al. 2002)

Medusa

Medusa provides a networking infrastructure for distributed processing. For example, it implements a distributed naming scheme for connecting workflows on one site to one on another. It multiplexes different streams to reduce the number of TCP/IP connections to improve efficiency. (Sbz et al. 2003) According to Cherniack et al. (2003), Medusa is most useful in geographically distributed environments, because it facilitates federation and has a market system for sharing workload to different parties. These features are not discussed further, because MMEA Bus is a centrally managed system, and the workload is not geographically distributed to different organisations.

Borealis

Cherniack et al. (2003) envisions a distributed version of Aurora. The system is later implemented in the Borealis project (Abadi et al. 2005). The architecture allows running Aurora queries on clusters of machines located in several administrative domains. I will here describe only the intra-domain parts, because those match best to the goals of MMEA Bus.

Load balancing in Borealis is based on two features: repartitioning of the Aurora networks and load shedding. The repartitioning is done by sliding EPAs from an overloaded node to a neighbour, which has spare capacity. Most of the operators, which don't have massive data dependencies, are easily movable. One remarkable feature of the EPA sliding is that the current windows of data don't have to be lost. After the initialisation all the EPAs might run on a single node and then be redistributed to the other nodes. (Xing et al. 2005)

Because finding an optimal solution for the load distribution is an NP-hard problem and not feasible, Borealis uses a much simpler scheme for sharing work. The optimisation is mainly done on local level. Every Aurora node runs a local optimiser, which schedules

events for processing and makes load shedding decisions. Moreover, every node has a neighbourhood optimiser, which negotiates with the nearest nodes in the network and makes decisions for sliding windows. If these mechanisms cannot bring the performance to an adequate level in a certain time, a global optimiser is triggered to make more thorough modifications to the system. (Abadi et al. 2005)

To make sliding the EPAs more efficient, the system allows splitting heavy EPAs into smaller ones, as illustrated for the sake of an example in figure 2.3. A splitted EPA is preceded by a filter, which divides the load in two streams, which can be handled independently. After the splitted filters there is a union, which combines the streams to one again, and the result stream is equivalent to the one before splitting. (Cherniack et al. 2003)

Borealis uses quite sophisticated methods for distributing the load. Xing et al. (2005) describe a method to minimise the variance of event processing latency. The system produces performance data for the streams, and it is used to calculate correlations coefficients for the load experienced in each of the EPAs during bursts. The optimiser then tries to assign negatively correlated EPAs to the same machine to minimise slack.

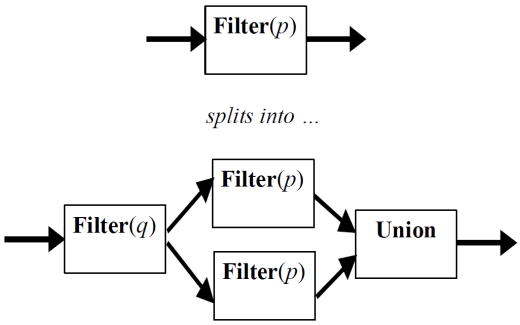


Figure 2.3: Splitting a filter EPA in Borealis (Cherniack et al. 2003)

Borealis also adds some other novel features. Abadi et al. (2005) states three new requirements for a "second generation" CEP. Borealis combines the functionality of both Aurora and Medusa, but also adds support for revisioned query results and query modification. It also enhances the optimisations.

Revisioned query results address the issue that a CEP engine must sometimes operate on incomplete data. The data model of Aurora is an append-only stream of tuples. Borealis adds the possibility to remove and modify already processed tuples later. This is useful sometimes when data producers have some preliminary information on events and get more accurate data later. An enhanced query model includes control lines for query modification. Control lines allow changing parameters of the EPAs, for example giving a new test function for a filter. (Abadi et al. 2005)

2.4.2 STREAM

The Stanford Data Stream Management System (STREAM) is one of the earliest big efforts to create a CEP engine. The greatest contributions of the project are a continuous query language and their adaptive optimisation schema. The STREAM project has been discontinued, but many of the key people have moved to develop commercial applications. (Owens 2007)

Continuous query language

The CQL continuous query language takes its inspiration from SQL. It extends SQL by adding operations, which operate on streams. The streams differ from relations in that they are unbounded in length. The elements of streams also have timestamps denoting the logical arrival time of the event. (Arasu et al. 2006)

CQL is not the only SQL-like language for complex event processing. Many other projects have defined their own languages with stream operators. Some of the most famous languages are StreamSQL, TelegraphQL, Cayuga Event Language and Esper EPL. Because the languages mostly share the same purpose and the starting point (SQL), they are very similar. (Owens 2007) Although it is not published anywhere, it seems that CQL has acted as an inspiration for the EPL of Esper, which makes the research around CQL interesting in our case.

Listing 2.1 shows two example queries as given by Arasu et al. (2006). The first query, Q1, holds maximum value of field *S1.A* from the last 50 000 events. Q1 outputs a tuple containing two values: the field B of the last arrived event and the maximum of A. The second query, Q2, creates a sliding window over two streams. It operates on two windowed streams, S1 and S2, containing the last 40 000 events of S1 and all the events seen in S2 during the last 10 minutes, but no more. The Where clause is used to express that the events selected to the output window must have equal elements in their A fields, exactly like in SQL.

The queries are translated into query plans. The plans run continuously and include three types of components. First, there are operators, which execute some function (e.g. addition). Second, the operators are connected by inter-operator queues. Third, the operators are associated with synopses, which summarise the tuples seen on a given operator. (Motwani et al. 2003)

Figure 2.4 illustrates a query plan that can be generated from the queries in Listing 2.1. Note that the events flow in from the bottom, streams S1 and S2. The queues on top of the picture contain the results for the queries Q1 and Q2. For instance, the plan includes following components: Both queries Q1 and Q2 operate on stream S1 with 50 000 and 40 000 events, respectively, and this adds a sliding window seq-window$_{S1}$ to the plan. The

synopsis Syn_1 contains always the last 50,000 events, because it is the largest of the two. When new events arrive, the operator transmits inserts and deletions to the queues q_3 and q_4 so that both will contain the required events for their next operators. (Arasu et al. 2006)

The aggregate operator on the leftmost path of the plan is created by the max operator in query Q1. When the queue q_3 changes, it selects the maximum of As and maintains it in synopsis Syn_6. Although the operator can work incrementally using the latest inserted A as the maximum value, if it is bigger than the last maximum. However, the operator clearly cannot be incremental in a case where the event carrying maximum A is removed. In the worst case, this requires going through all the last 50 000 events in the queue. Without any optimisation, the operator might store the events in its input synopsis Syn_3. However, because all its contents are always already stored in Syn_1, Syn_3 can be dropped and replaced with a link to Syn_1. The other paths in the plan are described in detail by Arasu et al. (2006).

Listing 2.1: Two example CQL queries shown in (Arasu et al. 2006)

```
Q1: Select B, max(A)
    From   S1 [Rows 50,000]
    Group By B

Q2: Select Istream(*)
    From   S1 [Rows 40,000], S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

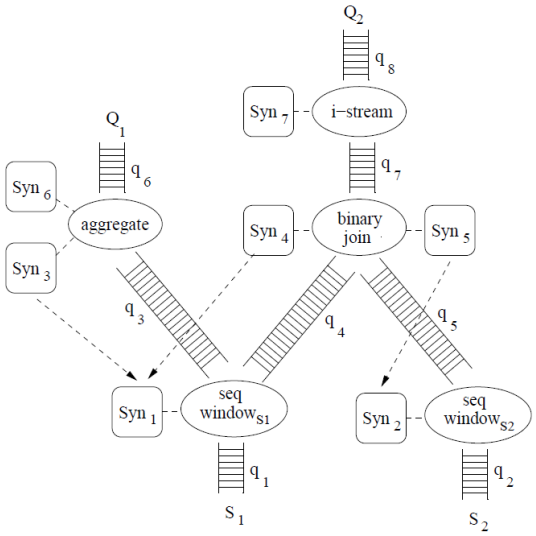


Figure 2.4: An concrete CQL query plan implementing the queries in listing 2.1 (Arasu et al. 2006)

Optimisation and approximation

The developed system includes an optimising query processing engine called StreaMon. This allows the system to maintain its low overhead in event processing and transmission. StreaMon includes an executor running the queries, a profiler as a part of the executor to collect performance data and a reoptimiser to make changes to the query plans and memory structures. (Babu and Widom 2004)

When STREAM becomes overloaded or encounters bursts, it may resort to combined load shedding and query approximation. STREAM supports static and dynamic approximation. Static approximation modifies queries before they are supplied to the query processor. It may reduce window size or sampling rate in operators which use them. In ideal case, the static modifications don't alter the query results at all. (Motwani et al. 2003)

Dynamic approximation leaves the queries unchanged. The dynamic approximation allows the accuracy of the results to vary over time depending on the current load. The dynamic approximation techniques include synopsis compression, sampling and load shedding. (Motwani et al. 2003)

Babcock et al. (2004) note also that the operator on which the approximation takes place has impact on the processing in downstream. Dropping messages mostly lowers the work required in the rest of the query, but sometimes the effect can be the opposite. One example is a sliding window dropping duplicates. Shortening the window makes the set of checked messages smaller and increases the volume of messages emitted.

STREAM is a centralised complex event processing system and runs on only one machine. In the project a need for developing a distributed system was recognised and expected to help with scalability. However, no papers on a distributed architecture or implementation were published. (Arasu et al. 2003, Babu and Widom 2001, Arasu et al. 2003; 2006)

2.4.3 Shared state solutions

Another approach to distributing complex event processing could be a solution based on shared state. JavaSpaces had already been used to issue computations in a cluster (Ku et al. 2008). In their solution they implemented an architecture based on the master-slave pattern, where one master node receives all the events and deals them forward to worker nodes for computation. JavaSpaces implements Linda spaces concept (eg. Gelernter (1989)), where nodes can write tuples to and read them from a logical memory space residing sharedly on the machines. It fits well the master-slave pattern, if the computations can be split into independent pieces.

According to Ku et al. (2008), worker nodes would run local event processing agents (LEPA) over the events generated by the same machine and also domain event processing agents (DEPA) to detect composite events outside the scope of a single machine.

Depending on the implementation details, the shared state solution could open up many interesting opportunities for optimisation. The first is multiquery optimisation (Heinze 2011). MQO searches for common parts in simultaneous queries. It has been extensively studied with database management systems (Sellis 1988). Because the EPL queries are run continuously and for long time after they are issued, MQO can be economically leveraged to a much greater degree than in a DBMS, where query is issued once, lasts less than a millisecond and then triggers and return only once.

Although (Ku et al. 2008) observed the system to scale linearly with the five nodes tested with, there is an obvious bottleneck in the master-slave pattern: the master. All the traffic must go the single node. Also in the implemented architecture the master node did some intermediate computations, which still adds to the burden of the master. Because of the nature of CEP, I believe it can be very difficult to remove these computations, as the dependencies of events can be very interwoven. It must also be noted that the performance results reported by Ku et al. (2008) are very vague and also the complex event detection rate seems quite poor, only 27 events per second. The paper is quite unclear on what they mean by a detected event.

This inspired us to consider group communication for sharing the state of the CEP engine. Apache Tomcat includes Tribes cluster communication module, which could be used for accessing objects on other nodes. Tribes would have been an ideal candidate, because it supports forming groups based on well-known-agents in contrast to the more popular multicast agent discovery, because AWS EC2 doesn't support multicast. However, modifying Esper engine to be shareable over Tribes turned out to be too big a task. Also there are some closed source add-on modules for Esper and in their case modifications wouldn't even be possible (EsperTech Inc. 2011).

2.5 Performance testing CEP

There are a couple of proposed benchmarks for performance testing of CEP. For the performance tests explained in chapter 5 I draw mostly from Linear Road (Arasu et al. 2004) and BiCEP (Mendes et al. 2008). They both describe a selection of standardised test cases that all CEP products supposedly are able to perform efficiently.

Linear Road was one of the earliest benchmarks, and it was endorsed both by Aurora and STREAM projects (described in sections 2.4.1 and 2.4.2, respectively). It simulates a road toll system in a simple city with multiple expressways. The toll system is required to detect accidents and congestion from the traffic data on the fly. It must also calculate

demand based toll prices. The traffic monitor must also support queries for historical data and the patterns appearing in it. (Arasu et al. 2004)

The test results by Arasu et al. (2004) suggest that CEP systems are superior over database management systems. The paper shows that Aurora topped the an unnamed DBMS by being able to process five times more traffic data during the specified time. Furthermore, the latencies of Aurora stood quite low even with full load when the DBMS could not produce results any more.

Mendes et al. (2008) describe FINCoS framework for evaluating CEP performance. The FINCoS tool includes load generators and sinks for performance testing. It includes a plug-in driver for Esper CEP engine and offers a Java API for developing custom drivers for other engines. The FINCoS framework was used to study the performance of three different CEP systems in micro benchmarks testing the most fundamental stream processing features of CEP. (Mendes et al. 2009)

Both Arasu et al. (2004) and Mendes et al. (2009) detect the following challenges in performing performance tests on and creating a benchmark for CEP. First, the CEP systems can produce multiple different correct results. This leads to challenges in the output validation. Second, there are no standards stating what features a good CEP product must have. Third, the metrics for results can vary by the application domain. Also mentioned are the lack of common query language and creating semantically valid input data.

Chapter 3

Architecture for complex event processing

The requirements are defined by a Tekes financed MMEA project work group. The vision of MMEA Bus includes developing an interoperable and scalable integration platform with complex event processing capabilities.

In this chapter I discuss the architecturally significant requirements (ASR) for the platform. I present an architecture enabling complex event processing in MMEA Bus.

3.1 Architecturally significant requirements

The categories of non-functional requirements for MMEA Bus consist of 1) scalability 2) high-availability and 3) ease of configuration and management. In this section I explore what these requirements mean in our system. I describe their significance and the possible solutions in the respective order. I also consider conflicts and trade-offs that might arise between them.

3.1.1 Scalability

The primary concern in this thesis is *throughput*. In the most optimistic business scenario there could be millions of sensors and other data sources feeding the system new data every second. This requires us to be able to provide either very powerful computing units for complex event processing or to be able distribute the workload over a large set of machines. Because there is always a limit to how big machines are available, scaling out (adding more parallel machines) is the only solution, when the system grows big enough (Michael et al. 2007).

Another concern is the event processing *latency*. Alerts and other perishable information must not be delayed because of the centralised system generating and mediating them (Wächter et al. 2012). For a single statement Esper can have latencies as low as a couple of microseconds (EsperTech Inc. 2007). There are also solutions that can handle hundreds of simple statements in 10 microseconds (Cugola and Margara 2012). Anyway, these latencies are small enough that they wont matter in a system that operates over the Internet with network latencies in excess of milliseconds.

To make the satisfying of the latency requirement measurable, I take 50 ms as a practical time limit to detect a triggering event. The starting time shall be the timestamp issued by the input queue, when the message arrives in the system. The ending time shall be determined by the output queue the system uses to send the generated complex event forward.

The rate of production of the data supplied to the system may not be a constant. In Amazon EC2 cloud the instances are billed by a machine-hour, so we can potentially reduce costs by not having any more resources than needed (Amazon Web Services LLC 2012a). A CEP system can be designed to scale up and down *elastically* (Heinze 2011). AWS provides developers with tools to programmatically add and remove instances to the cloud (Amazon Web Services LLC 2012b). This enables us to spawn new instances of similar computing units to handle the increased load and terminate the ones that are no longer needed.

Because most of the target data is generated by environmental sensors or batch driven computational models, the required throughput is fairly constant. However, for the periods longer than few hours it is conceivable that there is some variation, which might be periodic or aperiodic. For example, a periodic variation in the volume of events might arise weekly (e.g. during weekends many office buildings are empty and frequent updates from them are not needed) or yearly (seasonal changes in nature might impact the amount of data sensors produce). Periodic variation with known periods can be addressed by scheduled up and down scaling.

Acyclic variations in required throughput are harder to address, if the rate of changes is relatively fast and unexpected. If this case, the up and down scaling must be automated. *Autoscaling* raises several new issues. Some entity must coordinate the addition and removal of nodes. They must also be well defined metrics on which to base the decisions of scaling. Examples of metrics offered by AWS are messages per second and average request latency. (Amazon Web Services LLC 2012b)

In a distributed environment *load balancing* is an interesting issue. Especially in complex event processing when there are multiple interdependencies between the events, load balancing is not a trivial issue. The processing must be distributed evenly on the processing nodes. (Lakshmanan et al. 2009)

In a case where load spikes up quickly for a short time, the system must not choke under the flood of messages but to queue them up in a buffer. *Bursts* might appear for instance when some networking equipment is faulty or disconnected temporarily and a set of messages are released simultaneously from a lock-up. Bursts also affect decisions on autoscaling, because they might temporarily degrade performance metrics, even though the system could cope with the load and scaling up would be unnecessary. (Kleiminger et al. 2011)

3.1.2 High availability

Bursts are also an important issue from the high availability viewpoint, especially because they may be more usual during a crisis time. It is also the time when the information is most valuable, and in an emergency system errors or delays in processing may result in loss of lives. The effects of bursts can be mitigated by overprovisioning or by allowing the events to accumulate in a buffer. (Kleiminger et al. 2011)

When the load on the servers increase dramatically in a short time, some CEP systems have resorted to load shedding or approximate query answering (Gurgen et al. 2005). Load shedding in an emergency situation would require knowledge about which messages can be safely dropped.

Other sources of unavailability are hardware failures and networking issues. The more there is hardware, the bigger is the probability that at least some piece of it breaks down. In distributed systems the failures are often partial, which means that only some processing nodes or links are down. In many cases these failures can be masked or detected and tolerated. (Coulouris et al. 2011; pp. 37-38)

Although high availability issues side closely with the scaling issues in this thesis, I have limited them outside the scope of my research. Some sophisticated HA and QoS can be implemented as extensions to the system described here, especially to the ESB.

3.1.3 Configuration management

I describe here some configuration management requirements for the system to give a clearer picture on what the end product would look like. The issues described here are out of the scope of my research question. Nevertheless, they are anyway a very visible part of the system, and I discuss some possible solutions in the architectural design later in section 3.4.

Data sources and streams

The MMEA Bus is an integration product and will be connected to multiple data sources and endpoints of multiple customers. This requires the software to be easily configurable, such that new data sources and customers can be easily added, removed and reconfigured.

The different data sources in the system will be presented as streams to the users. The users can then subscribe to the streams and define their own CEP rules on a combination of those streams. The complex events produced by the CEP are then shown as their own stream and is again subscribable.

Access control

These streams require some kind of access control, because some derived events will be to private use. This is closely connected to the other user management issues. The users must be authenticated. In some cases the data flowing in and out the bus should be encrypted to prevent eavesdropping, tampering and forging events, preferably always.

CEP management

Carney et al. (2002) recognise a need for three types of queries: continuous queries, views and ad hoc queries. These were previously discussed in section 2.4.1. In MMEA Bus the aim is to first support continuous queries, because they can be executed without storing a history of messages. There are plans to incorporate a storage system to MMEA Bus, but the implementation falls out of the scope of this thesis.

To be usable in a wide range of applications, the CEP system must support defining new queries during runtime. In our initial plan CEP would be implemented as an complex event network, which means that we must support both the creation of the networks and the agents of those networks.

3.1.4 Other non-functional requirements

The potential customers and users of the MMEA Bus must be *billed*. This requirement is out of the scope of this thesis but can be taken into account when designing the architecture. The customers might want their costs to vary with their real usage of the integration service, which can be tracked with AWS tools.

In a distributed system operating over Internet, *security* is a key issue. The fine-grained access rights and application level security is best addressed in a separate security component in the ESB and is not considered in this thesis. However, for transport and network

level there are some points that must be taken into account when developing a platform. For example, our architecture must allow the usage of WS-Security add-on (OASIS 2006). On transport level it must be possible to use HTTPS. Furthermore, Amazon Web Services provide network level security groups for creating a simple firewall. Security groups can be used to block connection attempts from the outside world and to allow access from only the other nodes in the same cluster (Amazon Web Services LLC 2012b).

3.2 Eight rules for stream processing

In their seminal article, Stonebraker et al. (2005) introduce eight requirements for a real time stream processing system. These are all highly relevant to the MMEA Bus, and in the design and implementation I will follow most of them. Here are the eight rules in brief. I will return to these rules in section 5.8 and evaluate the implemented system in respect to them.

The first rule is to *keep the data moving*. Disk storage access is many orders of magnitude slower than any other activity in computing. Persisting events to a disc is often used as a fault tolerance feature. However, recomputing the lost data can often be faster option. The second rule prompts to use an *SQL-based language*. A domain specific language makes the system much faster and easier to develop and maintain.

The third and fourth rule deal with *stream imperfections* and *predictability of outputs*. In a networked environment messages can be dropped or arrive out-of-order. These effects can result in wrong outcomes in an unsophisticated system.

Rule five says to *integrate streams and stored data*. Users often want to compare past and present data. This is a very hard requirement to implement efficiently and also a bit in conflict with the first role, keep the data moving.

The last three concern the nonfunctional qualities of a CEP system. The data must be always *available and its safety must be guaranteed*. The system should *partition the workload and scale automatically*. Finally, the CEP engine must be *highly optimised* to handle all the data in real time without building up buffers or delaying processing.

3.3 Modelling MMEA Bus as an event processing network

To better understand the semantics of the required complex event processing system, I drew the event processing network corresponding to it. Figure 3.1 shows the MMEA Bus as an EPN. The defined topology draws heavily from Luckham (2001; p. 208) and adds a feedback loop like discussed by Adi et al. (2006). Effects of loops in event processing network were also discussed in by Etzion and Niblett (2010; p. 117).

Sharon and Etzion (2007) identify two traditional infrastructure types, which an EPN can be built on: Messaging Systems and Data Stream Management Systems (DSMS). The messaging system is closely related and often based on an event-driven service oriented architecture. ED-SOA has all the benefits described in 2.1.1, but implementations often lack the CEP capabilities, as noted by Wishnie and Saiedian (2009). DSMSs are readily capable of running continuous queries and other processing on the data. Thus it is already very ready component for an EPN.

Another model Sharon and Etzion (2007) present is a hybrid of a Messaging System and DSMS. In MMEA Bus we try to bring the best of both worlds by integrating CEP in an ESB. ESB provides us with flexibility in enterprise application integration and a custom stream processing system enables efficient complex event processing, expressed in an EPL.

Producers: All the events handled by the system are produced by environmental sensors. Because of its nature as an integration platform, we could of course allow also other event sources, if they would add some value to the users. Nevertheless, even then it might be sensible to view the new sources as "special sensors", as our sensors already can be a very heterogeneous set.

Adapters: An enterprise service bus by definition supports a wide variety of data sources. In our current implementation the most prominent protocol for submitting messages for processing is SOAP. Data is represented as XML, which can be adapted to our internal format using Extensible Stylesheet Language Transformations (XSLT). The SOAP messages are typically sent over an HTTP or HTTPS binding (Gudgin et al. 2007), and that is the case in MMEA Bus too. Another possible route for message transmission is Java Message Service (JMS).

Filters: Inside the ESB the relevant events are selected and forwarded to an event processing network constructed by a user. For scalability reasons the filtering step must be completely stateless, because we cannot give any guarantees on which CEP engine an event flows.

Pattern detection: Pattern detection can be defined arbitrarily by a user. Users can define their own event processing networks to handle complex processing. Events can be fed back to the filtering step to allow processing them again by the same or some other EPN. This acts as a feedback loop, as noted by Etzion and Niblett (2010; p. 117), and requires special care. By always forwarding back to the system the same events we can easily construct an infinite loop.

Consumers: The ESB provides us with several prebuilt and custom services, such as notification service and storage. These can be universally used by supplying events with correct structure to the ESB. These services act as event consumers, because we cannot influence the event flow at that stage any more. Nevertheless, for example the data stored in a database can be activated later, but later instantiations of data shall be considered separate data.

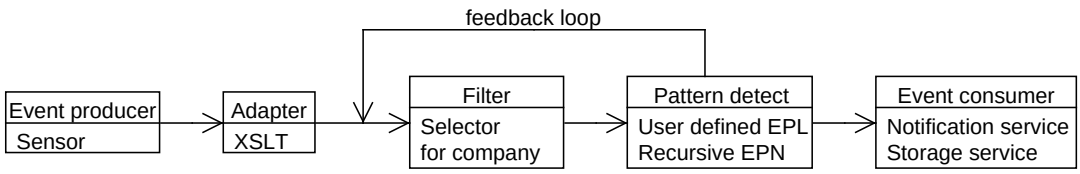


Figure 3.1: An event processing network containing the basic building blocks of the MMEA Bus and example functionality

3.4 Scalable architecture for MMEA Bus

The architecture implemented for this thesis is based on distributed, stateless instances of enterprise service bus that forward filtered events to a dedicated complex event processing cluster. Figure 3.2 shows the division of the responsibilities between the ESB and the CEP cluster, the physical setup and the message flows between the machines. The implementation of the architecture is described in chapter 4.

The architecture allows us to exploit the strengths of both the ESB and the CEP separately. The ESB is used to define adaptations and transports that operate only on one single message at a time. The scalability model utilises parallel, stateless instances to the extreme, and there is very little the ESB instances must know about each others.

In CEP the data dependencies play much bigger role. Thus we want to limit all the extra work done on the CEP cluster to the minimum. This means that the events supplied to the CEP cluster are already in the correct format and free from other complications, such as access rights management or encryption. The sole purpose of the CEP cluster is to execute the pattern matching and other fundamental functions of CEP.

3.4.1 Event flow in the system

Figure 3.3 depicts a logical flow of the events inside the system. Events produced by the sensors are fed to the enterprise service bus, which converts them to an internal format in the adapters. The variety of possible protocols for sending events to the ESB is wide, because supporting a new protocol in the ESB only requires writing a new adapter. The currently supported protocols are SOAP over HTTP, HTTPS and Java Message Service (JMS) with various data source dependent file formats.

The events received by the ESB are run through a series of filters, which act as selectors for interesting data. The filters are stateless and defined by the users of the data. By defining filters the users are able to subscribe to events and event streams they are interested in.

In fact, they implement the first stage of an event processing network and can be seen as event processing agents, as illustrated by Luckham (2001; p. 208) For example, filters could be implemented using XPath, which matches the interesting messages by some of its elements. If there are some restrictions (e.g. usage or billing limits) to the event streams the users are allowed to subscribe, they must be applied before these selectors.

The selected events are forwarded to a message queue, which is read by a Storm cluster. The Storm cluster consists of spouts, which read the message queue, and bolts, which run the data through EPL statements (i.e. are event processing agents) or forward the events back to the ESB for further processing (i.e. act as local event sinks). The basic Storm terminology is be explained in sections 4.2.1 and 4.2.2.

The architecture fully decouples complex event processing from the enterprise service bus. This enables us to handle their scalability separately. In fact, we could even replace the whole event processing system built on Esper and Storm with something completely different. The only requirement is an event-driven architecture and ability to interact via message queues.

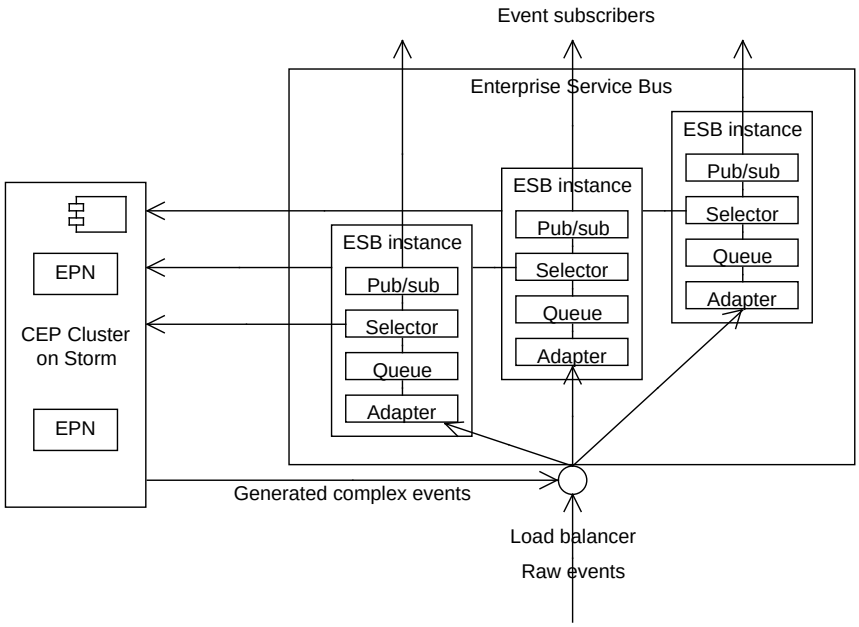


Figure 3.2: ESB is distributed on multiple cloud machine instances that forward events to a specialized CEP service

3.4.2 Distributed CEP service

The distributed CEP service aims to provide a very flexible platform for users to define their own event processing networks. There are three kinds of nodes in our EPN. First,

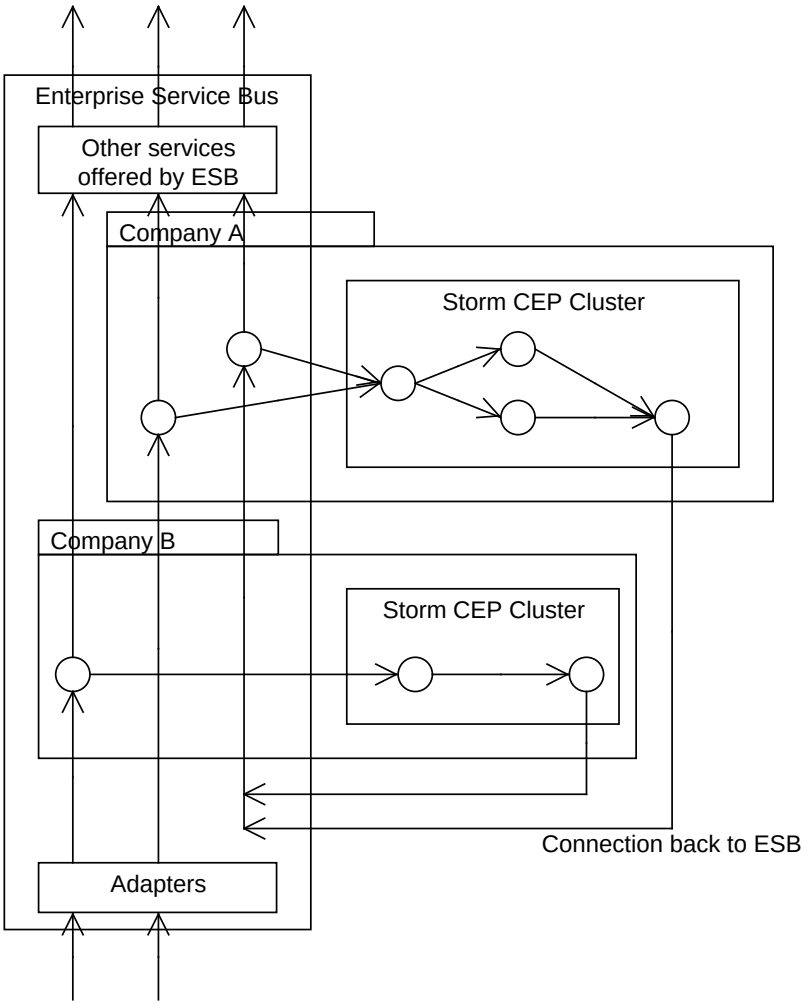


Figure 3.3: Closer look to the processing model: stateless ESB instances forward events to Storm clusters

there are event producers, which read tuples from a message queue. Second, these tuples are forwarded to the EPAs all running an instance of Esper engine. There may be several statements running on one instance but it is discouraged, because it inhibits parallelism, as explained later. The third type is a local event consumer, which acts as a leaf node in the network and forwards the received events back to a message queue of the ESB for further processing. Figure 4.1 should give a clear picture on an example network, presented as a Strom topology.

The selected level of distribution in this architecture is the engine level. It is a natural choice, because Esper does not allow distributing a single engine on multiple machines. This decision imposes some limitations on our implementation. As discussed by Heinze

(2011), engine level granularity makes multiple query optimization (MQO) less powerful. MQO would help pruning unnecessary, overlapping computations, when multiple queries have similar parts. Still, the queries running on the same machine could make use of MQO locally, if the CEP engine supports it.

Because we have multiple separate CEP engines running on different machines, the only way they can share data with each other is by messaging, that is, by creating and consuming events. This creates some overhead but makes implementation much simpler. This approach forces the user to think the event processing as a flow of events and define the desired computations as a network.

The architecture still leaves some requirements unaddressed. There is no support for automatic elasticity. When one wants to increase the size of a Storm cluster, the message processing must be stopped for a moment. Another issue is that we currently cannot control the internal load balancing of Storm.

3.4.3 Configuration and deployment management

Because the ESB and CEP cluster are completely decoupled, they don't share any configuration. The ESB usually uses its own registry for configuration. The CEP cluster is backed by another database.

ESB configuration

The responsibilities of the ESB include adapting data sources, identifying streams, filtering events in the selectors, forwarding streams to their subscribers and integration with other services like user management.

The adapters in the ESB are stored centrally in the configuration and governance registries. This allows all the instances to share same configuration and act identically. The implementation is discussed in section 4.1.1.

For identifying and selecting events in the ESB, every event belongs to a stream. This stream can then be selected for complex event processing in conjunction with an XPath selector or to be sent to stream subscribers. Note that the selectors are part of an EPN, even though they are not run on the cluster. They implement the stateless filter layer of an EPN. This means that the developer defining an EPN for the CEP cluster must also select the required events for processing from the ESB.

Another point of configuration in the ESB is a subscription management for external applications. The system must be able to forward raw and complex events to external

users, so that they can build their own applications based on them. This service is still under development.

The stream identification information, metadata and the subscriptions are served by a separate web service called Stream Catalog. The service is accessed through a proxy service defined in the ESB, which makes the service seem like an undistinguishable part of the MMEA Bus. The data storage used by the Stream catalog can be chosen independently from the rest of the system, because the interface of the service doesn't have to change. The ESB can retrieve the subscription lists from the service periodically.

CEP configuration

The requirements of the CEP service state that one must be able to define event processing networks with replaceable event processing agents. Because the EPNs are defined as Storm topologies, changing them is not possible. However, we can program the single bolts to be able to change the EPL they are executing on the fly.

I designed a web based cluster configurator that could be used to define Storm topologies and to deploy them to a Storm cluster via the API provided by Storm. The topologies could include arbitrary number of data sources reading events from the ESB, EPAs, sinks sending events back to the ESB and connections linking these components. Because modifying a topology would require first completely removing it, modifications to a running EPN would not be supported. A brief prototyping attempt is described in section 4.2.8.

Anyway, single EPAs can be modified while a topology is running. To make this possible, the EPLs are stored in a database accessible from all the nodes of the cluster. The database backed EPAs would then periodically poll the database to retrieve the possible changed EPL.

Chapter 4

Implementation

The biggest contribution of this thesis is the complex event processing platform, MMEA Bus, which was implemented following the architectural design presented in 3.4. The system consists of two separate parts: a distributed enterprise service bus and a complex event processing cluster. This approach enables us to address the radically different scalability requirements of the two components.

The users of the data streams can define simple filters selecting events they are interested in. These filtered streams are forwarded to a CEP service that implements a high-performance, customizable event processing network. In this section I show how to implement the architecture on WSO2 ESB and Storm stream processing cluster with Esper event processing engine.

The implemented system makes use of a collection of open source software. The ESB is built on WSO2 ESB, which in itself includes many other open source projects: Apache Tomcat runnin Axis2 for web services and Synapse for providing configuration and integration features. In addition, there is Apache Tribes for group communication between the ESB nodes.

The CEP cluster is built on Storm stream processing system, which offers a distributed framework. CEP capabilities are provided by Esper. The JMS of choice is ActiveMQ, but also a subproject called ActiveMQ Apollo was tested. For message transmission from ESB to CEP ZeroMQ was used, because it offers superior performance. These software products are further explained later, when I explain how they are used in the system.

4.1 Distributed ESB

The basic infrastructure for MMEA Bus is provided by the WSO2 ESB. WSO2 is a software vendor specialising in enterprise middleware. It has developed and packaged

many software products, which can function as components of an enterprise system. Most of the products and solutions offered by WSO2 are based on open source projects and open standards. (WSO2 2012) Specifically, WSO2 ESB is built on Apache Synapse, to which WSO2 has added their own package management system and a management console.

The ESB cluster consists of multiple separate EC2 instances, whose awareness of each others in our setup is only limited. In my design there are two kinds nodes, masters and slaves. All the nodes run CentOS 5.5 and WSO2 ESB. In addition, the master nodes run WSO2 Governance Registry (G-Reg) and PostgreSQL database management system.

The purpose of a master node is to manage the configuration of the ESB. WSO2 ESB offers a web console, which can be used to define endpoints, mediation sequences, tasks and other services provided by the ESB. The only difference in the WSO2 ESB configurations between masters and slaves is that the slaves have read-only rights to the shared registries that carry the configuration information.

The architecture supports more than one master node. In the case of multiple masters, one of them acts as an "active master", meaning that it holds the Elastic IP, and the others are "passive." The passive masters act normally like the slave nodes mediating the messages, but they are ready to receive the Elastic IP and take the role of the active one, if there is a failure in the previous master instance. The databases running on the passive masters are replicated from the active master.

For the deployment on AWS EC2, I have created two elastic block store (EBS) images, one for the master nodes and one for the slaves. When booted, the slave (and passive master) nodes retrieve the configuration from the master node. Although WSO2 promises that the ESB instances are able to copy the configuration of the master (Azeez 2008), I encountered some severe bugs in it: the deployed JARs do not get copied to the newly starting nodes, which did not previously have them. Nevertheless, the bugs could be worked around with some start-up scripts, which copied required libraries to the booting slave node. The modifications done to the WSO2 ESB during runtime replicated correctly to all of the slaves.

4.1.1 Registries

The registry of WSO2 ESB consists of three parts: local, configuration and governance. The local registry contains runtime data and system configuration that is local to each running ESB instance. It resides always on the same machine as the software that uses it and nothing of it is shared. (Fernando 2010)

The MMEA Bus runs a WSO2 Governance Registry (G-Reg) as a separate service on the master nodes. In our setup, G-Reg maintains the configuration and governance registries, like in the strategy B explained by Fernando (2010), and it is stored in the PostgreSQL

database running on the master nodes. The configuration registry includes the definitions for mediation sequences, proxy services, queues and other activities performed by the ESB. It is also used to distribute Java archives containing the bytecode of our own custom mediators.

The governance registry carries SOA metadata for the use of the ESB (WSO2 2011). For example, it includes the XML schema definitions for the internal messages used by the bus. In the future, it could also be used to serve metadata of the data sources. The local registry is run in the embedded Derby database bundled with WSO2 ESB.

4.1.2 Communication between ESB instances

There are still some applications where the ESB instances cannot operate completely independently of each others. One is the polling of external services. Because all data sources are not able to push their data to the ESB, the platform must fetch it from a remote server by periodically checking if there are new messages and then retrieving them. To make sure that the polling task runs only on one ESB instance, they must somehow decide the host for it.

WSO2 ESB is based on multiple open source projects maintained by Apache Software Foundation. It comes with Apache Tribes, which is a group communication module for Apache Tomcat (Apache Software Foundation 2011b). Apache Axis2 web service engine (Apache Software Foundation 2012) provides several shared contexts in a hierarchy, where members of the groups can push objects and from where they can then be globally retrieved, and Tribes makes communicating the contents of these objects to the other members of a group.

Group communication was used to prevent multiple ESB instances from launching a poller and to monitor the state of the polling node. When each of the nodes starts up, they check in the shared context, whether there is a poller already running. If not, it registers itself as the poller and then, from time to time, updates a timestamp of the previous polling attempt. The other instances periodically check, whether the instance is still up by comparing the timestamp to the current time. If the timestamp is too old, the polling node is deemed failed and the next instance registers itself as the poller.

Group communication was first also planned for distributing objects in complex event processing. However, because of the heavy changes required in Esper and the lack of distributed locks in Tribes (Apache Software Foundation 2012), this line of research was abandoned. There were also some small hindrances with Tribes. According to the documentation of Axis2, it is supposed to handle all the calls to actuate the communication over Tribes (Apache Software Foundation 2012). However, in the mediators of WSO2 ESB the calls never happened but must be forced by adding extra calls to the replicator module.

The machines belonging to a group must first find each others to be able to communicate. The first choice for identification in a group on a local network is a multicast membership query. In addition to easy discovery, multicast also offers superior performance in message transmission, because it saves bandwith by sending single messages that can be received by multiple recipients. AWS EC2, however, doesn't support multicast (Amazon Web Services LLC 2012b), so it cannot be used. Another membership scheme offered by Tribes is using well-known addresses (WKA). A WKA allows nodes to first connect to a pre-issued network address belonging to the group, which then answers by giving the list of all the other nodes in the group. In my setup, every node knows the Elastic IP issued to the master.

Another solution for poller selection and other synchronisation issues would have been to use the governance registry. Governance registry could facilitate the development especially when there are multiple stakeholders making use of the shared contexts.

4.1.3 Load balancing

AWS Elastic Load Balancing (ELB) (Amazon Web Services LLC 2012c) is used as the load balancer for the ESB cluster. ELB passes through HTTP and HTTPS requests to the proxy web services running on the ESB instances. Load balancing works in a round-robin manner selecting the next node by taking the node with the lowest latency.

The EC2 instances running the ESB were configured to automatically register with the ELB. The registration was done in a boot script. Also, a deregistration script was added to the shutdown. Nevertheless, the ELB keeps account of the latest performance of the nodes and marks them as failed, if they don't respond in time.

If needed, ELB could also coordinate autoscaling. With Amazon Cloud Watch one can define metrics to monitor and threshold to decide, when to add new EC2 instances and when to remove. However, this was never tested. At least the scale-out should work fine, but the decision on which nodes to remove when scaling back would need some work. In principle, the removal procedure would consist just of deregistering the node from ELB and waiting until all the messages in the queue are all processed.

4.1.4 Adapters

Adapters are the first component of MMEA Bus, which are part of the complex event processing network, as described by (Luckham 2001; p. 208). Adapters are implemented as mediators in WSO2 ESB. They are deployed to the ESB and their purpose is to supply events in our internal XML message format to the selectors. In the simplest case the incoming messages are already in our format and a plain schema validation is sufficient.

Often the input is a some kind of an XML document and an XSL transformation can be specified to convert the messages into correct format.

Because an ESB is expected to handle all kinds of integration patterns, the adapters can be very powerful, if needed. WSO2 ESB allows implementing mediators and tasks in unconstrained Java, which we have exploited in our implementation. For example, some data producers allow fetching the data over FTP in their own CSV format. This data can be polled with a poller task and then parsed to an XML format with a Java mediator.

4.1.5 Selectors

Selectors supply messages to the CEP service according to user defined rules. They are implemented as mediators in the ESB. They correspond to the filters in (Luckham 2001; p. 208). The selectors can be defined in XPath, which are fetched from PostgreSQL database on the master node. There is not yet a user interface for defining XPaths, but the plans for one are described in Section 3.4.3.

The selectors first check if a received message matches with some XPath rule. Then the corresponding consumers for the rule are looked up. As explained previously in Section 3.4, the consumers are event processing networks running on the Storm stream processing cluster, and a consumer is called a CEP service. The selected events are sent over ZeroMQ to the service. The ZeroMQ sockets use a push-pull pattern, which is suitable for pipelining processing in this way (ØMQ community 2011). As another choice I could have used publish-subscribe pattern, but in my case with only one or a few consumer the configuration would have been more complicated with no extra benefit.

The selectors are fully stateless. Initially, I planned to use Esper in the selectors too. However, there is no real use for the Esper engine at that point, because we cannot give any guarantee on which events flow through a certain node. Anyway, it can still be considered as an option, if some need arises.

4.2 Complex event processing cluster

A complex event processing cluster was implemented in as a part of this thesis. In the cluster, instances of Esper CEP engine are run on a distributed real time computation system called Storm.

In this section, I explain how Storm works, the basics of Esper, and how I use Esper on Storm. I also demonstrate how a complex event network consisting of small and fine grained event processing agents is a very natural way of designing and implementing CEP. Such an EPN can also be distributed efficiently on a cluster of computers.

4.2.1 Storm

Storm is a distributed stream processing cluster framework developed for near real-time computation. It aims for extreme scalability while still guaranteeing fault tolerance and lossless data. Storm is often described analogous to Hadoop, with which one can easily parallelise MapReduce batch jobs. (Marz 2012c) Though excellent in stream processing, Storm lacks any built-in complex event processing features. Nevertheless, the focus on scalability and reliability make it a very usable platform to extend with CEP.

The computational model of Storm relies heavily on the notion of a topology. Topologies are essentially flow graphs similar to ones used in Borealis, described in Section 2.4.1. The nodes of a topology are very powerful parallel processing units, which can be deployed on a cluster of computers. Storm provides a command line utility for controlling a cluster (e.g. deploying topologies) (Marz 2012d). For creating a Storm cluster in AWS EC2 one can use storm-deploy, which includes an automatic setup for the required components of a cluster (Marz 2012e). There is also possibility to test a topology locally, or even a production system could be run only on one system, if no scalability or fault tolerance requirements are set.

Storm was initially developed by BackType, which was subsequently acquired by Twitter during summer 2011. It is free software and published under Eclipse Public License (Marz 2012a).

4.2.2 Components of Storm

The Storm tutorial (Marz 2012d) explains the basic concepts very clearly. Storm cluster has three components, Nimbus, supervisors and ZooKeeper. Nimbus is the master node and is comparable to Hadoop's JobTracker. It distributes code to the cluster, assigns tasks and monitors for failures.

Supervisors are run on every node participating in Storm. It listens to the commands from Nimbus and manages worker processes. Each worker runs its own part of the topology assigned to Storm, as explained later.

ZooKeeper (Apache Software Foundation 2011c) cluster handles the coordination between Nimbus and supervisors. It offers often needed primitives for implementing synchronisation, configuration management, groups and naming in distributed systems. Although ZooKeeper plays very central role in Storm, no messages from node to node pass through ZooKeeper, as it would grow a bottleneck fast.

The computations performed on Storm are defined as *topologies*. The topologies consist of spout and bolt nodes. *Spouts* are the source of data in a topology. *Bolts* can perform arbitrary functions on the data streams. Spouts emit data as tuples to the topology in

a stream. A stream is an unbounded sequence of tuples. For example, a stream emitted by a spout in MMEA context could consist of CO_2 level measurements. A bolt reading the stream might calculate the average level over last five minutes and push the average downstream. There is no limit on how many streams a bolt may subscribe to.

The idea is that the spouts and bolts can be run as parallel tasks, which is shown in figure 4.1. The figure exhibits an example topology, with two spouts and four bolts, drawn as boxes. Both spouts are run as two separate task (depicted as circles), which implies that there can be in total four separate machines supplying data to the topology. The level of parallelism on bolts depends on their data dependencies.

In addition to their functional difference, the biggest implementational difference between a spout and bolt is in the fault tolerance features. If there is a failure detected in some stream and some tuple is missing, *fail* method on the respective spout is called, and the spout is expected to replay the respective upstream tuple, if necessary. The missing tuple is then reprocessed by all the downstream bolts. However, because the fault tolerance model is based on reprocessing, it has some disadvantages in complex event processing. Namely, some messages may be observed multiple times on some nodes in a stream. To address this issue, there are also methods to guarantee at-most-once semantics for message processing. (Marz 2012b)

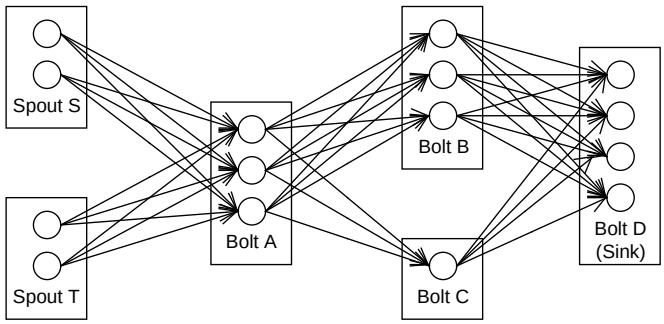


Figure 4.1: A storm topology showing multiple spouts and bolts (boxes) running several parallel in several tasks (circles) on separate machines

4.2.3 Esper basics

Esper is a complex event processing engine. It is written in Java and can be included as a JAR archive in any software supporting Java bytecode. Esper is developed by EsperTech Inc. and it is available in both Java and .Net versions. The core of Esper is licensed under GNU GPL, but there are also proprietary parts. The most interesting add-on is EsperHA, which provides resiliency to Esper, which otherwise has no fault tolerance capabilities.

There are also extra input adapters and graphical tools for EPN definition. (EsperTech Inc. 2011)

The basic interfaces provided by Esper are *EPServiceProvider*, *EPStatement* and *UpdateListener*. EPServiceProvider represents an instance of Esper engine. It provides means for defining EPL statements as EPStatement objects. In Esper one EPL is equivalent to an EPA. An EPL can be an INSERT statement, which creates a new stream inside the engine, and the stream can be read by the other EPLs in the engine. Note that in Storm we have streams on different level, flowing between the nodes of a topology. Also one bolt running (possibly distributed) Esper engine corresponds to an EPA in an EPN spanned by a topology. This is an example of a recursive EPN, which was discussed in Section 3.3. (EsperTech Inc. 2012)

Events can be input to the engine as Java objects, Java Maps, Java Object[] arrays or XML documents (EsperTech Inc. 2012). Because our internal format in MMEA Bus is an XML document, the last solution sounds good. In an early phase of the development, when we experimented with running Esper inside the ESB as a mediator, we used to send events to the engine in XML format. This worked fine, but in the final implementation events are first efficiently converted to Java objects to maximise performance.

To act on the events triggered by an EPL, an EPStatement must be associated with an UpdateListener. UpdateListener is an interface, which defines an update method that is always called, when the selection matched by an EPL changes. An UpdateListener has access to the events, which have just entered the selection, and those, which have just been removed from it. (EsperTech Inc. 2012)

Although Esper can be run without any configuration, usually at least event definitions are given to the engine. Event definitions can be given programmatically as Java classes, as an XML schema or as an Apache Axiom XML definition. With Esper as a mediator, we tried defining events with Axiom, because it could have been embedded in the configuration registry of the ESB (Section 4.1.1). However, the Axiom interface in Esper cannot handle arrays at all, which is a serious limitation in our use case. (EsperTech Inc. 2012)

4.2.4 Esper on Storm

Esper is run on special bolts on Storm. Because there was already a working implementation aimed to bring Esper's functionality on Storm, I based my work on it (Dudziak 2012). Each *EsperBolt* defines a number of EPLs and their inputs and outputs. When the topology and its bolts are deployed, every task running EsperBolt has its own instance of Esper engine.

In Storm the UpdateListener can be very simple. Its only purpose is to emit the matched events downstream. Because Storm needs all the fields of the emitted tuples to be named,

the names for output fields are given during topology construction. The same mechanism allows defining multiple streams, where events can be pushed to. The field names link directly to the names used in EPL statements. In EPL one can name elements in SELECT clause with the AS keyword (like in SQL) and the element names must match the names used in the Storm topology.

4.2.5 Input and output

As described in Section 3.4, the communication between the ESB and CEP service is conducted over ZeroMQ and JMS. Selectors push events to the CEP service in over ZeroMQ and the CEP service transmits complex events back to the ESB over ActiveMQ JMS. The CEP cluster supports all three major datatypes of MMEA Bus, *SensorData*, *ForecastMessage* and *ComplexEvent*, as both input and output types.

ZeroMQ spout

ZeroMQ acts as an interface to the CEP. For approximately every worker node of Storm, one ZeroMQ spout is created during runtime. It starts listening to a port by opening a pull socket on it. It then registers itself as a listener in a database held on the ESB master node. The registration information includes the IP address and port of the ZeroMQ socket, the name of the EPN and a timestamp. The name and the address information can be used by the selectors of the ESB to route events to the correct event processing networks. The timestamp is updated every 30 seconds and can be used for error handling in selectors.

The ZeroMQ spout receives events in XML format. Because only one thread can read from a ZeroMQ socket (ØMQ community 2011) and unmarshalling the XML messages computationally very costly, the unmarshalling is done in a separate bolt to prevent it blocking the receiving thread. This unmarshaller bolt can be then parallelised more aggressively. The unmarshaller converts the XML messages to a Java object representing the corresponding *SensorData*, *ForecastMessage* or *ComplexEvent* message. To improve performance in the internal message passing of Storm, the Java objects are serialised with Kryo, which is much faster than using the built-in Java serialisation or XML (Smith 2010).

JMS bolt

The CEP cluster sends its produced events back to the ESB via JMS. It first serialises the complex events to an XML format with JAXB and then puts the XML messages to a queue read by the ESB. The JMS sender bolt uses the ActiveMQ client libraries. In addition, because I also tested the performance oriented, experimental child project of

ActiveMQ, Apollo, an implementation bound to Apollo client libraries was created and tested.

Apollo offers a JMS API for Java applications. It differs from ActiveMQ in that it uses different wire protocols. ActiveMQ uses OpenWire version 2 by default (Apache Software Foundation 2011a), while ActiveMQ Apollo is built on STOMP 1.0 and 1.1 (Apollo community 2012). STOMP is a simple, text oriented for message oriented middleware. It has an easy wire format, which allows interoperability between different stakeholders. (FuseSource Inc. 2011) Because FuseSource, the developer of ActiveMQ products, offers a JMS wrapper library (Chirino 2011) for STOMP, it was very easy to use it as a replacement for ActiveMQ.

Creating complex events

The event creation in the EPL clauses of Esper is a bit too limited for our needs. Although one can define schemas for complex events in Esper (EsperTech Inc. 2012), there are no straightforward ways for creating events of our internal complex event type, *ComplexEvent*. The internal type is very useful, because it can readily be understood by our facilities offered by the ESB. *ComplexEvent* also has a well-defined XML representation, which can be used in external communications, too.

Currently, complex event creation is handled by separate bolts. An *EsperBolt* selects in its SELECT clause the fields needed as parameters of the event creation. These fields are packed in a map object and fed to complex event factory, which constructs a *ComplexEvent* object and emits it downstream as an event. This event can then be processed in CEP like any other event or be sent via a JMS bolt back to the ESB, or both. Similar bolts could also be used for the creation of new *SensorData* and *ForecastMessage* events, if needed.

4.2.6 Fault tolerance in Storm

The current implementation provides only partial fault tolerance. As explained before, Storm guarantees that all the events sent to it will be processed at least once. Also a message-wise at-most-once guarantee is available. What it doesn't guarantee is by which physical machine the events are processed. If there is a failure in some machine, it will be removed from the cluster and events will be routed to other participants in the cluster. Eventually a new EC2 node can be instantiated and attached to the cluster.

The reliability API of Storm is very easy to use. Only thing one must take care is anchoring the emitted events to a corresponding tuple tree and acking them, when they are processed further in the downstream. This is also implemented in EsperBolt. The API is further described in (Marz 2012b).

The problem with CEP on Storm is that we lose the current window of the previous messages. By default all the data used by the engine is held in memory and is lost, if the machine suddenly goes offline. (EsperTech Inc. 2012) If the used windows are small, this might not be a problem. When a new window is created to another machine, it begins filling when new events arrive, which is very similar in behaviour observed when the service is started. However, if we use long windows, which store days worth of history, it can take very long time until the EPA starts to function at its full potential. In any case, it is a case which must be taken into account when creating the EPLs and EPNs.

One solution to increase the fault tolerance is EsperHA (Esper High-Availability) add-on to Esper. EsperHA enables saving the state of the processing to any database, which has a JDBC driver (EsperTech Inc. 2008). Of the databases recommended by EsperTech especially Cassandra has promising scalability attributes when the total throughput is considered. In a recent study it achieved linear scalability in throughput when adding nodes. However, latency figures cast some doubt on how well it would really perform under heavy CEP load. (Rabl et al. 2012)

4.2.7 Partitioning example

In the following example I describe a quite simple set of four EPL queries, shown below in Listings 4.1 to 4.4. I try to argue that creating simple event processing agents forming powerful event processing networks is a natural and easy way designing complex event processing. The simple EPAs also exhibit beneficial features in the context of a complex event cluster, because they can be easily pipelined. Reducing the size of an EPA also allows running it in parallel, because the dependencies between the input events are simpler.

The purpose of the example EPN is to detect open doors via monitoring the temperature differences received from thermometers installed near the doors. The example EPN takes SimpleSensorData objects as an input. SimpleSensorData is a simplified version of the SensorData messages and defined for illustration purposes in this example. It has three fields: temperature, door and location. Temperature is a floating point valued temperature measurement denoted in Celsius. Door denotes the nearest door. Location tells whether the sensor is inside or outside the building.

The first EPL, represented in Listing 4.1, is a filter. It drops all sensor events coming from doors that are not listed as monitored in a database. Note that in the where clause we can call an arbitrary static Java method, and e.g. access a database. Esper caches the result of getMonitoredDoors(), because it takes no parameters. Filters like this are infinitely parallelisable and can be run on any number of nodes of Storm.

Listing 4.1: A filter EPA dropping non monitored doors

```
INSERT INTO MonitoredDoor
SELECT *
FROM SimpleSensorData
WHERE door IN Database.getMonitoredDoors()
```

The second EPL, shown in Listing 4.2, calculates a moving average for every source sensor identified by a door-location key. For simplicity, we may assume that there is only one sensor on both sides of every door. This EPA can also be easily parallelised. Storm provides a fieldsGrouping primitive, which ensures that all events with the same contents in certain fields are sent to the same tasks. The grouping is done by calculating a hash from the fields and is barely slower than the default routing option (Marz 2012d).

Listing 4.2: An aggregation EPA calculating a moving average

```
INSERT INTO Average(temperature, location, door)
SELECT AVG(temperature), location, door
FROM MonitoredDoor.win:time(1 min)
GROUP BY door, location
```

The third EPL (Listing 4.3) in effect makes a join of the Average and MonitoredDoor streams created in the previous EPAs. The join key used is again door-location, which can be supplied to fieldsGrouping to parallelise the EPA. Note that as a side effect of splitting the messages to different Esper engines, the windows on which the joins operate become drastically smaller. The EPA outputs the difference of the average and the latest measurement, if it exceeds the threshold of 2 °C.

Listing 4.3: A combined composition and filter EPA

```
INSERT INTO Diff(average, diff, door, location)
SELECT a.temperature, a.temperature - d.temperature AS diff,
        d.door, d.location
FROM Average.std:lastevent() a, MonitoredDoor.std:lastevent() d
WHERE a.location = d.location
        AND a.door = d.door
        AND Math.abs(a.temperature - d.temperature) > 2
```

The last EPA of the example matches the temperature differentials from both sides of the door. It then creates an alert with a message "Door open." The whole EPN is summarised in figure 4.2 to show the dependencies between EPAs. The figure is also equivalent to a topology, which could be running on Storm.

Listing 4.4: A composition EPA triggering an alert

```
INSERT INTO Alert(door, message)
SELECT inside.door, 'Door open' AS message
FROM Diff(location = 'inside').std:lastevent() inside,
        Diff(location = 'outside').std:lastevent() outside
WHERE inside.door = outside.door
```

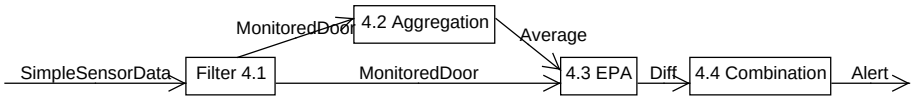


Figure 4.2: An EPN and a topology summarising the example

4.2.8 Web based CEP configuration

We designed a web based topology creator and configuration management system for the CEP service. The work for a prototype written in Grails was initiated, but because of resource allocation in the project, it was never finished. During early prototyping the GUI was not yet practical for development purposes, but the necessary topologies were hand coded. Anyway, I describe here the design we arrived at.

A web GUI would be used to define a topology by first creating the required spouts and then adding the required data types, bolts and streams. There would be a set of predefined spouts, for example reading a JMS queue, which could be instantiated and given the correct configuration (server URL, queue name, authentication credentials and other JMS configuration). The user would then create the bolts and connect the spouts to bolts (and bolts to the downstream bolts) by defining the flows of tuples between the nodes. The connection information would also carry the types of the data read from the streams. This topology definition is then saved to PostgreSQL.

Note that defining EPLs for the EPAs doesn't appear anywhere in the topology definition process. EPLs could be dynamically added and removed to and from a running topology. They would simply be added to a database, and the bolts running Esper would check for configuration updates periodically.

Before the topology definition could be deployed, it must pass a validation. The validation makes sure that all the EPAs really have the required inputs and that they produce the promised outputs. The inputs are clearly defined by the topology, and only thing we have to and can do is to check that the upstream spout or bolt has the required events defined. Because the outputs are dependent of the runtime behaviour of the EPLs, the outputs cannot be verified more strictly. Also, when EPLs are defined, they are validated by precompiling them before attempting to deploy them on the cluster.

For an end user, the usability of the web-based topology creation can be disputed. However, for our development team the attempt brought much needed experiences with bot Storm and Esper. Whatever the final form of the MMEA Bus will be, an end user interface for topologies, parts of topologies or single EPLs might very well be practical. For example, there must certainly be a means for changing threshold values in EPLs, and that would require an approach similar to ours.

Chapter 5

Results and evaluation

The main contribution of this thesis project was to create a scalable, distributed enterprise service bus with complex event processing capabilities and to evaluate its performance. In this chapter I describe the performance tests run on the system and their results. I also evaluate the system qualitatively with respect to the rest of the requirements described in section 5.8.

5.1 Performance test practicalities

5.1.1 The goals of performance tests

The combined ESB and CEP service architecture is designed to achieve a high throughput of events. In the tests I focus on measuring the throughput while varying the number of computing instances participating in the cluster. In addition, I give the approximate message processing latencies where applicable.

The *throughput* of the system is defined as the number of events that can be read from an input queue of the ESB in a unit of time while the system is under a full load. The *latency* is the time that it takes to read an event from an input queue of the ESB and to place the events derived from it back to the same queue. The latency is measured on a system, which is under a very minimal load. Because ZeroMQ and Storm can buffer substantial amounts of events before processing them, it is clear that we cannot give an upper bound to the latency, when the input rate is higher than the maximum throughput.

To get a complete picture on the bottlenecks, I measured the system from bottom up. I tested the system first in parts and then as a whole. The tested parts were ActiveMQ, ActiveMQ Apollo, ZeroMQ, WSO2 ESB, Storm, CEP cluster and the system including all these components. The results of previous stages were used to ensure the feasibility of next level tests.

5.1.2 Test environment

All the tests were run on Amazon EC2. I used large instances (m1.large), which provide four EC2 compute units and 7.5 GB of memory. Amazon doesn't give out any number on the internal network bandwith in EC2, but in my experiments it was not a relevant bottleneck. Only ZeroMQ was possibly affected by it, and that didn't matter to the system. The chosen instance type is by no means the most powerful type available on EC2. For example, "Cluster Compute Eight Extra Large Instance" (cc2.8xlarge) offers 60.5 GB of memory and 88 EC2 compute units. (Amazon Web Services LLC 2012b)

I chose m1.large instances, because my intention is to demonstrate the scale-out capabilities. With smaller and cheaper instance types I could deploy the system on a bigger number of instances (an on-demand m1.large costs $0.34/hour whereas cc2.8xlarge is $2.7/hour (Amazon Web Services LLC 2012a)). A less powerful cluster also makes performing the tests easier. Because the throughput of the cluster is not in millions of events per second, I could run the tests with only one load driver and after minor modifications with only one sink.

Using a cloud environment adds noise to the tests. In EC2 it is not possible to control the other users on the same physical hardware. Neither can an EC2 user control, whether a cluster of instances is located physically close to each others. Especially the speed of I/O, which happens often over the local network in an Amazon datacenter, can be variable.

The instances used as load drivers and sinks used Centos 5.5 like the ESB instances. For setting up Storm cluster I used storm-deploy utility (Marz 2012e) (described briefly in 4.2.1). Storm version was 0.8.1 and Esper version 4.6.0.

5.1.3 Effects of JVM and JIT compilation

Because most of the software to be measured is written in Java and run on the Java virtual machine, one must take possible effects of JIT compilation into account. Because JVM optimizes the execution of bytecode during runtime, the efficiency of certain parts of the program may vary. This means that programs that have been run longer tend to be faster. The phenomenon is sometimes called a warm-up effect. (Bull et al. 1999)

Also in my performance tests I observed some variation in the performance during the beginning of the tests. To eliminate the warm-up effect the tests were run several times before collecting the final results. In practise this meant that the software was fed at least million events and run for at least five minutes to give the JIT compiler enough data to optimise the execution.

5.1.4 Latency and clock skew

Because Amazon EC2 cloud doesn't provide a synchronized global clock, a clock skew could inadvertently affect the test results. To address this one could run NTP on all EC2 instances, which should make sure that the clock differentials stay below 100 ms. However, if the NTP daemon would make corrections to the system clock during a test run, the whole test run would be invalidated. (Windl et al. 2009)

I measured the time differences by sending messages from client to servers and then comparing timestamps carried by the messages. For most of the tests depending on timestamps issued by multiple EC2 instances, I ran the clock test with 1 000 messages. The clock difference was then estimated with the following formula:

$$skew = \frac{1}{1000} \sum_{n=1}^{1000} \frac{received_{ts} - sent_{ts}}{2} - remote_{ts}$$

where $sent_{ts}$, $received_{ts}$ and $remote_{ts}$ are the timestamps issued by the client when sending a reply request, by the client when receiving a reply and the one issued by the server and included in the reply, respectively. The measured skew ranged from -500 ms to 500 ms and it is taken into account in the test results. Anyway, because the skew can be so surprisingly big, the only way to get reliable results is to try to cope with only one clock.

5.1.5 Ensuring the quality of results

Before calculating any result numbers for the tests, the test data was validated. The messages received from the message queues was checked by ID to contain all of and no more than the messages originally sent. With CEP tests I carefully took into account the peculiarities of CEP benchmarking explained in 2.5, most importantly that there might be multiple correct results.

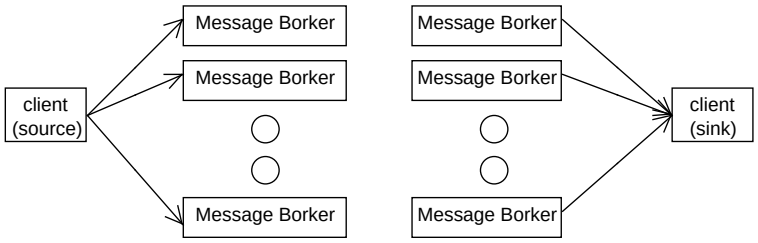
All the tests were run at least three times. If the results seemed stable, I accepted the middle number as the result. If the results seemed in any way sketchy, I repeated the tests. When I had enough measurements, I eliminated clear outliers and chose the median of the rest.

My criterion for "stable" was that all three were differed with maximum of 5 % of each others. I also analysed the sending and receiving rates graphically and rejected results at my discretion. If the graphs included unexplainable distortions or gaps, I made an attempt to find causes for them and rerun the tests. For instance, a longer than usual garbage collection run initiated by the JVM sometimes resulted in latencies of more than two seconds.

5.2 Message brokers

The message queues are the simplest components to test. For ActiveMQ and ActiveMQ Apollo servers I tested the input rate and the output rate separately. After some exploratory testing it became apparent that the broker (as opposed to a client acting as a driver or a sink) is the bottleneck for both input and output in both vanilla and Apollo implementations of ActiveMQ.

The test setups for input and output are shown in figures 5.1 (a) and (b). For input one EC2 instance was set up as a load driver (client), which would supply messages to one or more ActiveMQ servers, running on their own, dedicated machine instances. In initial testing it became apparent that one client could easily generate enough load for several servers. Same measurements also showed that the best performance would be achieved by having always two client threads per server.



(a) input (b) output

Figure 5.1: Test setup for ActiveMQ and ActiveMQ Apollo

I tested the message brokers with two sizes of messages, 40 byte long CSV strings and 470 byte long XML documents. Both included headers identifying the sender, an increasing ID number and a timestamp issued by the client right before sending the message. The message brokers were configured to add a timestamp to each message when received, and these timestamps were used to determine throughput and latency. While measuring the output rate, the client issued timestamps were used.

The messages were set not to be persisted in the broker. ActiveMQ was also configured to use asynchronous sending, which increased the throughput somewhat over the normal, synchronous method. Apollo did not offer a similar option for synchronisation. The brokers were run with -server and -Xmx2G JVM options.

The results for 40 byte messages are shown in table 5.1. I tested the brokers with combinations of one to four instances. When graphed, the results show a clear linear increase in throughput when more broker nodes are added.

Table 5.1: ActiveMQ and ActiveMQ Apollo performance with 40 byte messages

	input events/s		output events/s	
nodes	ActiveMQ	Apollo	ActiveMQ	Apollo
1	3 676	81 854	3 199	1 301
2	7 133	110 960	6 359	2 523
3	9 301	183 445	9 158	3 867
4	11 174	242 337	11 319	5 173

The tests with 470 byte messages were only run with one broker to get a picture on how the message size affects the performance. With ActiveMQ the difference was very small, and the input rate dropped to 3300 messages per second. With Apollo the impact was much bigger, and I could reliably send 20 000 messages per second. The difference in output rates in both cases was too small to be measured.

5.3 ZeroMQ

One key feature of ZeroMQ is that it is really lightweight and fast. From the beginning it was clear that ZeroMQ will not present any kind of bottleneck to the system. Hence I only validated my components that depend on it.

I ran a test using two EC2 instances, one sending messages and the other receiving them. The average throughput with 40 byte messages was at least 200 000 messages per second and with 470 byte messages about 120 000 messages per second. Latency was less than 1 ms, which was the finest granularity for the latency test. ZeroMQ was shown to be fast enough not to have impact to the overall results.

5.4 WSO2 Enterpirse Service Bus

I tested WSO2 ESB by running the ESB and ActiveMQ on the same machine instance. I first filled a JMS queue with a million messages and then opened a proxy service to read that queue and to record the rate of reading. The rate of reading was not measured rigorously, but it was about 2 000 messages per second with 470 byte messages. The scalability of the ESB is further discussed in the system tests in section 5.6.

The integration patterns and implementations on the ESB can be computationally very expensive. For example, doing an XSL transformation to all of the incoming messages can drastically slow down the throughput. Perera (2007) observed a three times decrease in transactions per second after introducing an XSLT mediator to a simple proxy service. Thus the most of the tuning of the ESB can be done in the adapters.

5.5 Complex event processing service

The performance of the complex event processing cluster is a very complicated issue compared to any other part of the system. We have already seen that the throughput of the message queues and the ESB can be increased just by adding more parallel instances. However, CEP can have very complex dependencies between the events. Inside the cluster there are also many distinct parts, which affect the performance in different ways.

5.5.1 Test setup

The test setup consisted of one load driver, one to ten Storm worker nodes and sink node running ActiveMQ Apollo. In Storm cluster there were also always one ZooKeeper and one Nimbus node for cluster management. The setup is illustrated in figure 5.2. In section 5.3 we saw that ZeroMQ can handle a very high number of messages per second. It also turns out that just one load driver is enough to keep the whole cluster busy. Being able to cope with one load driver makes testing and analysing the test results much easier, because all the sent messages can be issued a rising serial number and a reliable timestamp.

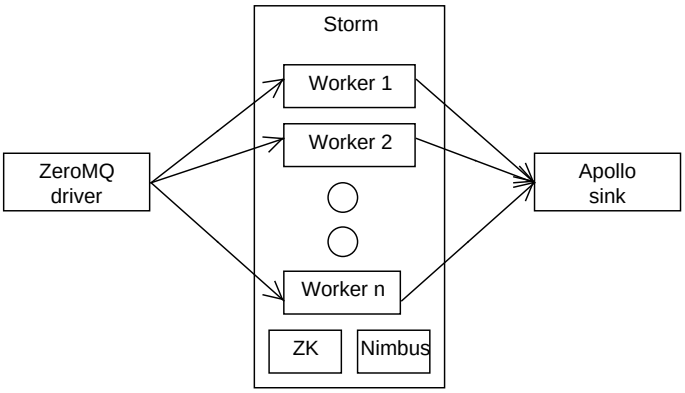


Figure 5.2: CEP cluster test setup

Although the overall performance of Apollo showed a bit more unreliable than that of the original ActiveMQ, I chose it as the sink because of its way superior input data rate. If we could use only one sink, it would give us similar benefits as with the single driver: messages would have unambiguous timestamps issued by a single receiving Apollo server.

Using only one sink naturally limits the maximum output rate. The maximum size of the messages produced in my CEP test cases is 470 bytes. In the previous section we saw that the maximum input rate of Apollo with this message size is 19 000 per second. However, because CEP systems, by their nature, often take in much more events than they produce, we are primarily interested in the input rate of CEP and can limit the output to a tolerable levels.

Storm allows setting parameters to control the number of workers on a topology and the number of tasks per bolt. Finding the best parameters took quite a bit time, but after exploring the effects, I ran the tests with the fastest parameters I found. As a rule of thumb, on AWS EC2 the best number of workers was six workers per m1.large instance. The task parallelism is a more complicated issue, but I'm confident that I managed to find reasonable settings.

5.5.2 Minimal topologies

First I tested the very simplest EPN possible. The only components of the EPN were a ZeroMQ spout for getting the test data in and a JMS sender bolt to get the results back to a queue. The topology would get input strings over ZeroMQ and then forward them to a JMS queue.

The input consisted of 470 byte XML documents. No processing on it was done in Storm, but the events carried the common headers for identification and timing measurements. With one processing node of Storm the measured throughput was 19 720 messages per second. This is clearly limited by Apollo. To confirm this I ran the same test with two nodes and got 19 897 as the result. The result is same and limited by Apollo. The average latency was 4 ms and 95 % arrived in under 35 ms.

The second test topology was almost as simple. The only thing that was added was an Esper bolt in between the spout and the sender bolt. The Esper bolt matched all the messages as they were and forwarded them to the JMS sender bolt. With one node the throughput was 8 791 messages per second and with two nodes 19 483. We clearly see that Esper engine slows down the system even without any meaningful function. The latency was around 4-5 ms.

5.5.3 Micro benchmarks

Mendes et al. (2008) presented a collection of micro benchmarks. They devised seven test cases for benchmarking the fundamental functionalities and also some nonfunctional features common to most of the CEP engines. The tests include filtering, aggregation, joins, pattern matching, large windows, handling bursts and multiple simultaneous queries. I chose the first three of them to be run on the CEP cluster. In addition I experimented with an identity EPL, which matches all the events and forwards them.

The test topology included a ZeroMQ spout for input, a separate unmarshaller bolt, an Esper bolt with the given EPL and a JMS sender bolt. The XML documents are bound to Java objects in a separate bolt, because parsing XML is very time consuming. With this arrangement the thread reading the ZeroMQ socket is always dedicated to transferring data and the unmarshaller can be parallelised separately.

I ran the tests with six different cluster sizes. The throughput results are given in the table 5.2. The first thing to note from the results is that the throughput is at about five times less than in the minimal topology. This is due to the parsing of XML documents when the events arrive to the cluster. According to the statistics produced by the Storm control panel, unmarshalling one message took between 1-3 ms.

In the figure 5.3 we see that identity and selection scale linearly. This is expected, because these cases don't have any data dependencies. The reason why selection is faster than identity is that it drops 90 % of the events before sending them back to a queue while the identity case must send all the events back.

Aggregation I used calculates an average of a field over a moving window. In the system there is no simple way of distributing this computation and the aggregating Esper engine is limited to a single machine instance. The same is true with join, which calculates a Cartesian product over two 1 000 event windows. In both cases the throughput increases first fast when the new resources can be used to receiving XML messages. When the speed-up from parallel unmarshalling is exhausted, running the EPLs becomes the bottleneck.

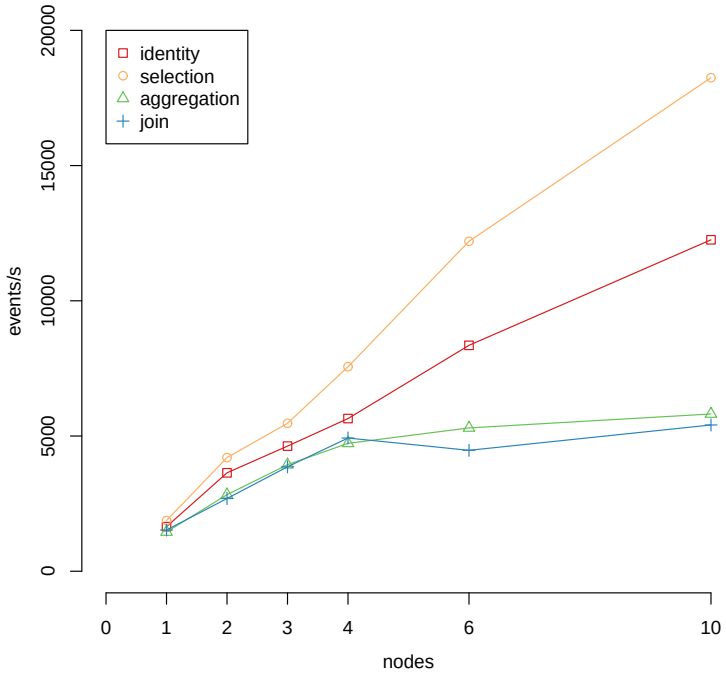


Figure 5.3: Four Esper micro benchmarks on Storm

The measured latencies for the micro benchmarks are shown in figure 5.4. These results are from a cluster with four worker nodes. The latencies were quite stable: In identity, selection and join test cases 80 % of the were processed arrived in less than 10 ms and 95 % in less than 20 ms. Aggregation, which did not scale so in our naïve case processed 75 % under 10 ms and 90 % under 20 ms.

Table 5.2: Four micro benchmarks on Storm with one to ten m1.large EC2 instances

	events/s			
nodes	identity	selection	aggregation	join
1	1 647	1 873	1 452	1 524
2	3 639	4 204	2 831	2 689
3	4 627	5 470	3 939	3 862
4	5 645	7 561	4 729	4 926
6	8 353	12 200	5 300	4 471
10	12 255	18 250	5 815	5 408

5.5.4 Performance of the partitioning example

In section 4.2.7 I presented a simple topology, which could be used to detect open doors by comparing temperature differentials over time and matching anomalous readings. I implemented the presented EPLs as a Storm topology and tested, how many temperature readings the CEP cluster can handle per second. The aim of this test is to see how well the cluster performs under a more real life complex event detection workload.

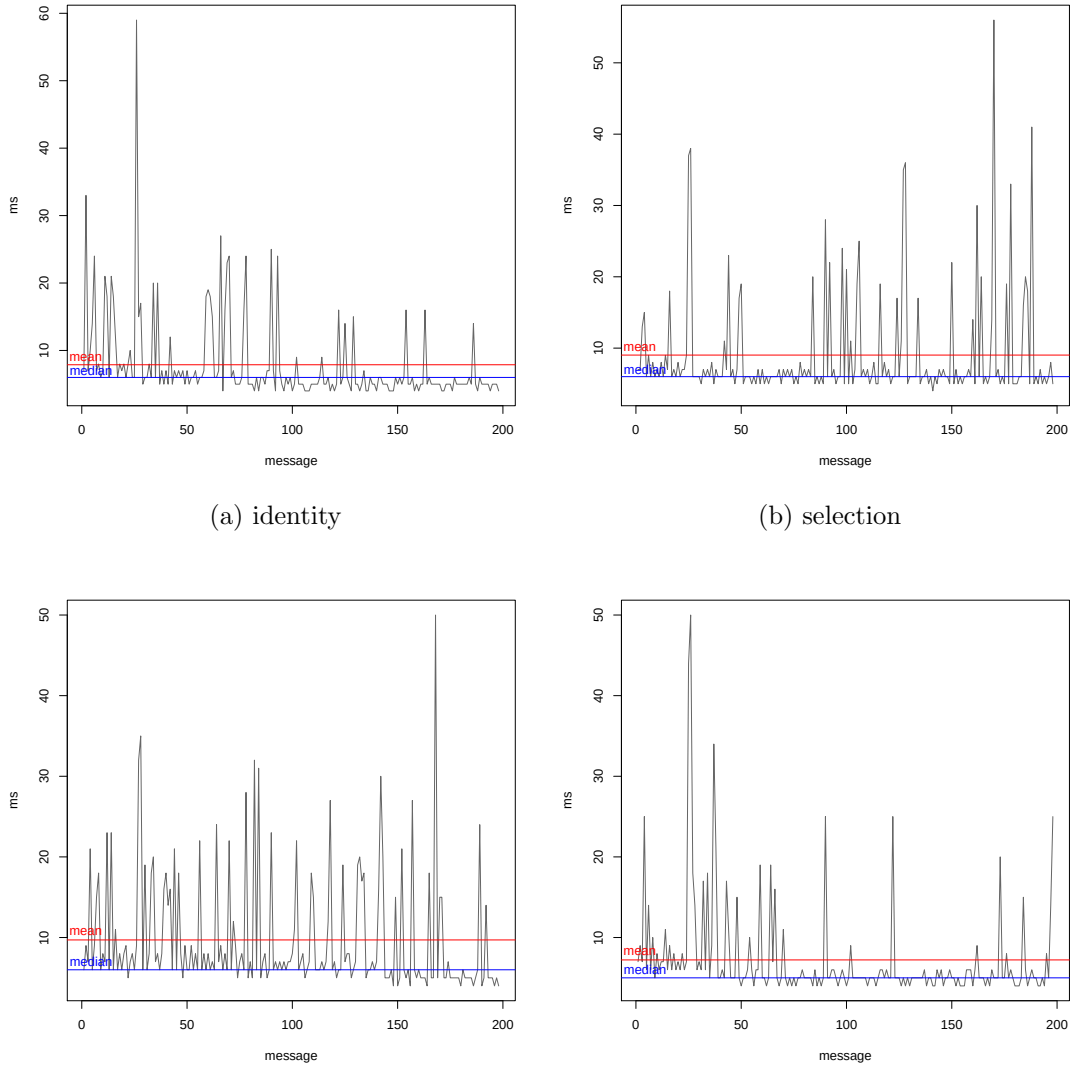
Because XML unmarshalling and binding to Java objects was observed to take so much time, I used a much simpler data format for this test. The temperature readings were supplied as 20 byte CSV strings. The strings contained three values: temperature, door ID and indication whether it represents a reading of an internal or an outside thermometer.

The partitioning example tests the CEP cluster in a very different way than the micro benchmarks. The computational requirements for data type transformations and the data transmitted over the network per event are greatly reduced. This means that more resources are used for pattern matching and other CEP functions.

The practical example shows that a distributed EPN running on Storm fits the problem class well. The results of the throughput tests are shown in table 5.3 and plotted in figure 5.5. Because most of the operations can be run completely in parallel, the throughput increases fast when more processing nodes are added to the CEP cluster.

Table 5.3: Tests run with the example topology described in section 4.2.7

nodes	events/s
1	8 282
2	10 084
4	13 962
6	20 687
8	28 206



(a) identity

(b) selection

(c) aggregation

(d) join

Figure 5.4: Measured latencies from the micro benchmarks with red line (upper) showing mean and blue line (lower) showing median

5.6 System performance

For the final validation of the system I tested it as a whole. I set up four m1.large EC2 instances running WSO2 ESB and ActiveMQ. I chose ActiveMQ to be used with the ESB, because the output performance of Apollo was inadequate and it crashed too often. For CEP I built a simple topology running the selection test selecting 10 % of the processed events.

I used one EC2 instance to drive load to the ActiveMQ queues on the ESB instances. The ESBs read the messages from the queues and forwarded them to the CEP service

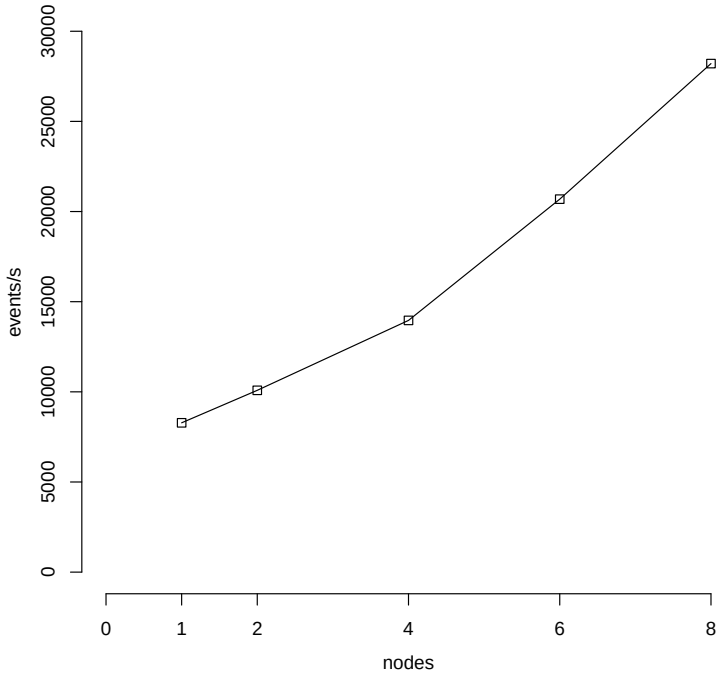


Figure 5.5: Throughput of the partitioning example

over ZeroMQ. CEP service filtered out 90 % messages based on the message ID and then sent the rest to a JMS queue acting as a sink. The events preserved in the sink queue were retrieved and their timestamps analysed to get the total throughput. The results are shown in table 5.4. The system is shown to scale linearly when more ESB nodes are added, if the CEP service is not overloaded.

No filtering was used in the ESB, but all the messages were subject to forwarding to the ESB. Filtering will probably slow down the ESB a lot, because it would require parsing the XML documents. However, the ESB already must manipulate the SOAP elements of the message, and the slowdown might not be as huge as in the CEP cluster. Furthermore, we observed in section 5.5.4 in the case of the partitioning example that we can save a lot of processing time by using a more concise event representation.

Table 5.4: System tests with a four node CEP cluster and one to four ESB instances

nodes	events/s
1	1 757
2	3 417
3	5 264
4	7 017

I tested the latency of the system when it was idle. I sent single messages with the size

of 470 bytes to the input queue. The average time it took for them to return to the same queue was 9 ms and most of the events were processed and transmitted in under 35 ms.

When the system is overloaded, the latencies approach infinity. The message queues (JMS and ZeroMQ) and Storm can build huge buffers of events. During the tests I observed Storm taking almost half an hour to empty an input buffer, after the load driver sending the messages had been shut down.

5.7 Discussion of performance test results

I now summarise the results and discuss the performance of the components of MMEA Bus. Although the system is functionally divided in two separate parts, an ESB and a CEP cluster, the machines running ESB provide much of the infrastructure that make CEP easier. For instance, the CEP outputs its complex events to a JMS queue running on an ESB node. Therefore I proceeded in tests from the individual ESB components towards the CEP cluster and then the whole system.

The message queues used in this project, ActiveMQ, ActiveMQ Apollo and ZeroMQ can be trivially scaled just by adding multiple parallel instances. With the message size of 470 bytes, one ActiveMQ node was measured to handle about 3 000 messages per second on our hardware. The performance of Apollo was much more volatile, and while an Apollo broker could receive 80 000 messages per second, those messages could not be sent forward with a comparable rate. The output rate was only 1 300 messages per second. The much more simplistic ZeroMQ could send about 120 000 of the same messages per second.

The performance of CEP is a very complex issue. We saw that when computation is simple, Storm can process a magnitude more of messages. Storm creator Nathan Marz has on multiple occasions announced on *storm-user* mailing list performance results of more than 100 000 messages per second per worker node. Most recently, on 29.6.2012, he claimed that he sees throughputs of over a million messages per second per node with Storm version 0.8.0. However, the type of hardware was not specified, but I assume it is comparable to the largest machines available from Amazon Web Services, Cluster Compute Eight Extra Large Instance (*cc2.8xlarge*).

In my tests the throughput of CEP cluster was limited to maximum of 1 600 messages per second per node when using the chosen 470 kB messages. My best explanation for the puny performance is the slow XML unmarshalling, which is performed for every message received at the CEP cluster. The statistics of Storm reveal that unmarshalling one message takes 1-3 ms. If the execution is heavily pipelined, one CPU could handle 1 000 unmarshallations per second. Because the *m1.large* instance type used in these tests has two CPUs, the result of 1 600 events/s is very close to this estimate.

The explanation that the system is heavily constrained by the XML unmarshalling performance fits also the experiments with the minimal topologies. When Esper engine was introduced to a dummy topology (section 5.5.2), the throughput dropped from something more than 20 000 events/s to 8 000 events/s (my setup did not allow measuring the peak performance of Storm). Esper supports XML documents as one of its input object formats (EsperTech Inc. 2012), but it is not clear, what kind of processing Esper did to it in our case. Nevertheless, even if Esper engine did not build comparable Java objects, because the simple EPL did not require it, we clearly see that the format radically slowed down the system.

In the case of the other two micro benchmarks, which were not parallelisable, we saw that the performance tops at about 6 000 events/s. This is due to the change of a bottleneck from the XML unmarshalling to the CPU-limited EPA running aggregation or join. If we considered using the fastest instances offered by AWS, cc2.8xlarge with 88 EC2 compute units (m1.large has four), we could expect seeing 20 times speed-up in ideal case, which totals 120 000 events/s. However, it is unlikely that Esper could make use of all the available CPU cores and hyperthreads. Anyway, this kind of performance would certainly be adequate to fulfil any requirements of MMEA Bus. This speed-up should be experimentally verified, but it requires a much more scalable test setup.

One option to increase the performance would be to serialise the events already in the ESB. The ESB is required in any case to peek into the headers and possibly contents of the XML-based messages. By converting the XML documents to e.g. Kryo serialised messages, we could reduce duplicate processing on CEP. This would especially beneficial, if the ESB forwards the same events to multiple EPNs.

I also considered some other possible bottlenecks. On *storm-user* mailing list, there are many different sources of misconfiguration slowdown. For instance, the number of executors on each worker nodes must be sufficiently high. Also, garbage collection of the JVM can add delays. On faulty bolts the messages might not be acked correctly. Some of these issues were already discussed in section 5.1.3. Based on my tests run during the development, can quite reliably say that these are only minor issues at most.

5.8 Qualitative evaluation of MMEA Bus

In addition to the performance tests, I evaluated the other nonfunctional attributes of MMEA Bus qualitatively. In this section I go through the rest of the scalability, high availability and configuration requirements, which could not be measured in the previous tests. Finally, I compare the implementation with the eight requirements posed by Stonebraker et al. (2005).

5.8.1 Comparison to the requirements

The ESB can be scaled up *elastically*. I tested adding new EC2 instances to the ESB cluster, and the instances registered to the load balancer, fetched the required configuration and started acting as slave nodes of the ESB as expected. I could also scale down the cluster by removing nodes, but this sometimes resulted to missing messages, if the clients were not configured to resend messages or a polling task running on the killed instance was not shut down cleanly. The AWS EC2 environment provides tools for *autoscaling*. Scaling out could be enabled with a correct configuration, but it was not tested.

Storm cluster does not support moving running tasks from a node to node without pausing the execution. Additionally killing moving the context of an Esper engine from worker to worker is not possible.

The elastic load balancing feature of AWS is responsible of balancing the load on the ESB. In the CEP cluster Storm should distribute the work equally on every node. However, these features were not tested.

The high availability issues were not addressed in this thesis. The ZeroMQ spout I wrote does not support resending messages, which is a requirement for the Storm fault-tolerance API. However, Storm still handled removing arbitrary nodes quite cleanly.

The current version of MMEA Bus provides a solid base for building configuration management features. There is no user interface for the configuration but some plans were discussed in sections 3.4.3 and 4.2.8. Also the stream and subscription management is not yet complete. The ESB provides many opportunities for defining access controls and securing connections.

5.8.2 The eight rules revisited

The eight rules for stream processing (summarised in section 3.2) give clear guidelines for CEP. The first rule is actualised completely in MMEA Bus: the data is not stored during processing. The data is only persisted to a database after the analysis is complete.

To fulfil the second rule, MMEA Bus supports Esper EPL language. EPLs can be used to define event processing agents. The third and fourth rules, stream imperfections and ensuring predictable outputs, are not enforced on the platform level. The application developers who write EPLs and define EPNs must take into account the implications of a distributed system.

Although not discussed in this thesis, the platform has a preliminary support for storing data into a cloud storage service. To prevent the conflict with the first requirement, the data is stored only after all the processing is completed and all the directly derived events

are created. However, as of yet, this stored data cannot be mixed with stream data in CEP, which was required by the fifth rule.

The sixth rule says that the data must be highly available and safe. These concerns were left out of the scope of my research. Also the architecture doesn't have any facilities to help with the partitioning of the workload, as suggested by the seventh rule. Nevertheless, the CEP cluster provides a highly scalable platform which suits event processing networks well.

The last rule says that the CEP engine should be highly optimised. The components used in the CEP cluster, Storm and Esper, are both known for their performance. However, the benchmarks run and presented previously in this chapter leave some doubt on whether the implementation is really optimal. For instance, using XML and parsing it showed very expensive.

Chapter 6

Conclusions

Complex event processing is an emerging technology which operates on event streams and historical data. It can be used to detect patterns consisting of multiple events. CEP offers huge performance improvements over traditional database management systems when applied to real time data.

Enterprise service bus is an integration product. It can be used to connect multiple endpoints in a heterogeneous environment. The enterprise software uses often event-driven service oriented architecture. Many academics say that as an integration product acting as a centralised mediator for the business events, an ESB is a natural host for complex event processing.

This thesis focused in the scalability issues of CEP as a part of an ESB. The main issue is that the scalability models required by CEP and an ESB are completely different. ESB can often be completely stateless, because it usually operates only on a single message at a time. However, in complex event processing the data dependencies between events can be very complicated. To explore this issue I developed a prototype for complex event processing enabled enterprise service bus, called MMEA Bus.

The architecture of MMEA Bus tries to answer this mismatch by deploying CEP as a separate service outside the ESB. I described a dedicated CEP cluster built on Storm real time stream processing framework. The cluster performed CEP with multiple Esper CEP engines, which were run on different machines with their own contexts. The communication between the Esper engines is done over the network by passing complex events created by the engines.

The two parts of MMEA Bus can be scaled out separately by adding more processing nodes in a cloud computing environment. The performance tests I performed show that one ESB instance can mediate 1 750 messages of 470 bytes scaled linearly by adding more instances. The throughput of the CEP cluster depends a lot on the computational requirements of

pattern detection. Nevertheless, in a simple real life example case the throughput was 28 000 events per second on a cluster with eight worker nodes. The latency of the system was very low, usually less than 10 ms.

The implemented platform fits its integration purposes well, because the ESB product offers a wide variety of different adapters and mediation patterns. The platform allows building authentication services and ensuring the security. The system also offers a solid base for configuration management.

Further research and development

There are several rough corners in the current prototype implementation. The biggest problem with the current implementation is the lack of fault-tolerance and high availability features. Although Storm provides primitives for guaranteeing message processing, MMEA Bus doesn't currently implement the required interfaces.

Storm requires a reliable input queue to be able to replay messages that go missing during the processing on the cluster. This would need a separate, distributed message broker to act as a reliable message store. In my implementation the communication between the ESB and the CEP cluster was done over ZeroMQ, which doesn't have a separate broker. Thus messages can be lost, if the other endpoint of a communication fails.

Replaying the missing events is not enough for complex event processing, because the CEP engines rely heavily on their current state. A highly tuned engine can hold in its context windows thousands or even millions of previous events. If this window is lost because of some failure, the following produced results tend to be incorrect. The provider of Esper offers a proprietary add-on, EsperHA, which persists the current state in a database and allows recovering it after a failure.

There are also some more minor issues, which would be nice to have addressed in the system. The CEP cluster doesn't support any kind of elastic scaling without stopping the processing for a while. In configuration management of CEP there are many things that could make the development much more user friendly. One thing is making tools to check and test the topologies before deployment. The topology definitions would allow many kinds of static analysis, which are currently done only during runtime.

One interesting opportunity for future research lies in the the performance tests run on the CEP cluster. I focused only in a limited number of CEP functionality. The performance tests described by Mendes et al. (2009) have yet five more micro benchmarks that were not run yet on my implementation.

Bibliography

Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., hyon Hwang, J., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the Borealis stream processing engine. In *In CIDR*, pages 277–289.

Adi, A., Botzer, D., Nechushtai, G., and Sharon, G. (2006). Complex event processing for financial services. In *Services Computing Workshops, 2006. SCW '06. IEEE*, pages 7–12.

Ahmad, Y., Berg, B., Cetintemel, U., Humphrey, M., Hwang, J.-H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., and Zdonik, S. (2005). Distributed operation in the borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 882–884, New York, NY, USA. ACM.

Amazon Web Services LLC (2012a). Amazon EC2 pricing. `http://aws.amazon.com/ec2/pricing/`.

Amazon Web Services LLC (2012b). Amazon elastic compute cloud - user guide. `http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/using-network-security.html`.

Amazon Web Services LLC (2012c). Elastic load balancing. `http://aws.amazon.com/elasticloadbalancing/`.

Apache Software Foundation (2011a). Apache ActiveMQ - OpenWire. `http://activemq.apache.org/openwire.html`.

Apache Software Foundation (2011b). Apache Tribes - the Tomcat cluster communication module. `http://tomcat.apache.org/tomcat-6.0-doc/tribes/introduction.html`.

Apache Software Foundation (2011c). ZooKeeper: Because coordinating distributed systems is a zoo. `http://zookeeper.apache.org/doc/r3.3.3/index.html`.

Apache Software Foundation (2012). Axis2 clustering support. `http://axis.apache.org/axis2/java/core/docs/clustering-guide.html`.

Apollo community (2012). Apollo. `http://activemq.apache.org/apollo/`.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). STREAM: The Stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 665–665, New York, NY, USA. ACM.

Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment.

Azeez, A. (2008). Auto-scaling web services on Amazon EC2. `http://people.apache.org/~azeez/autoscaling-web-services-azeez.pdf`.

Baarah, A., Mouttham, A., and Peyton, L. (2011). Improving cardiac patient flow based on complex event processing. In *Applied Electrical Engineering and Computing Technologies (AEECT), 2011 IEEE Jordan Conference on*, pages 1 –6.

Babcock, B., Datar, M., and Motwani, R. (2004). Load shedding for aggregation queries over data streams. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 350–, Washington, DC, USA. IEEE Computer Society.

Babu, S. and Widom, J. (2001). Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120.

Babu, S. and Widom, J. (2004). StreaMon: an adaptive engine for stream query processing. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 931–932, New York, NY, USA. ACM.

Bo, D., Kun, D., and Xiaoyi, Z. (2008). A high performance enterprise service bus platform for complex event processing. In *Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, GCC '08, pages 577–582, Washington, DC, USA. IEEE Computer Society.

Borealis team (2006). Borealis application programmer's guide. `http://www.cs.brown.edu/research/borealis/public/publications/borealis_application_guide.pdf`.

Box, D. C. F. e. a. (2004). Web services addressing (WS-addressing). `http://www.w3.org/Submission/ws-addressing/`.

Bull, J. M., Smith, L. A., Westhead, M. D., Henty, D. S., and Davey, R. A. (1999). A benchmark suite for high performance Java. In *Proceedings of ACM 1999 Java Grande Conference*, pages 81–88. ACM Press.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., and Zdonik, S. (2003). Scalable distributed stream processing. In *In CIDR*.

Chirino, H. (2011). The JMS interface to STOMP. `https://github.com/fusesource/stompjms`.

Coulouris, G., Dollimore, J., and Kindberg, T. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 5th edition.

Cugola, G. and Margara, A. (2012). Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 72(2):205–218.

Dong, L., Wang, D., and Sheng, H. (2006). Design of RFID middleware based on complex event processing. In *Cybernetics and Intelligent Systems, 2006 IEEE Conference on*, pages 1 –6.

Dudziak, T. (2012). Storm-Esper. `https://github.com/tomdz/storm-esper`.

Eckert, M. and Bry, F. (2009). Complex event processing (CEP). *Informatik-Spektrum*, 32:163–167. 10.1007/s00287-009-0329-6.

Eckert, M., Bry, F., Brodt, S., Poppe, O., and Hausmann, S. (2011). A CEP babelfish: Languages for complex event processing and querying surveyed. In Helmer, S., Poulovassilis, A., and Xhafa, F., editors, *Reasoning in Event-Based Distributed Systems*, volume 347 of *Studies in Computational Intelligence*, pages 47–70. Springer Berlin / Heidelberg.

EPCglobal Board (2008). Class 1 generation 2 UHF air interface protocol standard "gen 2". `http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2`.

EPCglobal Board (2009). Application level events (ALE) standard. `http://www.gs1.org/gsmp/kc/epcglobal/ale`.

EsperTech Inc. (2007). Esper performance. `http://docs.codehaus.org/display/ESPER/Esper+performance`.

EsperTech Inc. (2008). EsperHA: High-availability for event processing. `http://www.espertech.com/products/esperha.php`.

EsperTech Inc. (2011). Understand EsperTech licensing. `http://www.espertech.com/download/public/EsperTech%20licensing%20v5.pdf`.

EsperTech Inc. (2012). Esper reference. `http://esper.codehaus.org/esper-4.6.0/doc/reference/en-US/html_single/index.html`.

Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Fernando, S. (2010). Sharing registry space across multiple product instances. `http://wso2.org/library/tutorials/2010/04/sharing-registry-space-across-multiple-product-instances`.

FuseSource Inc. (2011). STOMP protocol specification, version 1.1. `http://stomp.github.com/stomp-specification-1.1.html`.

Gelernter, D. (1989). Multiple tuple spaces in Linda. In *PARLE '89 Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer Berlin / Heidelberg.

Ghalsasi, S. Y. (2009). Critical success factors for event driven service oriented architecture. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 1441–1446, New York, NY, USA. ACM.

Godage, U. (2007). Writing a mediator in WSO2 ESB - Part I. `http://wso2.org/library/2898`.

Gudgin, M., Hadley, M., Mendelsohn, N., Lafon, Y., Moreau, J.-J., Karmarkar, A., and Nielsen, H. F. (2007). SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

Gurgen, L., Labbe, C., Olive, V., and Roncancio, C. (2005). A scalable architecture for heterogeneous sensor management. In *Database and Expert Systems Applications, 2005. Proceedings. Sixteenth International Workshop on*, pages 1108 – 1112.

Haas, Hugo; Brown, A. (2004). Web services glossary. `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`.

Heinze, T. (2011). Elastic complex event processing. In *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, pages 4:1–4:6, New York, NY, USA. ACM.

IBM (2004). *Patterns: implementing an SOA using an enterprise service bus*. IBM Corp., Riverton, NJ, USA.

Kellner, I. and Fiege, L. (2009). Viewpoints in complex event processing: industrial experience report. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 9:1–9:8, New York, NY, USA. ACM.

Kleiminger, W., Kalyvianaki, E., and Pietzuch, P. (2011). Balancing load in stream processing with the cloud. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 16–21, Washington, DC, USA. IEEE Computer Society.

Kotovirta, V. (2012). Ympäristömittaustiedosta uusiin palveluihin. *Automaatioväylä*, 2:30–21.

Ku, T., Zhu, Y., and Hu, K. (2008). A novel complex event mining network for monitoring RFID-enable application. In *Computational Intelligence and Industrial Application, 2008. PACIIA '08. Pacific-Asia Workshop on*, volume 2, pages 925 –929.

Lakshmanan, G. T., Rabinovich, Y. G., and Etzion, O. (2009). A stratified approach for supporting high throughput event processing applications. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 5:1–5:12, New York, NY, USA. ACM.

Leavitt, N. (2009). Complex-event processing poised for growth. *Computer*, 42(4):17–20.

Luckham, D. and Schulte, R. (2008). Event processing glossary - version 1.1. *Processing*, 1.1(July):1–19.

Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Luckham, D. C. and Frasca, B. (1998). Complex event processing in distributed systems. Technical report, Program Analysis and Verification Group Computer Systems Lab, Stanford University.

Magid, Y., Sharon, G., Arcushin, S., Ben-Harrush, I., and Rabinovich, E. (2010). Industry experience with the IBM Active Middleware Technology (AMiT) complex event processing engine. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 140–149, New York, NY, USA. ACM.

Maréchaux, J.-L. (2006). Combining service-oriented architecture and event-driven architecture using an enterprise service bus. Technical report, IBM.

Marz, N. (2012a). About Storm - Free and open source. `http://storm-project.net/about/free-and-open-source.html`.

Marz, N. (2012b). Storm - Guaranteeing message processing. `https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing`.

Marz, N. (2012c). Storm - Rationale. `https://github.com/nathanmarz/storm/wiki/Rationale`.

Marz, N. (2012d). Storm - Tutorial. `https://github.com/nathanmarz/storm/wiki/Tutorial`.

Marz, N. (2012e). storm-deploy - Home. `https://github.com/nathanmarz/storm-deploy/wiki`.

Mendes, M. R. N., Bizarro, P., and Marques, P. (2008). A framework for performance evaluation of complex event processing systems. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 313–316, New York, NY, USA. ACM.

Mendes, M. R. N., Bizarro, P., and Marques, P. (2009). A performance study of event processing systems. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 221–236. Springer Berlin Heidelberg.

Menge, F. (2007). Enterprise service bus. *Free and open source software conference*.

Michael, M. M., Moreira, J. E., Shiloach, D., and Wisniewski, R. W. (2007). Scale-up x scale-out: A case study using Nutch/Lucene. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8. IEEE.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. (2003). Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 245–256, Asilomar, California.

ØMQ community (2011). ØMQ manual - zmq_socket. `http://api.zeromq.org/2-2:zmq-socket`.

Mukherjee, A., Diwan, P., Bhattacharjee, P., Mukherjee, D., and Misra, P. (2010). Capital market surveillance using stream processing. In *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, pages 577–582.

OASIS (2006). OASIS web services security (WSS) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.

Owens, T. J. (2007). Survey of event processing. Air Force Research Lab, In-house technical memo.

Paschke, A., Kozlenkov, A., and Boley, H. (2010). A homogeneous reaction rule language for complex event processing. *CoRR*, abs/1008.0823.

Perera, A. (2007). WSO2 ESB performance testing round 1. `http://wso2.org/library/1721`.

Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.-A., and Mankovskii, S. (2012). Solving big data challenges for enterprise application performance management. *Proc. VLDB Endow.*, 5(12):1724–1735.

Ruh, William A.; Maginnis, F. X. B. W. J. (2001). *Enterprise Application Integration - A Wiley Tech Brief*. John Wiley & Sons.

Sbz, S. Z., Zdonik, S., Stonebraker, M., Cherniack, M., Etintemel, U. C., Balazinska, M., and Balakrishnan, H. (2003). The Aurora and Medusa projects. *IEEE Data Engineering Bulletin*, 26.

Schmidt, M.-T., Hutchison, B., Lambros, P., and Phippen, R. (2005). The enterprise service bus: making service-oriented architecture real. *IBM Syst. J.*, 44(4):781–797.

Sellis, T. K. (1988). Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52.

Sharon, G. and Etzion, O. (2007). Event processing network – a conceptual model. In *Proceedings of VLDB, Second International Workshop on Event Driven Architecture and Event Processing Systems, 2007*.

Sharon, G. and Etzion, O. (2008). Event-processing network model and implementation. *IBM Syst. J.*, 47(2):321–334.

Smith, E. (2010). Comparing varius aspects of serialization libraries on the JVM platform. `http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking`.

Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47.

Wächter, J., Babeyko, A., Fleischer, J., Häner, R., Hammitzsch, M., Kloth, A., and Lendholt, M. (2012). Development of tsunami early warning systems and future challenges. *Natural Hazards and Earth System Science*, 12(6):1923–1935.

Wheeler, J. (2011). Mediation - separating business logic from messaging. `http://www.mulesoft.org/documentation/display/MULE3CONCEPTS/Mediation+-+Separating+Business+Logic+from+Messaging`.

Windl, U., Dalton, D., and Martinec, M. (2009). The NTP FAQ and howto. http://www.ntp.org/ntpfaq/NTP-s-algo.htm.

Wishnie, G. and Saiedian, H. (2009). A complex event routing infrastructure for distributed systems. In *Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference - Volume 02*, COMPSAC '09, pages 92–95, Washington, DC, USA. IEEE Computer Society.

WSO2 (2011). WSO2 governance registry - distribution. `http://wso2.org/project/registry/4.0.0/docs/index.html`.

WSO2 (2012). About WSO2. `http://wso2.com/about/`.

Xing, Y., Zdonik, S., and Hwang, J.-H. (2005). Dynamic load distribution in the Borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 791–802, Washington, DC, USA. IEEE Computer Society.

Zang, C., Fan, Y., and Liu, R. (2008). Architecture, implementation and application of complex event processing in enterprise information systems based on RFID. *Information Systems Frontiers*, 10(5):543–553.