Event Processing Applied to Streams of TV Channel Zaps and Sensor Middleware with Virtualization

Pål Evensen

April 4, 2013

Abstract

The last decade has seen an exponential increase in mobile computing devices, as well as an increasing adoption of sensor technology in process industry, homes and public spaces. The increasing amount of information made available by such devices has led to a class of pervasive systems that require little or no user input. Smart home systems is an example of such pervasive systems. A main obstacle for application developers dealing with sensor-based systems is *heterogeneity* of devices and protocols. A common obstacle for end-users is the manual configuration of networked devices.

Our first research contribution is a middleware that overcomes these obstacles: The SENSEWRAP middleware addresses the problem of heterogeneity in a smart home setting through the *virtualization* of hardware and services. Furthermore, it provides automatic network configuration and service discovery.

The usefulness of pervasive systems usually correlates with their ability to perform their functions in the background, without user involvement. Instead, these systems base their actions on available information relevant to their application, e.g., they are *information-driven*.

For information-driven systems, like smart-home systems and other pervasive systems to be able to decide on the correct action at the right time, it is vital that the correct information is made available to them in a timely manner. A primary asset of publish/subscribe interactions is the immediate distribution of new information available to interested parties, and as such, it is a well-suited model for building highly scalable and flexible systems that are able to cope with a dynamic environment.

Complex event processing is a fairly new paradigm that refers to the pro-

cessing and correlation of events as they occur. There exists several specialized programming languages for performing complex event processing. A main goal of such languages is to enable the programmer to express patterns of events in a simpler and more straightforward manner than what is possible with a general-purpose programming language.

A main contribution of this thesis is an exploration of the tradeoffs involved in using a specialized, declarative event processing language versus using a general-purpose, imperative programming language for event processing applications. Our results indicate that going the specialized language route does indeed simplify development of event processing applications, but that this comes at the expense of performance.

Furthermore, we present the EVENTCASTER platform for building event-based systems, on which we have built two novel event processing applications: The viewer statistics and ADSCORER applications are research contributions in their own right. The viewer statistics application demonstrates how event processing techniques can be applied to broadcast television, in order to provide more accurate viewer statistics than what is currently available, in near-real time. With the ADSCORER application, advertisers and broadcasters are provided with a detailed evaluation of each individual advertisement, previously only available to advertisements distributed on the web.

Acknowledgements

First and foremost, I would like to thank my advisor, Associate Professor Hein Meling at the University of Stavanger, who co-authored of all the publications included in this thesis. His high standards and work ethic has been an inspiration throughout this whole process. I am grateful for his valuable guidance, and for the considerable time and effort he has spent on helping me – not only during the PhD programme, but also after his obligations towards me as a PhD student had expired.

A large part of the work presented in this thesis was completed after I returned to my job at Altibox, after the three-year period of the PhD programme at the university was over. In addition to Hein, I owe the completion of my work to a handful of people at Altibox who made it possible for me to finish what I started out to do. Without the encouragement, involvement and generosity from my managers at Altibox, I would not have been able to see this project to its completion.

In particular, I would like to express my sincere gratitude towards Dagfinn Wåge, Ronny Lorentzen and Omar Langset, who believed in the project, and helped me steer it in a direction that was beneficial for both the company and my research.

I would also like to thank my good colleague Per Fjeld, whom I collaborated with on the commercial side of the viewer statistics project at Altibox. His considerable experience and knowledge of the media industry has been a valuable asset in much of the work presented here.

Thanks to Professor Roman Vitenberg, Associate Professor Alberto Montresor and Professor Reggie Davidraju for being in my dissertation committee.

Finally, I would like to thank my parents and my brother Øyvind for their relentless support, and my friends for giving me some much needed diversion from work. Special thanks goes to my good friend Sveinung Hevrøy for the countless dinners he has served me during these years.

Published Parts of this Thesis

[1] Pål Evensen and Hein Meling. Adscorer: an event-based system for near real-time impact analysis of television advertisements (industry article). In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 85–94, New York, NY, USA, 2012. ACM.

[2] Pål Evensen and Hein Meling. A paradigm comparison for collecting tv channel statistics from high-volume channel zap events. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*, DEBS '11, pages 317–326, New York, NY, USA, 2011. ACM.

[3] Pål Evensen and Hein Meling. Sensewrap: A service oriented middleware with sensor virtualization and self-configuration. In *ISSNIP 2009: Fifth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 261–266, Piscataway, N.J., December 2009. IEEE.

[4] Pål Evensen and Hein Meling. Sensor virtualization with self-configuration and flexible interactions. In *Casemans '09: Proceedings of the 3rd ACM International Workshop on Context-Awareness for Self-Managing Systems*, pages 31–38, New York, NY, USA, 2009. ACM.

Contents

Acronyms

AMQP Advanced Message Queuing Protocol. 48, 49

API Application Programming Interface. 42, 43, 45, 47–49, 74, 76, 85, 102, 103, 111, 119

ARP Address Resolution Protocol. 65

CEL Cayuga Event Language. 154, 205, 206, 208

CEP Complex Event Processing. 55, 58, 59, 62, 97, 111, 158, 159, 185, 187

CORBA Common Object Request Broker Architecture. 43, 44

CRUD Create, Read, Update and Delete. 42

DDS Distributed Data Service. 49, 55, 187

DHCP Dynamic Host Configuration Protocol. 65

DNS Domain Name System. 65, 67, 69

DNSSD DNS Service Discovery. 65, 67–69, 82

DSLAM Digital Subscriber Line Access Multiplexer. 118

EAR Enterprise Archive. 98, 99, 103

ECA Event-Condition-Action. 57

EDA Event-Driven Architecture. 26, 29, 50–52, 55, 60, 61, 70, 71, 95, 96

EPA Event Processing Agent. 51, 71, 95–97, 99, 100, 107

EPG Electronic Program Guide. 116, 133

EPL Event Processing Language. 58, 59, 97–100, 104–108, 112, 127, 128, 132–134, 136–141, 143, 149, 150, 152–156, 159, 169–173, 205, 206

EPN Event Processing Network. 51, 52, 71, 96, 100

ESP Event Stream Processing. 59

FIFO First In First Out. 33

FTP File Transfer Protocol. 46

GPRS General Packet Radio Service. 75, 93

HDMI High-Definition Multimedia Interface. 119, 123, 160, 165, 166, 169

HTML HyperText Markup Language. 18, 19, 33

HTTP HyperText Transfer Protocol. 32, 33, 42, 43, 68, 69, 74, 83

IAR Initial Audience Retained. 120, 121, 163

IEEE Institute of Electrical and Electronics Engineers. 73, 85

IETF Internet Engineering Taskforce. 49, 64, 67, 69

IGMP Internet Group Management Protocol. 118

IP Internet Protocol. 63–65, 68, 69, 74, 116, 129, 156, 187

IPTV Internet Protocol Television. 113, 115, 116, 118, 119, 129

J2EE Java 2 Enterprise Edition. 96, 102, 110

J2ME Java 2 Micro Edition. 68, 84

JMS Java Messaging Service. 32, 37, 47–50, 56, 102, 103, 111, 187

JMX Java Management Extensions. 106

JVM Java Virtual Machine. 68, 88, 151

MOM Message-Oriented Middleware. 29, 33, 38, 45–48, 50, 51, 70, 99, 102, 103, 187

NFS Network File System. 43

OSGi Open Services Gateway initiative. 91

P2P Peer-to-Peer. 49, 92

PVR Personal Video Recorder. 116, 128

QoS Quality of Service. 48–50, 119, 187

REST Representational State Transfer. 42, 43

RF Radio Frequency. 63, 74

RFC Request For Comments. 64

RMI Remote Method Invocation. 43, 44, 68, 83

RPC Remote Procedure Call. 29, 38–46, 70

SOAP Simple Object Access Protocol. 43, 44, 68, 69, 83, 92, 119

SPOT Small Programmable Object Technology. 77, 82, 84–87, 92

SQL Structured Query Language. 25, 37, 91, 97, 104, 205

SSDP Simple Service Discovery Protocol. 69

STB Set-Top-Box. 26, 54, 62, 104, 113, 115–119, 121–123, 126–133, 136, 137, 141, 144, 149, 156, 158, 159, 161, 165–167, 169, 170, 173, 175, 180, 184, 187

STOMP Simple Text Oriented Messaging Protocol. 48, 49, 103, 111

TCP Transport Control Protocol. 63, 64, 67, 68, 74, 77, 80, 83, 85, 86, 98, 111

UDP User Datagram Protocol. 63, 68, 74, 77, 83, 98, 129, 145, 149, 150

UPnP Universal Plug and Play. 68, 69

URI Uniform Resource Identifier. 42

URL Uniform Resource Locator. 107

USB Universal Serial Bus. 75, 85

VoD Video-on-Demand. 116, 128

VoIP Voice-over-IP. 116

WSDL Web Service Definition Language. 44

XML eXtensible Markup Language. 37, 49, 68, 69, 96

XMPP Extensible Messaging and Presence Protocol. 49, 50

Chapter 1

Introduction

Networked devices with computing capabilities can no longer be assumed to be stationary. The trend towards information-driven systems, coupled with an exponential increase in mobile communicating devices has led to a new class of distributed applications with ever-increasing demands for timeliness, scalability and dynamism. Coping with these demands is unattainable using the traditional request/reply interaction model of the client/server paradigm. This new class of applications include ubiquitous and mobile computing systems such as smart home systems and context-dependent car navigation systems. In the business world, automated stock trading, automated inventory management and real-time business intelligence applications are additional examples of this new breed of applications.

A key requirement for many systems like these is that they should be more or less *autonomous*, meaning that they should demand little or no human intervention. A smart home system exemplifies this, as the value it brings to its inhabitants to a great extent depends on its ability to remain "invisible" to the end users. A prerequisite for creating an autonomic environment is for the system to have knowledge of the context and activity of other resources within the infrastructure [120, Ch. 1.3].

In order for systems to operate independently of user interaction, they must be able to make the correct decisions at the right time by themselves. Because these systems primarily base their actions on information, they are said to be

information-driven [114, Ch. 1.1], as opposed to driven by user interaction or time. Thus, for information-driven systems to be effective, accurate information must be readily available in a timely manner. Depending on the context, information is often worthless to the class of applications discussed here if not delivered immediately, like in the case of automated trading, where a delay of as little as a fraction of a second may amount to a missed opportunity.

As a contrast to information-driven systems, consider a batch system, processing batches of information overnight: Here, time is the initiator of action, not the information itself, thus it is *time-driven*.

Historically, middleware has been centered around a request/reply interaction model, and was designed in a world where networked computing devices were stationary, and for the most part had predictable behaviour. This model is an excellent and time-proven match for user-request handling and time-driven batch systems. Yet many attempts have been, and are still being made to amend this model to fit scenarios that it was not designed for. Even though it is possible to hammer a nail with a crowbar, using a tool for something that it was not designed to do is rarely a good solution.

A concrete example of the hammer/crowbar analogy is the display of data feeds on web pages, using polling techniques. Currently, one can observe this by visiting any of the major Norwegian news sites, such as `dagbladet.no`, which updates their news feed by having the clients reload the entire front page at fixed intervals instead of sending updates to the clients when news stories are published.

The web was originally a collection of static HTML pages, connected with hyperlinks, and in this context, the request/reply interaction model makes perfect sense: A client establish a connection to a server, requests a document, and receives it in response from a server, whereupon the connection is closed. However, when applied to continuously updated *streams* of content, the request/reply interaction pattern requires the client to repeatedly poll the server for new information, resulting in unnecessary setups and teardowns of connections.

Due to their massive user base, web standards move slowly, and technologies for enabling push to the browser, such as WebSockets [88] (included in

the upcoming HTML5 [84] specification) are not yet part of any official web standard. It is currently only supported in a few select browsers, leaving web developers with little choice of interaction models for pages with dynamic content. Workarounds such as Comet [104] and Ajax [33] use long-polling [87] techniques to reduce the number of unnecessary requests, but these only serve as temporary band aids, even introducing some new problems themselves [88], to a problem whose root cause is use of the wrong interaction model.

Timeliness is often the first thing that suffers when applying the client-/server model to large-scale information-driven systems. A consequence of the request/reply interaction model that follows the client/server paradigm is that the consumers of information need to *poll* for it, instead of having information *pushed* to them when it is available.

In a poll interaction, the client asks the server "do you have information for me?" whereupon the server either returns new information, or nothing at all. In either case, the server responds immediately, whereupon the connection is closed, leaving the client free to go on about its business.

The potential latency of the information flow is thus affected by the polling interval, and increases with the number of system components the information has to traverse, each introducing a new delay. Figure 1.1 illustrates how latency accumulates in poll-based systems where components are indirectly addressed. In a system where the information has to traverse two servers in order to reach the client, and given a polling interval of 30 seconds for each component along the path, with each server caching the last read value, the potential latency is 1.5 minutes with two intermediaries between the client and the information source.

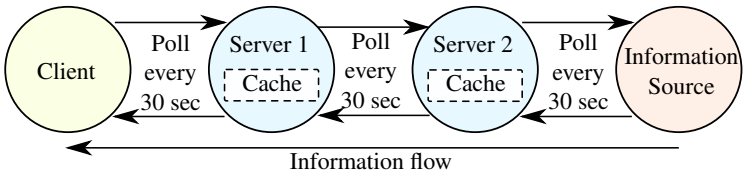

Figure 1.1: Indirect poll

The resulting latency observable from the information consumer, in such multi-tiered client/server systems can be expressed as $\sum_{i=1}^{n} a_i$, where a_i represents

the latency of the i^{th} component in a chain of interdependent components. As can be deducted from this, latency can be reduced by decreasing the polling interval of information consumers. However, this wastes processing and network resources, causing a tradeoff between latency and resource waste.

If we consider a push interaction model for the same scenario, the latency from polling intervals is eliminated, as the servers forwards the information "immediately" upon reception (Figure 1.2). In this model, the servers are part of an *event notification service*, an abstraction layer on top of the underlying components, responsible for delivering events to subscribers. Some added latency for each component is still unavoidable due to network and processing delays, but this is equally true for both scenarios, and comes in addition to the poll-induced latency.

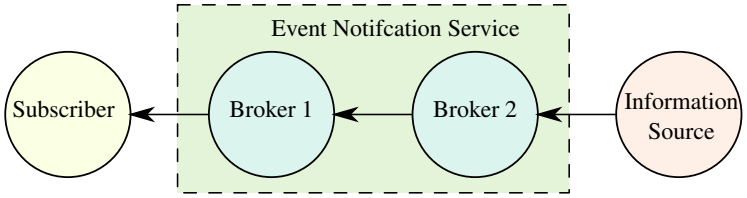

Figure 1.2: Indirect push

While the traditional models for building distributed applications are indeed the optimal solution for a number of well understood applications, this thesis presents some scenarios where the client/server model falls short, and discusses why event-based interactions is the appropriate approach for these.

A fairly recent area of research, event processing originates from the research communities of publish/subscribe and its predecessor; group communication, as well as the active database community. In publish/subscribe systems, subscribers generally express their interest in events sent to a topic, or events of a certain type, while active databases allows users to specify Event-Condition-Action rules in the form of triggers. Common for these paradigms is that they operate on single events in isolation, allowing only limited expressiveness [41].

Complex event processing expands upon these paradigms by introducing *context* as a subscription criteria, enabling consumers to express their interest in

events in much greater detail. The Merrian-Webster dictionary defines context as: *"The interrelated conditions in which something exists or occurs."* In this thesis, context refers other events that occur within the same system, or any dynamic or static property that may be of interest to the application, such as time, geographical location, etc. For the application of detecting credit card fraud, for instance, event processing technologies allows for the expression of complex patterns, involving events from multiple sources and taking causality and context into account.

The banking industry is an example of an industry that would benefit from the continuous processing capabilites of event-based systems. Here, transactions are typically processed overnight in 24-hour batches [144], and one could argue that the speed at which business could be conducted on a global level would greatly benefit from an event-driven approach. The slow adoption of event-based technologies within the banking industry can probably be attributed to the high cost of replacing legacy systems responsible for core business functions.

Sensor/actuator networks is another prime example of distributed event-driven systems, where a reduction in manufacturing cost, size and power consumption has led to an increasing adoption of wireless sensor technology in process industry, homes and public spaces. Areas of deployment include environmental monitoring [56, 74], military and security applications such as target tracking and intrusion detection [8], as well as automated supermarkets, where the groceries in your cart are automatically charged to your credit card as you leave the shop, made possible using radio-frequency identification technology [123]. In the field of healthcare, sensor technology enables medical personnel to monitor patients from within the patient's own homes, making the treatment more convenient and comfortable, while at the same time freeing capacity at the hospital [130].

Some advantages of wireless sensors are cost of deployment and flexibility in placement. However, the mobility of wireless sensors usually means that they are battery dependent, and have limited resources, presenting application developers with some new (and some not so new) challenges that will be discussed

later in this dissertation.

The examples mentioned above are just some of the possible areas for deployment of sensor technology. For the sake of narrowing down the scope and to provide a reference point, we will focus mainly on sensor technology in the context of smart homes. In the field of event processing, the focus is on complex event processing and the detection of patterns within time-ordered event streams. As such, related areas of research, such as streaming databases and rule-based processing are covered briefly in Chapter 2.

The rest of this introductory chapter provides a description of the project context, an overview of the research challenges considered in this dissertation, followed by a summary of contributions, and finally, an outline of the thesis.

1.1 Project Context

Some of the work presented in this dissertation was developed as part of the Integrated IP-based Services for smart Home environments (IS-Home) project [128], a larger effort aimed at offering an autonomic communication middleware platform to simplify development and deployment of integrated and context-aware services in a smart home environment. The IS-Home project was a joint effort between industry actors represented by service provider Altibox AS, medical equipment vendor Lærdal Medical AS and telecommunications hardware vendor Telsey (Italy), and academia, represented by the University of Stavanger.

Near the end of the IS-Home project, which lasted from 2007 to 2010, we observed several application needs for which event processing technologies was a natural solution within the Altibox organization. These applications include real-time television viewer statistics, telephony fraud detection, and the distribution of soccer match results in real-time. As such, the focus shifted from sensor middleware to creating an industry-ready platform that would facilitate general event processing.

Working closely with the Norwegian service provider Altibox gave us the opportunity to address real-world challenges. This partnership also provided us with valuable insight concerning industry demands for robustness, scalability

and maintainability, and most interestingly; access to actual user-generated data.

However, doing research in cooperation with a commercial company has not been without challenges, and during the project there has been some stumbling blocks along the way: Changes in the organizational structure, the reliance on external vendors for developing project-critical code, time consuming organizational bureaucracy, the problem of getting priority for a research project in a production-oriented environment as well as conflicting commercial and academic interests are all obstacles that have been dealt with throughout the project period. These are obstacles that are likely to face any long-term academic research project performed in cooperation with a commercial organization, and would be wisely considered in advance by anyone planning to undertake a similar task. Fortunately, my immediate supervisors throughout this period have all recognized the value of the research, and allowed me the necessary wiggle room to complete the project.

Given the industrial nature of the project, a main goal has been to create implementations that are usable in an industrial setting. For this reason, state-of-the-art technological platforms with some industry momentum already established has been favored over research prototypes.

1.2 Research Challenges

Five main research challenges in the context of sensor networks and event-based systems are addressed in this dissertation:

HETEROGENEITY: *Overcoming heterogeneity in communication and application protocols for sensor devices*. A major obstacle to the adaption of sensor technology is the sheer variety of communication and application protocols used by sensor devices. Complicating matters more, many of these are proprietary. A middleware that hides the difference between sensor protocols would greatly benefit the development of, ease the adoption of, and improve the flexibility of applications interacting with sensors.

INTERACTION STYLES: *Supporting both pull and push interactions in sensor middleware*. Applications interacting with networks of sensors and actuators

will likely have two distinct styles of interaction: Applications controlling actuators like switches and locks will typically interact with these in a request/reply (pull) manner, while event-driven applications, reacting to the output of sensors and state of actuators need a publish/subscribe (push) interface. The flexibility of the middleware mentioned in the previous paragraph will be further enhanced if the middleware support both *request/reply* and *publish/subscribe*-style interactions.

SERVICE DISCOVERY: *Finding a scalable and convenient way of handling service discovery in smart home sensor networks.* Finding a scalable, distributed way of keeping track of resources within sensor networks is key to a successful middleware platform. Since embedded devices have limited resources in terms of processing, bandwidth and power, it is essential to use a mechanism that is lightweight and resource-efficient. Avoiding unnecessary polling and traffic within the network conserves bandwidth and prolongs the life of sensor batteries — something that becomes increasingly important as the sensor network expands in number of nodes. An inefficient service discovery algorithm could easily exhaust the available bandwidth for polling and control messages.

EVENT PROCESSING ARCHITECTURE: *Providing a general architecture for efficient processing of high volumes of events.* The processing requirements of events from sensor networks and other producers of data ranges from simple, stateless queries, operating on a single value, to complex, stateful event processing across multiple data streams. A common requirement for event processing in these types of systems is that the processing be performed in near real-time. This becomes a serious challenge if the number of input events and subscribers are high and the amount of state to be handled are significant. Furthermore, the complexity of the system performing the event processing can make it difficult to maintain.

TRADEOFFS: *Evaluate the tradeoffs between a declarative versus an imperative programming model for event processing.* A challenge in event processing is the cognitive load of implementing and administer applications where the amount of state to be maintained is large. Furthermore, it is often difficult to keep the performance at an acceptable level in these types of applications. In

current research, a popular approach to event processing is to use SQL-derived declarative query languages to represent continuous queries [42, 60, 7, 28, 21, 67, 147, 17], operating on streams of events. It is a challenge to understand the performance and complexity impact of the query-based approach versus the general-purpose language approach.

1.3 Summary of Contributions

The research contributions presented in this dissertation are twofold: in the area of middleware for sensors, we present some novel ideas of how heterogeneity in hardware and application protocols can be hidden from application developers in the form of hardware *and* service virtualization, addressing the challenge of HETEROGENEITY. A middleware implementation, SENSEWRAP has been developed as proof of concept. This middleware support both request/reply and publish/subscribe interactions, addressing the challenge of INTERACTION STYLES. Furthermore, SENSEWRAP also enable sensor services to be seamlessly located and accessed, using the standardized ZeroConf [23, 25] suite of protocols, addressing the challenge of SERVICE DISCOVERY.

The second area of contribution is related to the dissemination and processing of events, such as those produced by sensors and humans interacting with appliances, to name a few. These contributions include:

1. *A paradigm comparison for stateful event processing*. Through the implementation of concrete use cases, we evaluate the tradeoffs in performance and complexity between using a specialized, declarative event processing language and using a general-purpose imperative programming language for performing Complex Event Processing. Our findings indicate that there is a significant performance tradeoff in favor of the general-purpose programming language. However, we conjecture that, given adequate performance, using a specialized event processing language is still a better solution for building more advanced Complex Event Processing applications, due to the simplicity gained by this approach. This contribution addresses the EVENT PROCESSING ARCHITECTURE challenge.

2. *An industry-proven architecture for general, stateful event processing*. Addressing TRADEOFFS, the EVENTCASTER implementation demonstrates how Message-Oriented Middleware for the distribution of events, coupled with a complex event processing engine can be used to make an extensible event processing system, capable of handling stateful and complex event patterns while maintaining adequate performance for a wide variety of use cases.

3. *Novel applications for real-time television statistics and advertisement scoring*. Providing a new way of scoring televison advertisements that is more in line with current measurement methods for online media, the ADSCORER application not only demonstrates the capabilites and usefulness of the EVENTCASTER middleware — it is a contribution to the field of media measurement in its own right. The same applies for the application generating viewer statistics in near real-time, developed for the paradigm comparison part of the dissertation. This last contribution adds further support to the EVENT PROCESSING ARCHITECTURE challenge.

1.4 Impact

Currently, a single EVENTCASTER instance generates viewer statistics for over 320,000 Set-Top-Boxes (STBs) in near real-time, handling over 38 000 events per minute during peak hours. Running on modest hardware, and deployed in a production environment, the EVENTCASTER architecture serves as proof that an Event-Driven Architecture (EDA) is indeed an optimal solution for services such as this, and that the EVENTCASTER platform is fully capable of handling the demands of an industrial deployment.

In the time after the publication of the DEBS papers [51, 52], the rate of announcements of collaborations between providers of STB data such as Rentrak and smaller television networks have only increased [109, 108, 64]. This adds weight to our observations of an ongoing paradigm shift in the media measurement industry, and illustrates the timeliness of capitalizing on measurements obtained from STBs to provide more accurate viewer statistics.

The combination of *hardware virtualization* and *service discovery* introduced in the SENSEWRAP middleware has been introduced in a number of middleware for mobile services, such as Serval [117] and Hydra [61], after the SENSEWRAP papers was published. Although we cannot take credit for influencing the design of this middleware, it speaks to the relevance of the work presented in Chapter 3.

1.5 Outline of Thesis

- Chapter 2 gives a general overview of middleware, service discovery protocols and event-driven architectures. These are key technologies and paradigms that this dissertation builds upon.
- Chapter 3 addresses the challenges HETEROGENEITY, INTERACTION STYLES and SERVICE DISCOVERY, and presents and evaluates the SENSEWRAP middleware for sensors.
- Chapter 4 introduces the EVENTCASTER middleware, which addresses the EVENT PROCESSING ARCHITECTURE challenge, and is the underlying platform for the applications presented in Chapters 6 and 7.
- Chapter 5 provides some background on media measurement, establishing the context for the following chapters.
- Chapter 6 addresses the TRADEOFFS challenge by evaluating the suitability for event processing of two distinct programming paradigms through the implementation of a real-time television statistics application.
- Chapter 7 presents an application for scoring televised advertisements in near real-time, built on the EVENTCASTER middleware and the viewer statistics application introduced in Chapter 6.
- Chapter 8 concludes this thesis. This chapter provides conclusions along with a summary and directions for future work.

Chapter 2

Middleware: Abstractions, Interactions and Paradigms

Middleware is the core issue of this thesis, and this chapter provides an overview of concepts and technologies relevant to the research challenges presented earlier. The aim is not to provide a complete taxonomy of the paradigms discussed, but rather to provide some background on middleware in general, and to give an overview of the most relevant technologies.

The first part of the chapter covers a brief walk-through of the purpose, history and applications of middleware. Section 2.3 and 2.4 gives an overview of the primary interaction models, and discusses the Remote Procedure Call (RPC) and Message-Oriented Middleware (MOM) abstractions in detail.

Following the middleware discussion, Section 2.5 provides an introduction to Event-Driven Systems, and covers the main processing models found in these kinds of architectures. Since the proposed solution to the TRADEOFFS challenge comes in the form of an EDA, some background of the different event processing models, as well as an overview of current research is necessary.

Section 2.6 discusses the challenges facing developers of applications for sensor networks and smart homes, providing some necessary background information for the challenges HETEROGENEITY, INTERACTION STYLES and SERVICE DISCOVERY, while at the same time relating these challenges to the concepts and models presented in Section 2.5. Furthermore, the section contains an overview

of Service Discovery Protocols, and a discussion on how these can be used in a smart-home context. Concluding the section is a more detailed presentation of the ZeroConf [25] protocol suite that is used in the SENSEWRAP sensor middleware implementation, presented in Chapter 3. A summary concludes the chapter.

2.1 The Motivation Behind Middleware

With the introduction of networked computing, the need for a layer to facilitate communication between heterogeneous systems emerged. The main attributes of middleware is to hide differences (transparency) between systems and data formats (abstraction) in order to provide a homogeneous view of the world to applications. Middleware reside between distributed applications and the operating system [121] (Figure 2.1).

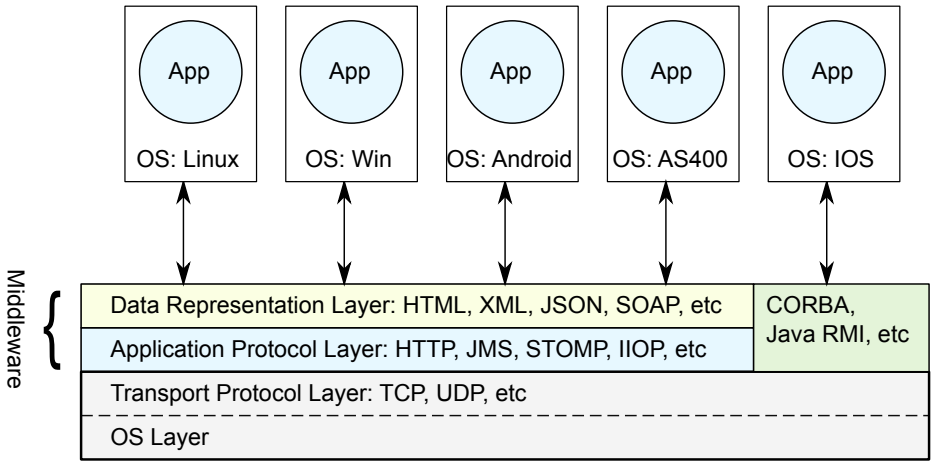

Figure 2.1: Middleware

In early distributed computing systems, application developers typically had to relate to connections at a very concrete level for handling communication between systems, involving the design of low-level protocols that all involved systems had to follow. However, the design of custom, low-level protocols is error-prone [35], and does not make for reusable software components.

Ideally, application developers should only have to worry about problems

within the application domain, where they can add the most value. By providing higher-level interfaces between systems, middleware makes life easier for developers, masking much of the complexity of networks and protocols [14]. One can say that middleware grease the proverbial wheels between the components of distributed applications.

2.2 The Client/Server Model

The client/server-model is the most commonly found design in distributed applications, and is a time-proven architecture where its status as the foundation of the web speak to its successfulness. In client/server interactions, the server offers a set of services that clients make use of. The server listens for requests from clients, and returns a response to each request.

The guiding principle behind the client/server model is a *separation of concerns*, typically handing over the responsibility for presenting the user interface to the client, while assigning the application logic to the server. Ideally, this frees up resources at the server, making the application more scalable. Furthermore, the separation of functionality should make it easier to make changes to the application, as there will be no need for upgrading the clients when making optimizations to the application logic, provided there are no changes to the interface [54].

Even though the client/server model usually implies a request/reply interaction model (discussed in the following section), it is not always necessarily so; it may also refer to any distributed software architecture with clearly separated client and server-components, where the “heavy lifting“ is assigned to the server component. Case in point, Java Messaging Service (JMS) (discussed in Section 2.4.2) is referred to as a client/server-oriented middleware, even though its interaction model is message-based. This is because the specification assigns the responsibility of disseminating messages to a central server component.

2.3 Interaction Models

In this section, we cover the basic interaction models of middleware in the context of distributing information from different sources.

One distinction is whether the information is pulled by the consumer from the producer, or pushed by the producer to the consumer. In other words, in a *push*-style interaction, the interaction is initiated by the producer of information, while in a *pull*-style interaction, the consumer initiates the interaction with a request for information.

Another distinction is whether the interaction is *synchronous* or *asynchronous*. In a synchronous pull-style interaction, the consumer blocks until it has received a response from the producer. In an asynchronous pull-style interaction, the initiator is free to perform other tasks while the producer generates a response.

The following subsections discusses the nature of three commonly found interaction models in middleware, as well as their application areas.

2.3.1 Request/Reply

One of the oldest and most widespread pattern of interaction between networked computers, request/reply (Figure 2.2) is normally found in client/server-oriented middleware, where the client issues a request, and the server responds with a reply.

Request/reply is a pull-style interaction that is most commonly used in a synchronous style, although asynchronous implementations also exist. HyperText Transfer Protocol (HTTP) is a prime example of a middleware protocol that is based on the request/reply interaction pattern. A typical example of a request/reply interaction is when a client, in the form of a web browser, issues a HTTP GET command to a web server, and receives a HTTP response message with HTML content.

2.3.2 Message Queueing

The message queuing model [48] introduces a message queue between information producers and consumers, where producers append messages to the end

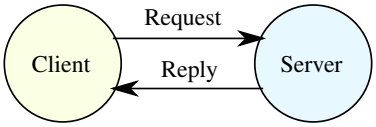

Figure 2.2: The request/reply interaction model

of a FIFO queue, which is pulled from the queue by a consumer, when available (Figure 2.3). The queue may have several producers and consumers, but each message is only consumed once, by a single consumer. A guarantee of delivery in the form of message persistence is integral to this design. When a message has been delivered, the consumer acknowledges that it has received the message, and it is subsequently removed from the queue, not to be delivered to another consumer. This also provides an easy way of setting up effective load balancing, as it enables several consumers to collaborate on the processing of a single queue.

On the consumer-side, both push- and pull-style interactions may be realized with this model, as the consumers may choose to pull messages from the message queue at any time, on demand, or have them delivered immediately, subscription-style. In cases where there are many subscribers to a single message queue, the MOM usually allows for a number of strategies for distributing the messages among the subscribers, such as round-robin or random distribution.

An ordering system is an application where this model would be applicable: only-once delivery prevents orders from being double processed, while persistence ensures that no orders are dropped.

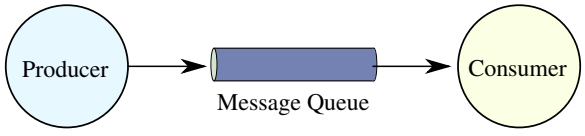

Figure 2.3: The message queueing interaction model

2.3.3 Publish/Subscribe

In publish/subscribe middleware [48, 121, 30], subscribers express their interest in specific types of events through subscriptions to an *event notification service*, that is responsible for matching incoming events to the subscriptions that has been registered.

Publish/subscribe is an asynchronous push-style interaction model, where publishers anonymously send messages to an unknown number of subscribers, without knowing anything about the subscribers or target applications. This effectively decouples the publishers and subscribers from each other, and is referred to in the literature as *The Principle of Decoupling* [119]

Four actions make up the interface to a publish/subscribe middleware: *publish*, *notify*, *subscribe* and *unsubscribe*. The event notification service notifies subscribers of events that matches their subscription, and occurs between the subscribe and unsubscribe action. If a subscriber is not available at the same time an event of interest is published, it will be held by the event notification service, and forwarded as soon as the subscriber is available. Furthermore, if a publisher does not have an active network connection available at the time of an event occurence, it will hold on to the event, and publish it as soon as a working connection is established, decoupling the time dimension between the systems.

Each message may be delivered multiple times, as every subscriber receives the message. There may be several subscribers, but each subscriber only receives each message once. Messages published prior to a subscription is not considered, as subscribers are only able to express their interest in *future* events.

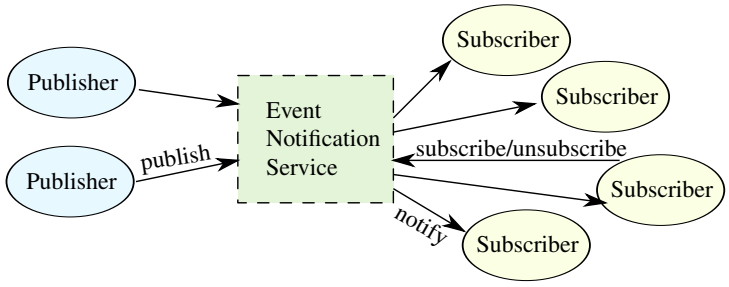

Figure 2.4: The publish/subscribe interaction model

Its decoupling of the time, space, and synchronization dimensions [48] makes publish/subscribe an ideal model for applications where broadcasting of data is involved, such as news or stock ticker services. These are cases where producers and consumers may be dispersedly located (space) and have limited connectivity (may not be connected at all times, and not necessarily at the same time). Broadcasting-type services may also have a very high number of clients. Not having to block while receiving or sending messages (synchronization decoupling), is essential for the scalability of such services.

Figure 2.4 illustrates the components in a publish/subscribe system. The borders around the event notification service is dashed to illustrate that it is not necessarily a single entity with firm boundaries, but may be distributed, depending on the implementation.

A downside to the publish/subscribe model, the way it has been implemented traditionally, is the reliance on a central entity doing the matching of content or topics against subscriptions. Not only is this a single point of failure, but it is also likely to become the bottleneck in scenarios involving very large numbers of publishers and subscribers.

Different approaches to this problem includes arranging the message brokers in clusters, adding distribution to this otherwise centralized component. This is the approach used in Johka [16, Ch. 5]. Another approach is to add an overlay network of logical message brokers on top of the underlying, physical network [11, 124, 10, 121]. Figure 2.5 shows an overlay network with a star topology implemented on top of a physical network with a ring topology.

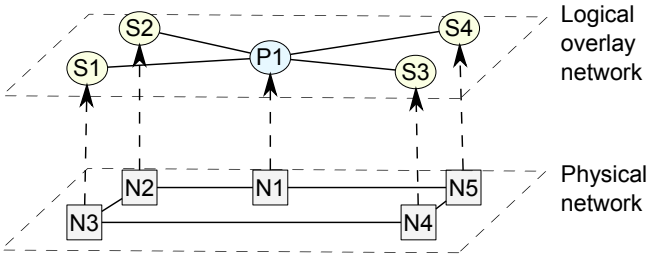

Figure 2.5: Overlay network

Subscription Models

Publish/subscribe systems offers different ways for subscribers to express their interest. This section covers the most basic subscription models.

Topic-based subscriptions is a static subscription model, where events are grouped into *topics* that can be subscribed to. For instance, a topic name for political news from the region of Europe could be `news.europe.politics`. Topics are closely related to the concept of *groups* from the research area of *group communication*, and most early implementations of the publish/subscribe paradigm were based on the topic subscription model [47, 34].

Since subscribers may only subscribe to predefined topics, and the publisher must decide the topic before sending an event, the model offers limited expressiveness [30, 11]. However, more recent implementations [2, 75] have extended the model with concepts such as hierarchical topics and convenience operators such as wildcards. Using the previous example, a subscriber to the topic `news.europe.>` would receive any news event sent to any subtopic of `news.europe`, while a subscription to `news.*.economy` would match all economy news.

More expressibility is offered with **content-based** subscriptions, which enables subscribers to filter the events sent to a topic based on content. By evaluating message properties, expressed with SQL-like syntax, regular expressions or with eXtensible Markup Language (XML), the subscriber can express interest in events that match a specified set of conditions.

To exemplify the SQL-like syntax for content filtering, a filter like `price < 20 AND name = 'IBM'` applied to a topic subscription for stock quote events would only forward events where the stock being exchanged is named 'IBM' and the price is below 20.

Most current topic-based publish/subscribe specifications, such as JMS [69], allows for filtering based on properties. As such, events are usually represented as maps of primitive types rather than objects. Relying on indexed structures like key-value pairs facilitates very efficient implementations, but comes at the cost of type safety, as misspelled property names may only be detected at runtime [47].

Another approach to content-based filtering is to represent events as binary objects, which ensures type safety, but requires deserialization at the inspection stage. For this reason, this approach does not scale well [47].

A more recent model, **type-based** subscriptions, introduced by Eugster [47], alleviates many of the shortcomings mentioned in the previous paragraph. The event communication model is based on objects, and does not include any notion of topic hierarchies, instead, the *type* of the event object is the primary event discriminator. Because there is no need to encapsulate event objects into predefined schemas, transformation and deserialization along the chain of distribution is no longer necessary. Furthermore, the object based data model, means that evaluation of content can be provided through publicly accessible *methods* on the event objects, ensuring type safety. To exemplify, with the type-based model, the previous SQL-like example of content filtering, can be expressed in the following manner:

```
e.getPrice() < 20 && e.getName().equals('IBM')
```

In this example, `e` represents an arbitrary event object, where `getPrice()` and `getName()` are typechecked methods.

2.4 Middleware Models

RPC and MOM are arguably two of the most widespread models of middleware [97], and builds upon the basic interaction patterns described in the previous section. A majority of distributed applications use one or both of these as the underlying model for communication.

This section provides a broad overview of the characteristics of RPC and MOM, discusses strengths and weaknesses, and includes a short presentation of commonly found implementations for each respective paradigm. The relationship between the middleware models and the basic interaction models is illustrated in Table 2.1.

	RPC	MOM	
Interaction	Request/Reply	Message Queuing	Publish/Subscribe
Style	Pull	Push/Pull	Push

Table 2.1: Relationship between middleware models and interaction models

2.4.1 Remote Procedure Calls

Aiming to hide much of the network complexity of distributed applications, RPC was introduced in the early 1980s [145], providing application developers with a higher-level abstraction for communication. Procedures could then be invoked remotely from systems running different operating systems, appearing to the programmer as if they were locally stored procedures.

Most incarnations of RPC use the synchronous request/reply interaction model (Figure 2.6), although asynchronous callback has been introduced in some variants. With asynchronous callback, the caller does not block after the invocation, but instead specifies a callback object that will be notified when a result is ready, leaving it available for other tasks while the call is being processed. In cases where no return value is needed, asynchrony can also be achieved by simply having the server acknowledge that the call will be processed, immediately after it has received the call (Figure 2.7), instead of waiting for the operation to be carried out before replying [140].

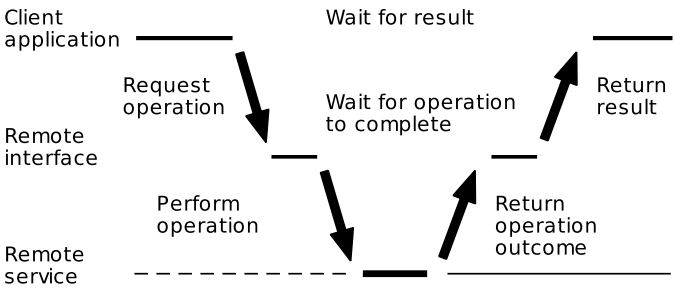

Figure 2.6: Synchronous interaction with RPC. The figure is adapted from [96, Ch. 3.1].

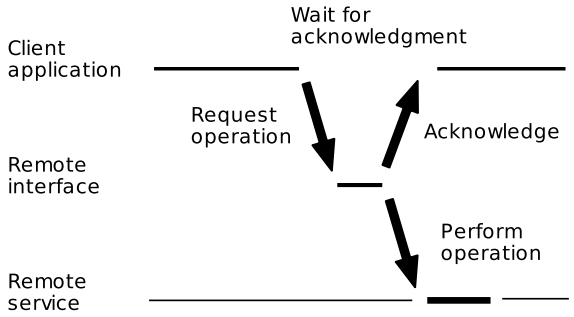

Figure 2.7: Asynchronous interaction with RPC.

The RPC model comes with some serious limitations in terms of scalability and flexibility [48, 114] that stems from its tightly coupled nature. In the context of information systems modeling, *coupling* refers to the relationship between components. When software components are interdependent, they are said to be tightly coupled.

With the RPC model, the caller has to know the address of the remote interface, and it has to have what is referred to as a *client stub*, which is a local interface that serializes the method call and its parameters into a message before sending it across the network. On the server side, a *server stub* receives the call, deserializes it, and sends it to the native server-side procedure. In order for this to work, both the client and server stub has to be in sync, which means that one cannot make changes to one part of the system without potentially affecting functionality located elsewhere. Communication is done in a point-to-point manner, where distributed parts of the system are interacting directly with each other. As far as the application is concerned, all procedure calls are made on local objects, and thus, the distribution layer of the application is transparent to the programmer.

Because of this distribution transparency, it is very easy for application developers to use, and hence, very popular [48] [140, Ch. 2.2.3]. However, in order to provide this transparency, it requires both the caller (client) and executing part (server) to be present on the network at the same time, with the client being blocked until the server returns with the result value (unless a separate

thread is assigned to this task).

A real-life analogy to the RPC model is when you, as a customer calls a support line and is put on hold if there is a queue, forcing you to hold the line until it is your turn. As such, it is not suitable for long-running procedures, because it ties up the caller for a corresponding amount of time. Conversely, asynchronous RPC is analogous to the customer service lines that lets you press a key, and have them call you back when it's your turn. Instead of holding the line, the customer is free to do other tasks in the mean time. Asynchronous RPC helps alleviate the problem of tying up the caller, but still requires the caller to know the location of the addressee, and therefore is still tightly coupled.

The performance of the RPC abstraction can be acceptable in a LAN where network bandwidth is cheap and node locations predictable, but quickly breaks down in a WAN environment, where communication links and computing nodes can be volatile [121, 12]. Because it ignores the possibility of *partial failures* caused by either failures in the network, or in remote systems, RPC introduces the need to handle exceptions that would never occur in locally stored procedures [145]. As such, it could be argued that the abstraction of RPC in many instances introduces more complexity than it hides, especially in large and widely distributed systems.

RPC implies a one-to-one communication model, where systems are connected directly to each other. In addition to affecting scalability, a big drawback of the one-to-one communication model is the structural rigidness that comes with it, making it very hard to implement changes to one part of the network without potentially affecting other parts at the same time – a factor that becomes increasingly unpredictable as distribution grows. This is often referred to as *coupling* in the literature [82, 57], and is the term that will be used to describe the degree of interdependency between system components for the rest of this thesis. As illustrated in Figure 2.8, a single request from a client to a server may depend on an array of sub-invocations against other subsystems, where even the temporary unavailability of a single subsystem could potentially cripple the whole system, due to interdependency issues [35]. This is often referred to as multi-tiered systems.

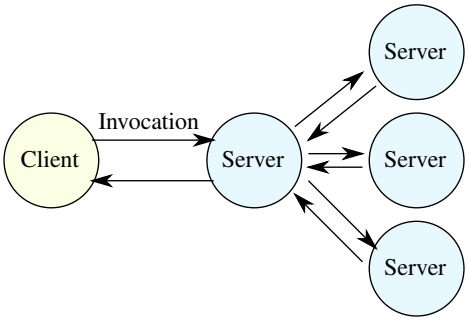

Figure 2.8: Multi-tiered method invocation

Representational State Transfer

A challenge in distributed systems is when multiple servers has to share a common application state. A common reason for designing systems in such a way is to reduce the number of method calls and limit the amount of information needed to be transmitted over the network. However, this may lead to conflicting states and complex systems that are hard to debug, where the complexity grows along with the distribution of the system.

Based on the principles of the modern Web and defined by Roy T. Fielding [54], the Representational State Transfer (REST) architectural style aims to solve these problems by eliminating server-side state alltogether. REST belongs in the RPC family of models, but puts a number of restrictions on the system design, with the intention of improving uniformity of interfaces as well as scalability.

RESTful interactions are stateless by definition, and constrains distributed applications to be designed in such a way that each request contains all the information needed for the server to understand the request, regardless of any requests that may have preceded it. This way, every interaction is *independent* from other interactions. The downside of this is that the entire context of each method call must be included every time, resulting in increased network traffic.

To illustrate this, consider a RESTful Application Programming Interface (API) for a shopping cart at a webstore: The state of the transaction will be stored locally at the user end, in order to keep the server stateless. In other

words, the user have to send the complete state of the transaction to the webstore API, including items and quantity for every step of the purchase process.

RESTful APIs use HTTP for all interactions, and are thus limited to the `GET`, `PUT`, `POST` and `DELETE` methods of the protocol, ensuring *uniformity* of interfaces. While the RPC paradigm promotes the implementation of custom methods like `deleteAccount()`, `getCustomer()`, etc., the advocates of the REST paradigm argues that this diversity of methods is unnecessary, and that all applications can be served using the four basic functions of persistent storage; Create, Read, Update and Delete (CRUD). In this style of API, the basic functionality of the method is described in the identifier, e.g. the URI of the method.

Furthermore, the REST paradigm follows the client/server principle of separation of concerns, claiming to improve *portability* of client code as well as *scalability*, as servers need not be concerned with user interfaces or client state.

Standards and Implementations

This section covers some of the most well known specifications of the RPC model. These include Open Network Computing (ONC) RPC [141], developed by Sun and used for the well-known Network File System (NFS) distributed file system, Common Object Request Broker Architecture (CORBA) [146], Java Remote Method Invocation (RMI) [66] [140], .NET Remoting [93] and Web Services [135], which are predominantly implemented in RPC-style.

Some of these implementations have a lower degree of coupling than others; The Web Services concept that was introduced around the millennium, and has proven successful in the inustry because of its platform *and* language independene through the use of text-based data formats exchanged over HTTP [53] [66, Ch. 2.2.1] for representation of method calls. A large contributor to the success of Web Services, over, say, Java RMI is its use of port 80, which is normally left open in firewall configurations, thus enabling deployments to bypass the most common network-level security obstacle.

The traditional way of exposing Web Services is through a Simple Object Access Protocol (SOAP) [135] API, although RESTful Web Services eschewing

the SOAP layer in favour of a simpler HTTP-based api has become popular in recent years.

All implementations referenced here, except .NET Remoting, are platform independent, although one *may* get .NET Remoting to work on Linux platforms, through the Mono framework [102]. However, this is not officially supported by the vendor (Microsoft), and there is uncertainty associated with using complex data types, such as hash tables, as these may have a different representation on other platforms.

The Java language is platform independent, which means that Java RMI can work on any platform that has a Java Runtime Environment (JRE) installed. Java RMI and .NET Remoting are not language independent, which comes with the obvious downside that objects written in different languages cannot communicate with each other. The advantage is that these are generally easier to work with, as the programmer does not need to account for typechecking, can use a richer set of data types, and can rely on language-specific functions, such as automatic distributed garbage collection [66, Ch. 23].

Using language-independent implementations of RPC limits supported functionality and data types to those included in all programming languages supported. Another advantage with language-tied RPC implementations, is the ability to send any serializable object across the network without having to define it in an interface first. With language independent implementations like CORBA or SOAP, one has to define the object, using Interface Description Language (IDL) in the case of CORBA and Web Service Definition Language (WSDL) Schema in the case of SOAP.

An extension of Java RMI; Java RMI over IIOP [127] [66, Ch. 23] enables clients written in other languages to communicate with distributed Java objects. IIOP is an acronym for Internet Inter-ORB Protocol and is the wire protocol part of the CORBA specification. Its interoperability with other languages means that the data types and functionalities are limited in the same way as with SOAP, meaning that data types are limited to a selection of generic types, supported by most languages.

A part of the Java Enterprise Edition platform, the current version of Enterprise

Java Beans (EJB 3.1) uses Java RMI-IIOP for communication, and provides an array of additional functionality, such as transaction handling, support for message driven interactions and thread management. It will not be discussed in detail here, where the focus is on remote procedure calls.

An argument for using platform and language independent middleware implementations such as SOAP and CORBA is to avoid vendor lock-in. When choosing technologies such as Java RMI or .NET Remoting, it ties oneself to a particular programming language for all future interactions with the system (and in the case of .NET Remoting, to a particular operating system as well).

2.4.2 Message-Oriented Middleware

A common solution to the issues associated with synchronous interactions and the RPC paradigm is to loosen the coupling between communicating systems: By introducing a mediator between nodes in the form of a message queue abstraction (Figure 2.3), we are able to decouple both the time and space dimensions, given that the message queue is persistent, and at the same time remove the interdependency of systems. The message queue may be implemented in a centralized or distributed manner. Either way, the abstraction presented to the clients are a single message queue.

Middleware that provides the programmer with an API for messaging is called Message-Oriented Middleware. Compared to the RPC model, where the distribution is invisible to the application programmer, MOM adds a layer of complexity by exposing parts of the distribution. The operation of MOM is analogous to the postal service taking responsibility for the delivery of messages to the correct recipients.

Communication in MOM is usually asynchronous, which prevents the whole system from being bogged down by the single slowest entity, since message consumers does not have to wait for producers to process a request before continuing and vice versa. The flexibility of MOM facilitates support for messages that may take anywhere from seconds to hours to process. The two interaction models typically provided by MOM is *message queuing* and *publish/subscribe* [35].

As mentioned in Section 2.3, message queuing is a one-to-one, loosely

coupled interaction model, where a message is placed on a *queue* by a single producer, and pulled from the queue by a single consumer. Publish/subscribe, on the other hand, is a many-to-many interaction model, where events are distributed by an event notification service that matches events of interest to *subscriptions*. Thus, events may be delivered to more than one subscriber.

As long as message passing is not integrated into most established, modern high-level programming languages such as Java, C# and .NET, MOM is a necessity in order to solve certain types of tasks. Thus, the developer has to rely on external APIs for message passing, which has the drawback of the compiler not being able to statically type-check messages [121]. Some of the more recently developed programming languages, like Scala [115] and Go [106], includes message passing as one of the core features of the language itself, essentially integrating the middleware layer into the language.

Synchronous interactions with MOM

Although MOM is inherently asynchronous in design, synchronous interactions can be achieved by including a property that defines the address of the message queue where a reply to the message sent by the client will be found. The message producer may then block until it finds a reply on the specified address. In order to keep relationships between request and reply messages, the requester may specify a temporary reply-queue for each request. However, this is bad for performance, as the message broker would have to create a new queue for every request. A common way of avoiding this is to use an additional property, containing an identifier in order to correlate reply messages to their original request, using a single queue. These techniques involves maintaining maps of message identifiers linked with messages at the requester side.

From the steps described above, it is evident that achieving synchronous interactions using MOM is cumbersome. Developers seeking to add synchronous interactions to their applications will likely be better served utilizing a communication model that was designed with synchronous interactions in mind, such as RPC.

Common Functionality

A message queue in its simplest form could be an FTP server where one or more systems upload messages in the form of text files, while other systems monitor specific folders, and downloads new files as they appear. This is not MOM per se, as it lacks essential functionality typically expected from a MOM implementation such as delivery guarantees, transactional integrity and so forth. However, it is still a common method for many businesses to integrate their heterogeneous systems in place of a fully fledged MOM. Below follows a list of some of the additional services to messaging that are commonly found in MOM [35].

- **Guarantee of message delivery**. Depending on configuration, messages can be stored on persistent media such as disk until they are delivered to all the subscribers, or until their Time To Live (TTL) expires, ensuring that information is not lost in case of a server crash or network failure.
- **Prioritizing of messages**, affecting messages in transit. In other words, it affects the ordering of when queued messages are delivered to the consumer. In the case of events demanding immediate processing, such as alarms, prioritization will ensure that those messages are delivered to the consumer before less important events, such as logging notifications.
- **Transformation**. Methods for transforming messages from one format to another, or altering their attributes.
- **Message filtering** capabilities is a defining characteristic of MOM, and is implemented in the ways already discussed in Section 2.3.3.
- **Clustering functionality** to provide redundancy and load balancing. With message servers arranged in clusters, load balancing can be achieved by spreading the load of distributing, filtering and transforming messages among the cluster. Such arrangements also comes with the advantage that in the event of server crash, the messaging service will still be operational, as the rest of the servers in the cluster will pick up the slack left by the crashed host.

Standards and Implementations

This section covers some of the most commonly found MOM implementations and specifications.

JMS is a specification that defines a messaging API for Java [69] [114, Ch. 9.1.3] [35, Ch. 1.5]. In order for a system to be JMS compliant, it has to implement a set of interfaces that provide a defined set of functions and interact in a specified way. A goal of the specification is to provide the programmer with just enough tools and concepts to build sophisticated messaging applications. The subscription model of JMS is topic-based, although most current implementations, such as HornetQ [75] and ActiveMQ [2] also suppport content-based filtering.

JMS is a client/server-oriented specification, where components are divided into *clients* and *providers* (representing the server part). Some JMS implementations use a cluster of servers in order to provide load balancing and fault-tolerance. The provider is responsible for handling the messages, and clients can choose any JMS compliant provider. Clients can be both producers and consumers of messages. Furthermore, the specification requires two communication modes; *message queuing* and *publish/subscribe*. Queues are stored at the server, and direct communication between sender and receiver is not supported. Most JMS implementations include a native interface in addition to the JMS interface, in order to provide functionality that lies outside of the JMS specification. A benefit of sticking to the JMS interface of a MOM is that there are several implementations to choose from, and from these, an IT department can replace one with another relatively easily, should the need for this arise. Instances where implementation development and maintenance has ceased is an example of such a situation.

A challenge with JMS and other MOM implementations in certain industrial settings, where a failure in the distribution of messages may prove disastrous, such as aerospace and defense systems, is the lack of support for Quality of Service (QoS) and fault tolerance [30]. A consequence of JMS' lack of non-functional requirements is that although JMS specifies QoS policies for message delivery guarantees, no such guarantees for message latency exist. Furthermore,

fault-tolerance in JMS is left to the provider to implement, as it is not part of the specification. Lastly, JMS' lack of type-safety, means that errors caused by discrepancies of data types is harder to detect than in a type-safe system. Since JMS does not provide a wire-level protocol specification for the messages, and only provides a Java API, interoperability is limited to the Java world.

Advanced Message Queuing Protocol (AMQP) [126] and **Simple Text Oriented Messaging Protocol (STOMP)** [101] specify much of the same functionality as JMS, but includes the wire protocol as well, ensuring interoperability across platforms. As indicated by their names, STOMP is a relatively simple text-based protocol, while AMQP provides more functionality, and supports binary object representations as well.

Standardized by the Object Management Group (OMG), **Distributed Data Service (DDS)** [78] is a newer specification than JMS, designed to cater for demanding real-time applications, where dependability and timeliness of event notifications are paramount. It offers a fully distributed implementation of the publish/subscribe model, and is based on a Peer-to-Peer (P2P) networking model in order to cope with highly dynamic (unreliable) networks. Furthermore, it offers type-safety and uses a type-based subscription model, where QoS is part of the data type and thus, part of the subscription. While JMS is an API specification only, DDS also specifies the wire-protocol, as well as a number of non-functional requirements ensuring QoS and reliability.

A key abstraction of DDS is a *Global Data Space* (GDS), where data objects are available to publishers and subscribers, and a topic is a data type that can be legally written to the GDS. Because a primary goal of DDS is to ensure reliability and timeliness of event delivery, the DDS specification requires the GDS implementation to be fully distributed, in order to prevent a single point of failure or a single bottleneck. In order for a publisher to publish events in DDS, it associates a *DataWriter* object with a data type and a topic name, and use this to write to the GDS. A subscriber uses a *DataReader* object associated in the same way, in order to read values from a topic [31].

Originally developed for instant messaging, and endorsed by the Internet Engineering Taskforce (IETF), **Extensible Messaging and Presence Protocol**

(XMPP) (previously known as *Jabber*) [129] is a decentralized client/server-oriented messaging protocol for streaming XML in near real-time. In this case, decentralized means that the dissemination and filtering of messages can be distributed among servers, and that clients may communicate indirectly through multiple server hops. Its main interaction model is point-to-point messaging, but it also offers a general publish/subscribe interface, available as an extension [129, Ch. 3]. Because message content are limited to XML, it is not possible to transmit optimized binary content, which could prove a concern in scenarios requiring very high throughput. Furthermore, the focus of the protocol is one-to-one messaging, and publish/subscribe is not supported natively, but through an extension. Like with JMS, QoS is not part of the XMPP specification.

2.5 Event-Based Systems

Event-based systems represent a fundamental shift from the traditional ways of building distributed applications by inherently decoupling the components. This allows for more flexible and extensible architectures, as components can be removed and added without consequence for the rest of the system. Furthermore, the distributed nature of event-based systems allows for excellent scalability.

However, all of this comes at a cost: Whereas poll-based systems must trade the timeliness of data for resource expenditure, the tradeoff in event-based systems is between increased complexity and reduced control over the interactions on the one hand, and improved flexibility and scalability on the other [114, Ch. 2.2.5].

This section provides an overview of the components that make up event-based systems, and how these interact. We also discuss how these relate to previously introduced concepts. While Section 2.3 and Section 2.4 explained and compared different publishing models, this section focuses on the event processing part, and compares the various event processing models.

2.5.1 Event-Driven Architectures

While MOM provides a distribution layer for events in distributed applications, an Event-Driven Architecture (EDA) is a software architecture that introduces state in the handling of events. In this thesis, we define an event as "a significant change of state" [22]. According to Chandy and Schulte, an EDA adheres to the following five principles [22, Ch. 3]:

- Events are reported in real-time, as they happen
- Notifications are pushed by the event producer, not pulled by the consumer
- Consumers respond to events immediately
- Notifications are communicated one way ("fire and forget")
- Notifications are free from commands

A focus of event-based systems is to provide the quickest possible response to events of interest. Because notifications are only communicated in one direction, and are free from commands, producers and consumers in this kind of architecture have minimal coupling to each other. All a producer needs to know is where to send notifications, while all a consumer needs to know is where to listen.

An important distinction to be made between EDA and MOM is the level of abstraction and scope: One can look at MOM as the plumbing that enables events to propagate through an event-driven architecture, while the system itself also needs to know how to identify and handle these events in order to be called event-driven. The event processing part of the architecture can be separated from the publishing part, which can be anything from generating an email or sending an SMS, to sending a message to a queue, which brings us to the following point: When discussing event-based systems, the interaction model is often confused with the underlying implementation for distributing notifications. Even though a publish/subscribe service is an obvious candidate for implementing event-driven interactions, it is not the only available choice.

Notifications in event-driven interactions may be transported using any underlying communication implementation, provided that there is a publish/subscribe interface on top, facing the producers and consumers of events [114, Ch. 2.2.6].

Figure 2.9 illustrates the elements of an EDA: Event producers and consumers that communicate over an Event Processing Network (EPN), made of Event Processing Agents (EPAs). An EPA is a software component that performs one or more of the following three functions: *filtering*, *matching* and *derivation* [119, Ch. 6.2].

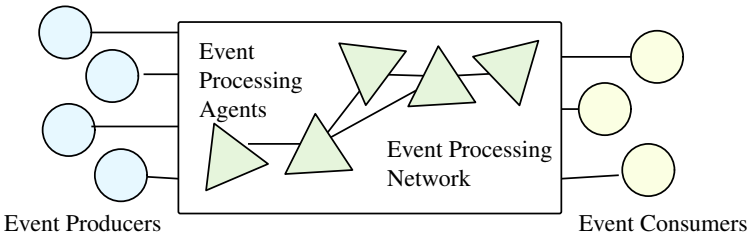

Figure 2.9: Event-Driven Architecture (adapted from [34])

Producers and *consumers*, sometimes referred to as *sources* and *sinks* [34, 22], represent the start and end of an event-driven interaction, starting when a producer emits an event to an event processing network, and ending when the event is received, and potentially acted upon by one or more consumers.

The event producer is the entity that introduces events from the outside world into the EPN [119, Ch. 2.2.1]. As such, the event producer is often is not the actual generator of the events, but rather an *adapter* that receives events from a source like a sensor, translating and communicating these to the EPN.

Producers are autonomous in that they decide for themselves when to emit an event, and furthermore does not rely on any further action to be made by other components [114]: They are unaware of the consumers of their emitted events. Likewise, consumers are unaware of the actual producers of events, and issue their interest in events through *subscriptions* to the event notification service.

In other words, the basic interaction model between event producers and consumers is publish/subscribe. The difference lies in the expressiveness of subscriptions supported by the event processing network that resides between the producers and consumers. While the event notification service in publish/-

subscribe systems generally only allows for the processing of single events in isolation, without regards for context, the event processing network (presented in more detail in Section 2.5.4) allows for stateful subscriptions. The various event processing models are covered in Section 2.5.5.

Even though producers and consumers in the context of the EDA model are limited to the entities facing the EPN, a discussion of the actual sources of events is in order, as they are an integral part of event processing applications. Categories of event producers include *hardware*, *human interaction* and *software*[119]. In this section, these categories are exemplified and discussed.

2.5.2 Event Producers

The following list contains examples for each category of event producers:

- Hardware
  - Sensors (temperature, luminosity, humidity, etc)
  - Detectors (smoke, pressure, intrusion, etc)
  - Cameras
  - Microphones
- Human Interaction
  - Appliance control (TV channel change, mute audio, turn alarm off, etc)
  - Identification (for instance using RFID chip or PIN code)
  - Payment through a device such as a credit card terminal
  - Placement of an order (via web or other)
  - Indication of presence (checkins through Facebook, etc)
  - Clicks on a web site
  - Social network activity
- Software
  - Applications
  - RSS feeds
  - Instrumentation (monitors, probes)
  - Adapters

For the main categories, the distinctions are rather subjective and their boundaries should not be considered rigid. Subcategories in the realm of hardware event producers include sensors, detectors, cameras and microphones. Sensors and detectors report one aspect of their environment, and as such, it could be argued that cameras and microphones are sensors, instead of having their own categories. However, the complexity of the output signals requires decoding in order to look at the video on a per frame basis, or a per sample basis with audio. Furthermore, there are many different aspects of the signals one could look at. In the case of a video signal, one could look at the movement in the picture, or at the color spectrum, among other things. Similarly, there are many aspects of an audio signal that can be analyzed; variations in amplitude, or the frequency content, to name a few.

The output of sensors is more continuous in nature than detectors, which are generally non-linear. Sensors may be polled, report at a fixed rate, or only output an event when its readings crosses a predefined threshold. Hardware sensors report a single aspect of its environment, measuring a physical quantity such as luminosity or humidity, while detectors detect the absence or presence of something, such as smoke or movement. Other examples of hardware sensors include medical equipment, such as heart rate monitors and blood pressure meters.

The Web is an abundant source of events generated by human interaction; clicks on web pages, payment through web shops and activity on social networks are all examples of such events. Another type of events in this category are those generated from users interacting with networked appliances such as STBs and alarm panels.

In the software category of event producers, we have code that generates events based on application logic, probes in industrial systems, and *feeds* of various kinds, such as RSS or Atom, or financial stock feeds [119, Ch. 4.2.2].

2.5.3 Event Consumers

Event consumers are at the other end of an event-based interaction, and is the entity that receives and possibly reacts on a produced event. The following list contains examples for each category of event consumers:

- Hardware
  - Actuators (door locks, alarm sirens, etc.)
- Human Interaction
  - Monitoring dashboard
  - Email, SMS
  - Social networking
  - Alarm systems
- Software
  - Event logs
  - A WebSocket listener, updating a web page

An *actuator* represents the logical counterpart to a *sensor*. Areas of deployment include home automation, industrial control and traffic control. A traffic light, for instance, is an example of a traffic controlling actuator. When controlled by a luminosity sensor, the small electrical motor controlling the blinds on the windows of a house is an example of a home automation actuator.

In the human interaction category, we find various monitoring dashboards, such as HP OpenView [76] and Cacti [103], which displays continuously updated graphs and readings for an array of various indicators for things like network latency, temperature, and CPU and memory utilization. Monitoring dashboards can also visualize other, non-system attributes, like stock prices or organizational health. The display of readings in context, enables operators to perform their own, manual Complex Event Processing (CEP). In cases like this, humans are the event consumers, and these systems depends on human intervention for actions to be taken.

Social media networks such as Twitter and Facebook presents another example of the human interaction category. In social networks, humans are both producers and consumers of events, where status updates, friend requests and shared links serve as the produced events, which in turn are consumed by other people, and possibly responded to, producing new events.

The software category of event consumers are reserved for applications without a user interface, such as event logs, and also include business logic that is not part of the EDA itself [119, Ch. 5.2.3].

2.5.4 The Event Processing Network

The disseminating part of EDAs, the event processing network, is made from event processing *agents*, connected by *channels*. Channels are any means of distributing events from one agent to another. One obvious technique for this is message passing, but there are also other ways, such as putting an event object in a shared place [22], like in DDS.

Event processing agents are software that consumes events, processes them, and emits new events, or simply forwards specific events, based on a filter criteria. Input events can either be *raw*, meaning unaltered events generated by event producers, or *derived*, meaning that they are events processed and emitted by other software agents. In that regard, event processing agents can be seen as both event consumers and producers, however, these categories are reserved for components residing outside the boundaries of the event processing network [119].

There are several subcategories of event processing agents: some merely perform simple filtering, while others perform more complex tasks, such as transformation, pattern matching and aggregation, composing new output events from heterogeneous input events (Figure 2.10). Examples of event processing agents include:

- JMS subscription filters
- Database triggers
- Pattern matching statements or queries written in a specialized language
- Blocks of code, dedicated to specific event processing tasks

2.5.5 Processing Models

While the traditional way of processing data is to store them in a database, and subsequently query them, many scenarios, such as event clouds within busi-

nesses (Figure 2.11) calls for event stream processing in real time. In many cases, the sheer volume of input events as well as the need to make decisions in real time, or near real time like in algorithmic trading, where a millisecond of latency could potentially cost millions of dollars, mean that storing the data prior to processing is not a viable solution.

Different models reflects different challenges, and here we will look at the main categories of event processing, and show some areas of usage for each of them.

Simple Event Processing

Most modern Database Management Systems (DBMS), such as Oracle, Postgres and MySQL, support some form of event processing in the form of triggers. *Active Databases* is a term commonly used with reference to these kinds of features. A trigger is a stored procedure that fires once a specific event is detected, e.g. when a table is updated with a certain value. Triggers typically come in the form of an Event-Condition-Action (ECA) rule, normally written as *WHEN event IF condition THEN action*. However, triggers does not cater for the demands of high volume, real-time event stream processing, as the data will have to be persisted prior to the firing of a trigger. Persisting the data prior to processing adds significant latency, and as already mentioned, the sheer amount of data will in many cases prevent this from even being an option. Moreover, simple ECA rules does not provide enough expressive power to represent complex events/patterns.

A common characteristic shared between ECA rules described in the above paragraph and the subscription models in publish/subscribe, presented in Section 2.3.3, is that they operate on single events, without regard for context. In other words, routing and filtering can only be based on the properties of a single event at a time, which characterize simple event processing. Another characteristic of simple event processing is that the only types of processing available is *filtering* and *routing* [119].

Operating on a single event at a time, simple event processing is restricted to trivial, boolean matching in isolation. Deciding whether a soccer event was a

goal or not, is a simple action that does not require any knowledge derived from other events.

Illustrating this, Listing 2.1 is an excerpt from the HornetQ configuration of a soccer event notification application, developed for Altibox. This configuration establishes a forwarding of events from a specified address to another address, filtering out events that are of the type specified in the filter string.

Listing 2.1 Message filtering in HornetQ

```
<divert name="livecenter-divert">
  <address>soccer.events</address>
  <forwarding-address>
    soccer.events.livecenter
  </forwarding-address>
  <filter string="type<>26 and type<>27 and type<>54"/>
</divert>
```

Complex Event Processing

CEP describes the process of identifying patterns of events in relation to other events. Complex events are sometimes referred to as *Composite Events* [114], and can be viewed as *superevents* because they are composed from simpler entities. Events of this type can be made of any combination of other events, both simple and complex, as illustrated in Figure 2.10.

Listing 2.2 Event Processing Language (EPL) query generating a complex event

```
1 insert into AdSummary
2  select * from pattern [every
3   (a=ViewersLost and b=MuteCount and c=VolSummary)]
```

Listing 2.2 shows an example from our ADSCORER implementation (presented in Chapter 7), and shows a complex event generated from other complex events.

Examples of complex events include intrusion attempts on a networked system and credit card fraud detection [132], where various events that may not mean much as isolated occurrences may be recognized as parts of a larger event when looked at in relation to each other. In order to detect a sophisticated attack, the detection system must correlate several different events from disparate

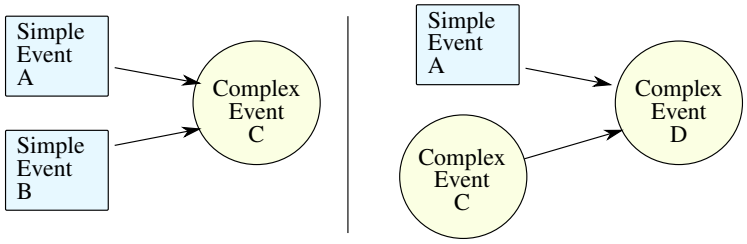

Figure 2.10: Complex events

sources, generated at various points in time and recognize an event pattern from these.

Detecting complex events is often no simple task and, depending on the complexity of the events, the number of states necessary to remember, the number of different event types, the volume and size of events, might require significant processing and memory resources. As such, the performance and scalability requirements are in many cases a concern when designing such systems.

Event Stream Processing

Any set of events ordered by time can be viewed as streams [89], and can be formally represented as an ordered pair (s, Δ), where s is a sequence of tuples and Δ is a sequence of time intervals.

A subset of CEP, Event Stream Processing (ESP) focuses on processing high-volume event streams, such as market stock ticker feeds and traffic data, where the number of input events are relatively large in comparison to the generated output events. Being able to process events in the order of their arrival allows for high performance and low memory usage, as the processing engine does not have to keep track of a very large set of events. A sliding window technique, where old events are discarded as new ones arrives is commonly used. This is illustrated in Listing 2.3, which is taken from an Esper [28] tutorial. Esper is a stream focused event processing engine, and is presented in more detail in Chapter 4. The query demonstrates a sliding window that returns the average stock price of the events that occurred over the last 30 second interval.
Note that the sliding window concept is not restricted to time-based windows; it

Listing 2.3 EPLnumbers query performing Event Stream Processing

```
1   select avg(price) from StockTickEvent.win:time(30 sec)
```

could also be specified as the number of consecutive events to examine.

One can look at ESP as an inversed model of the relational database management systems, where, instead of running a query over a set of stored data, pulling out information, you store your query, and run incoming data over it in a continuous manner. When incoming data matches the query, a notification is pushed out.

Algorithmic trading is a businesses example that fits this processing model perfectly, but also any other application that needs to handle near real-time processing of high volume event input streams, consisting of few and well-known event types.

2.5.6 Applications For Event Processing

This section covers some practical application areas for event processing:

Alarm systems are a good match for the EDA paradigm: events are passed through the system, interacting with event processing agents, possibly triggering an action based on the context of the events and the state of the interacting components. If an event occurs within a specified timeframe, for instance, the appropriate action might be to trigger an alarm. This scenario represents event processing in its simplest form. Continuing with the alarm example, take an IR detector that provides a continuous stream of events; by specifying a threshold for consecutive positive readings, we have an example of event stream processing, where an event processing engine is required to keep track of state.

Tools for **real-time log analysis**, like Splunk [110], give system operators a chance to detect network-wide failures with much less effort than allowed by manual log inspection. Basically, this type of tool monitor an array of system logs, and enable administrators to express interest in specific keywords, as well as specify rules and thresholds for when, where and how many times these keywords might appear. This can either be expressed in a specialized language, or through a graphical user interface. By combining logs from different

sources and analyzing them as a whole, such tools enable automatic plotting and bucketing of events according to properties such as time, host, and factors of interest [138].

Business Activity Monitoring [22, Ch. 10] (BAM) is another prime application for event processing, where events generated by different systems needs to be analyzed and correlated. Because the number of events generated in the event cloud of a business (Figure 2.11), or between businesses, can be very high (often in the scale of thousands per second), it is impractical, if not impossible to store all the data before processing them. This is where event processing enters the picture; by processing the events in real-time, an event processing engine makes it possible to detect critical business events (a pattern of event *A*, followed by *C*, followed by *B OR D* within a timespan of ten minutes, for instance). Specialized event processing languages simplify the expression of complex event patterns, even though this kind of logic is possible to implement using a general purpose programming language. However, it generally requires a significantly larger effort, as demonstrated in Section 4.

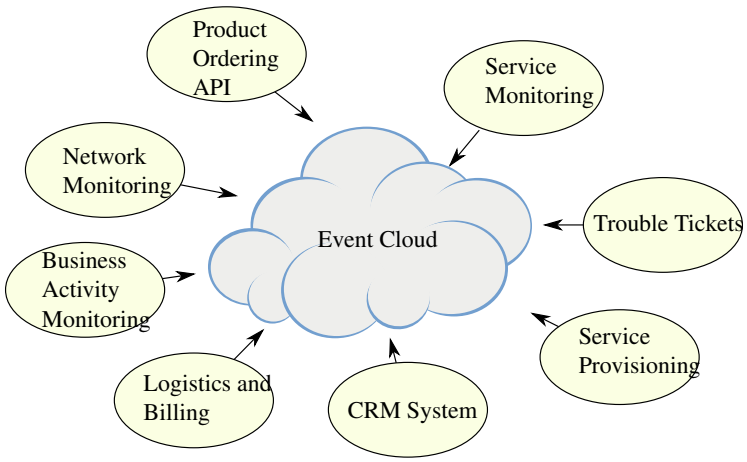

Figure 2.11: Business Event Cloud

Electrical power grids also fit neatly into the EDA model. Between the devices plugged into power outlets and the power plant, there is a multitude of additional components that can be classified as event producers, consumers, or both. Smart meters [81, 79] reports power usage for an installation in real time, and is a

typical producer, consumed by the backend applications at the power company.

A number of **smart grid** initiatives [122, 65, 70] aim to reduce the number of power outages and maintenance costs caused by overloaded transformers and at the same time reduce overall consumption through smarter utilization of the available power. The use of probes, installed across the power grid is central to achieving these tasks, but it is also possible to imagine that smart meters alone, combined with knowledge of the existing grid could provide sufficient information in order to calculate the load on the components that make up the power distribution network.

Sensor networks for **smart home** environments is another obvious arena for event processing. These come with their own set of challenges, which will be discussed in the following section.

2.6 The Sensor Network Application Domain

This section elaborate on the challenges involved in developing applications for sensor networks within the context of smart homes (Figure 2.12), and how event processing provides a natural solution to some of these, providing the background for the challenges HETEROGENEITY, INTERACTION STYLES and SERVICE DISCOVERY as well as motivation for the SENSEWRAP middleware presented in Chapter 3. Note that the focus will be on middleware functionality, rather than low-level wireless communication protocols.

Wireless communication technologies enable seamless communication between residential network entities such as STBs, sensors, control units and other devices, and are typically far less costly to install than their wired counterparts due to cabling. These technologies have opened up a whole range of new applications in the utility segment, like remote control of heating, security and safety systems and health monitoring.

Because the sensors in a wireless sensor network report aspects of their environment, there is a limit to how long the readings are accurate or can be trusted, thus it is important that they are delivered to the consumers in a timely manner. These requirements matches well with the publish/subscribe communi-

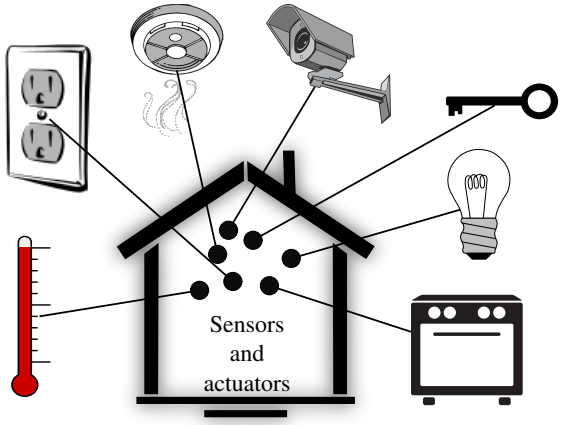

Figure 2.12: Smart home

cation model of event-based systems, while CEP is an efficient way of extracting meaningful information from a potentially vast amount of data produced by such networks [85].

In addition to the challenge of extracting meaningful information from a potentially massive amount of data from a variety of sensors, the heterogeneity of communication protocols and the mixture of addressing schemes used by networked devices of different make and model is one of the biggest challenges when developing integrated smart home services.

Most smart home systems offered today are based on proprietary all-in-one solutions, where the sensors and actuators might use a proprietary Radio Frequency (RF) protocol over the 868MHz band, while others might use ZigBee [68], Bluetooth [13] or WiFi. Furthermore, most devices have their own application-level protocol for communicating control commands and retrieving data. Moreover, due to the limited capabilities of many types of sensors, a full communication stack with IP addressing is simply unfeasible. Yet, it would significantly simplify application development if interaction with the sensors were based on UDP or TCP sockets and IP addressing schemes. Currently, these issues hamper innovation and development of new (possibly third-party) smart home services.

Another obstacle to the adoption of smart home technologies is the com-

plexity of setting up and managing the networking between devices, deterring most home owners from acquiring such solutions. Add to this the mobility and inherent unreliability (empty battery, communication problems, etc) of wireless sensors, one can imagine that the task of manually administering a smart home sensor network could prove to be very demanding indeed. Hence, it is paramount to the success of networked homes that device configuration is performed automatically.

2.6.1 Service Discovery

Minimizing the burden of manual configuration is a requisite for the success of smart home environments. Modern user devices are often *multi-homed*, meaning that they may connect through different networks and topologies, such as WiFi or 4G, which is not facilitated by the host-centric TCP/IP stack [117]. Finding a scalable and convenient way of handling service discovery in smart home sensor networks is a challenge, both due to the limited resources of wireless sensors and actuators, as well as the potential volatility of such devices.

In this section, we discuss the SERVICE DISCOVERY challenge in further detail, introduce the Zero Configuration Networking (ZeroConf) suite of protocols [25], and give a brief overview of other service discovery protocols..

Service discovery protocols are designed to enable systems to automatically find services in the network, without requiring user intervention. In order for an application to use a networked service, it will have to know the network address as well as the both the network and application protocols of the service. Configuring this manually might work satisfactory in a static environment administered by professionals, such as enterprises, but is not applicable to dynamic environments like sensor networks, where services may arrive, leave, or relocate.

Zero Configuration Networking

ZeroConf is endorsed by the IETF [77], through various RFCs. There are several implementations of ZeroConf for different platforms, e.g. Bonjour for Mac

and Windows and Avahi for Linux. ZeroConf has become a widespread protocol for automatically discovering external devices such as printers, cameras and iPads to communicate over an IP network. The protocol was designed for use in small (less than 1000 clients) local networks. In order to achieve automatic configuration of network devices, ZeroConf automates three core services: IP addressing, name resolution and service discovery [25]. In other words, IP addresses will need to be assigned automatically to each device and coupled with a meaningful name, and services have to be discovered automatically as they enter the network. This is achieved with the following combination of techniques [25]:

- *Link-local addressing*: Used to assign IPv4 addresses without relying on a DHCP server present on the network: The device picks an IP address from the reserved local private range of 169.254.x.x at random and sends some ARP requests, asking for the owner. If no reply is received, the device answers its own request, claiming the ownership itself. With IPv6, link-local addressing is no longer required, as each device already has its own local IP address, based on the MAC-address.

- *Multicast DNS (mDNS)*: IP addresses are impractical and difficult for humans to relate to, especially when picked randomly and subject to frequent change, as is the case with link-local addressing. When accessing web pages, users normally rely on the global Domain Name System (DNS) system to provide them with a map between a user-friendly address and the IP address of that site. The downside of the DNS system is that it requires dedicated servers which needs to be configured and administered, and is impractical for use within the home. mDNS offers the functionality of DNS in a maintenance-free version, for use in local networks, where the service runs on all the connected devices instead of a dedicated server. The principle behind mDNS is the same as with link-local addressing: Basically, the device sends a few mDNS queries for a self-assigned name, and takes ownership if no other device answers, providing name binding without the need for a DNS server.

- *DNS Service Discovery (DNSSD)*: Enable users to browse for services without having to know anything about the hosts providing them. It builds on existing standard DNS queries and resource types and provides service discovery without a centralized directory service. Instead each ZeroConf enabled device maintains its own directory of services, as shown in Figure 2.13.

The philosophy behind the ZeroConf platform is rooted in the assumption that end users are interested in services, not devices. The goal is that users should be able to select services from a list through a graphical user interface.

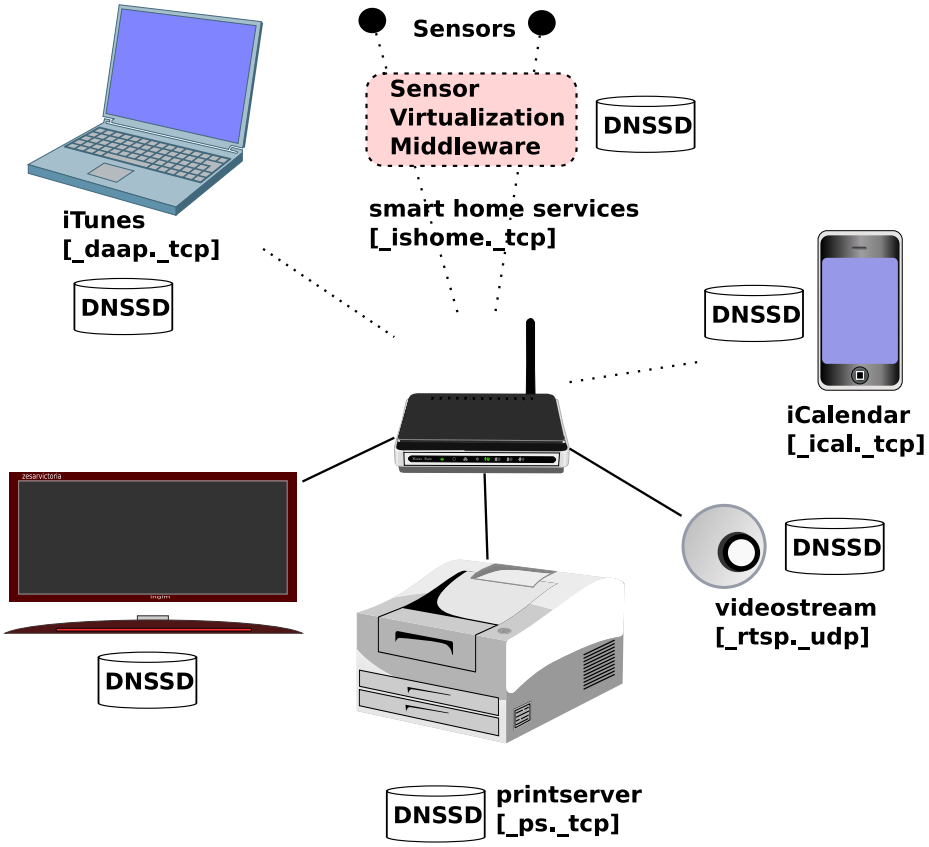

Figure 2.13: ZeroConf-enabled home network

Figure 2.13 illustrates a ZeroConf-enabled home network, where a laptop running iTunes software, a printer, a webcam, an iPhone and sensors can access

each other's services. Services are advertised in the format:

<Name><Type><Domain><Port>

where <Name> is the user-friendly name of the service, and <Type> is the service type. The webcam, for instance, will advertise its service as

''videostream'' _rtsp._tcp local 554

indicating that it offers a video stream over the RTSP protocol on TCP port 554. Each device has its own DNSSD instance, keeping a list of available services. In the illustration, the SENSEWRAP middleware (presented in Chapter 3) resides between the sensors and the rest of the network, as most sensor nodes does not have sufficient resources available to run their own DNSSD instances.

The list of services is kept up to date in a distributed and thought-out manner, using a combination of the following techniques to keep track of available services present on the network:

- Clients refreshes their local lists at irregular intervals, often as infrequent as once an hour, to keep network strain low.
- At startup, new services sends a few multicast DNS packets, notifying all clients on the network of their presence.
- When services leave gracefully, they send a multicast DNS goodbye message.
- If a service crashes, loses its network connection or in some other way leaves without being able to inform the network, the service stays in the clients' lists until the next time a client refreshes its list, or tries to access the service, in which case the client removes the service from the list and informs the other clients.

Combined, these methods prevent the network from being flooded with control traffic.

Other Service Discovery Protocols

ZeroConf is not the only protocol providing service discovery and automatic network configuration: Some protocols, like Jini, are solutions to specific problems, while others, such as Construct [44, 32] provides a complete platform for developing pervasive applications.

Service Location Protocol (SLP) is an IETF proposed standard, and is supported by some of the largest industry actors, including Hewlett Packard, IBM, Sun Microsystems [1] and Apple [9]. Apple did however replace SLP with DNSSD and mDNS as the preferred ZeroConf protocol between Mac OS X 10.1 and 10.2, which makes the technology somewhat obsolete.

Jini is Sun's take on service handling, and as such, it is Java-based. Theoretically, any communication protocol that supports serialization of objects could be used, but since Jini is built on top of RMI, it is not practical to use other protocols. Jini systems are divided into Service, Lookup Service and Client components. Although Sun maintains that it is platform independent, only Java is used in practice [15]. It needs to run within a Java Virtual Machine (JVM), and it is rather heavyweight. Even though it supports the Java 2 Micro Edition (J2ME) virtual machine, clients need to be able to dynamically download and execute Java classes, and small devices running J2ME typically does not have the processing power and resources to do this [40]. This can be worked around by including a proxy that executes the code and presents the data to the client through a servlet [139], but it nonetheless complicates matters.

Universal Plug and Play (UPnP) have many of the same objectives as ZeroConf, but while ZeroConf is a three layered foundation for automatic device configuration, UPnP is an organization, maintaining an open-ended collection of device-specific protocols [24]. Whenever a new device type appears on the market, the UPnP forum creates a working group to develop a protocol for that particular type of device. The application protocols is built on top of standard internet protocols such as IP, TCP, UDP, SOAP/XML and HTTP to ensure platform independence.

[1] Sun Microsystems was acquired by Oracle in 2010. We continue to use the name Sun herein for technology developed prior to this acquisition.

UPnP offers automatic addressing, service discovery, and comes with protocols for controlling sensors and actuators. A key difference between UPnP and ZeroConf is that while the UPnP organization is focusing mainly on application protocols without paying very much attention to the underlying layers, ZeroConf provides the underlying communication layers, but leaves it up to the developer to decide how the application protocol for a specific device is going to be implemented.

IP addressing is achieved in exactly the same manner as ZeroConf, using IPv4 link-local addressing. Unlike ZeroConf which have mDNS, UPnP does not handle name resolution and thus requires a DNS server present on the network to provide this. According to the UPnP Device Architecture definition [55], most often UPnP-enabled devices only provide URLs using numeric IP addresses. UPnP-enabled components are either devices, hosting services, or control points, controlling devices.

For use in smart home applications UPnP does have some disadvantages: For one, UPnP use heavyweight SOAP XML objects over HTTP for communication, requiring an XML parser on both ends and at the same time increasing processing and bandwidth usage. ZeroConf, on the other hand, uses standard DNS packets to advertise services, which are much smaller in comparison. Another problem with the UPnP protocol is its inherent chattiness; Its Simple Service Discovery Protocol (SSDP) was built on an IETF draft which was abandoned in 1999, partly because the working committee recognized that the network would become flooded with control traffic in a setting with more than ten SSDP devices communicating [24]. Another obstacle is that UPnP does not include support for prolonged periods of the network link being down, which is a likely occurrence in the noisy environment of a sensor-driven smart home.

Service Discovery in Retrospect

Some years have passed since the initial survey on service discovery protocols was performed. During this time, a couple of elegant solutions for overcoming the obstacles of the host-centric Internet protocols has emerged that is worth mentioning:

Serval [117] adds a service layer on top of the network layer, enabling applications to communicate directly via service names. This service layer is responsible for service discovery and resolving serviceIDs to network addresses, essentially providing the same functionality as DNSSD used in ZeroConf and SSDP used in UPnP. However, while the previously mentioned service discovery protocols returns an IP address and port number in response to a service lookup, needed by the application layer to communicate with the service, Serval enables direct addressing of services by the application layer through the use of a *serviceID*. Additionally, the middleware handles flow mobility and migration, leaving only the transmission of packets between endpoints to the transport layer, which allows addresses to change dynamically as hosts move.

eXpressive Internet Architecture (XIA) [72] presents a radical redesign of the Internet, and features a rich addressing scheme, where any attribute can be principal, in contrast to the current infrastructure, centered around the *host* attribute. The rationale behind not elevating one attribute above others is that it is impossible to foresee how the internet will be used in the future. Han et al. [72] describe how networks, hosts, services or content may be implemented as true endpoints using the XIA stack of protocols, giving extra attention to the case of elevating *processes* to addressable endpoints. In other words, XIA supports addressing services directly on the application level, much like Serval. Although current trends favours a service-centric approach to network addressing, services are only one of many possible principals of the architecture.

2.7 Summary

This chapter has introduced the concept of middleware at a general level, starting with the underlying motivation, presenting the basic client/server model and proceeding with the basic interaction models; request/reply, message queuing and publish/subscribe, providing special attention to the publish/subscribe model, as it is the interaction model of EDAs, presented in Section 2.5.

After covering the basics, we presented the RPC and MOM abstraction models, along with an introduction of their most prominent implementations.

The RPC abstraction effectively hides the system distribution from the developer at the expense of flexibility and performance, while the MOM paradigm is more complex, requiring the developer to be aware of the system distribution. However, MOM enables a looser coupling between components, which facilitates more scalable and modular software architectures.

Furthermore, we introduced the EDA paradigm and its components; event *producers* and *consumers*, and the *EPN* consisting of *EPAs* connected via *channels*. Following this, an overview of the various event processing models was provided. At the end of this section, application areas for event processing technologies was discussed.

Section 2.6 covered the sensor network application domain, linked it to the EDA paradigm and principles of event processing, and identified heterogeneity, dynamism and service discovery as its main challenges. This section also introduced the ZeroConf suite of protocols, and also provided an overview of other service discovery protocols.

Chapter 3

SENSEWRAP: Middleware for Sensor Virtualization and Self-Configuration

The SENSEWRAP middleware was developed with the goal of providing a general smart home services platform for the IS-Home project (presented in Section 1.1). The bulk of the content presented in this chapter has been published at the 3rd ACM International Workshop on Context-Awareness for Self-Managing Systems (CASEMANS) [50] in 2009, and at the 5th IEEE International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP) [49] later that same year.

Here, we present the design and implementation of a simple, yet elegant middleware architecture providing *virtual sensors* as representatives for any type of physical sensors. Our middleware, which we have named SENSEWRAP, combines the ZeroConf protocols with hardware abstraction, giving a service-oriented and lightweight middleware for application programmers to interact with.

Our virtual sensor abstraction provides transparent discovery of arbitrary sensor devices through the use of ZeroConf protocols [25]. This enable applications to discover sensor-hosted services through ZeroConf and it provides

a standardized communication interface that applications can use without having to deal with sensor-specific details. That is, virtual sensors also provides a uniform communication interface to clients, based on UDP/TCP connections or even HTTP. This is accomplished by abstracting functionalities common to most sensor models, and writing custom wrappers (drivers) for the specifics of each sensor model.

This way, applications need not know anything about the physical or logical communication protocols used by the sensors, making the same network services usable with any sensor model sharing the same basic functionality. For instance, a light-controlling application should be able to operate independently of the actual luminosity sensors used. Note that the architecture is generic and can be used in a wide range of application areas where sensors needs to be connected; however, for the sake of illustration, the examples presented here are framed in a smart home setting.

By using virtualized sensors, third-party developers do not need to learn any custom sensor APIs to interact with the sensors, even though the capabilities of the sensors are limited to low-level RF communication. Assuming sensor vendors provide the sensor communication API, third-party developers can supply the necessary custom wrappers for the middleware to use, or vendors can provide such wrappers.

Virtual sensors give flexibility to applications, since replacing sensor devices does not require modifying the implementation of applications using those sensors. This is assuming the basic interaction is the same or similar. Furthermore, with technology innovation, new sensor models may natively support ZeroConf and link-local IP addressing. Applications can then use these with minimal changes, bypassing the virtual sensors.

The chapter starts with background and assumptions, and proceeds with an overview of the architecture, implementation details, and a description of the middleware protocol. We then move on to a proof of concept, describing the setup and hardware used to test the middleware, before presenting some performance results. Following this is an overview of related work, before the final conclusions, which also outlines directions for future work.

3.1 Background and Assumptions

The middleware focuses on self-configuration and offers support for developing integrated services, where multiple services can interact to offer synergies across different technologies: For instance, a light-control service could interact with the movement sensors associated with the alarm service, in addition to luminosity sensors, to decide whether the light should be switched on.

In the context of the IS-Home project, we assume a residential networked device capable of running our middleware; this could be a simple *embedded computer* running the Linux operating system, like a base station, router etc. Further, we assume the computer has multiple interconnection interfaces, e.g. ZigBee, Bluetooth, WiFi, GPRS, Ethernet and USB ports for connecting other network devices. This computer may run one or more network services, and may act as a gateway between different network applications and devices.

3.2 Architecture Overview

The middleware architecture is organized into multiple layers of abstraction to provide sensor-based services to clients. That is, physical sensors appears to behave as if they provide ZeroConf-like services. Hence, the services provided to applications become independent of the sensor hardware. The middleware takes advantage of standardized ZeroConf protocols to provide automatic network configuration of sensors and service discovery to clients. This makes the sensor services available to any ZeroConf-enabled application on the same network.

Keeping the services separated from the sensors is the most flexible solution as it allows the system to support more than one service per sensor, e.g. a single sensor unit may contain both temperature and humidity sensors. The separation of services from sensors adhere to established object-oriented principles, as it promotes high cohesion and low coupling between components. The details of a sensor's physical connection and battery status does not logically relate to the attributes of, for instance, a temperature service. For the same reasons, the communication drivers are separated from the virtual sensors and services,

as the connection details between applications and services are neither related to the logic of the sensor nor the service. Having the services separated from the sensors gives the added advantage of allowing the service component to be generic for all supported sensor types. This approach is a good match with the ZeroConf APIs, as the methods provided by these are geared toward services instead of devices.

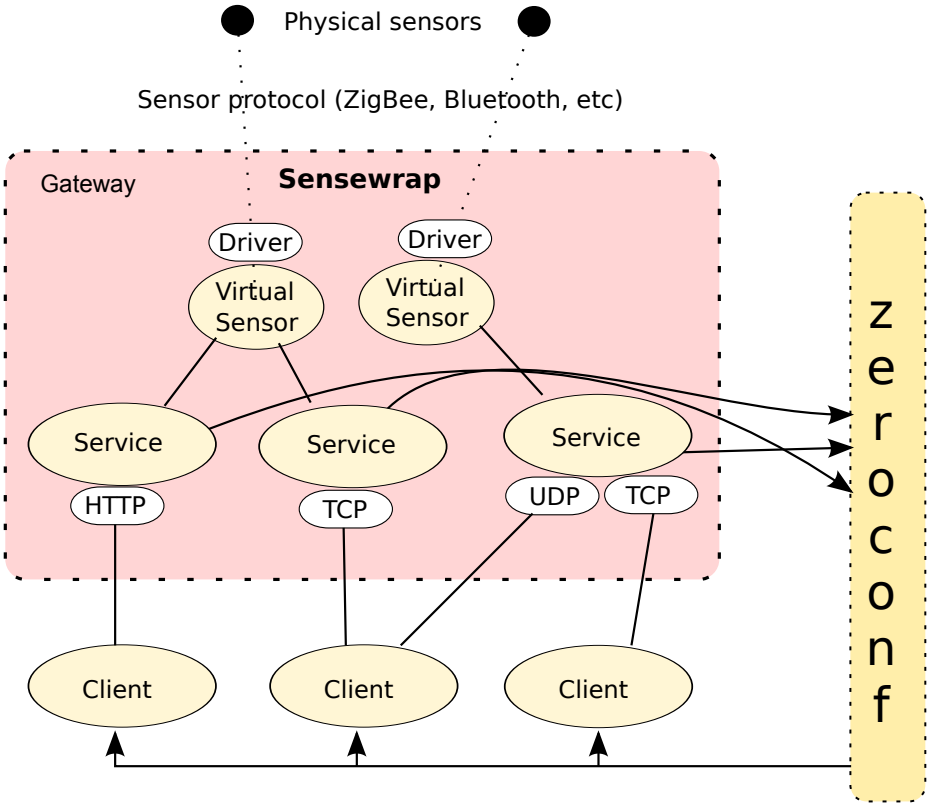

Figure 3.1: SENSEWRAP middleware architecture

Figure 3.1 show a conceptual view of the system, where the gateway hosts the main components of the middleware. Each physical sensor is represented by a corresponding virtual sensor. Furthermore, each service offered by the sensor is accessible through an instance of a *Service* interface. A virtual sensor can have many services, e.g. if the same hardware device hosts multiple sensors, the different sensor readings can be offered to applications through distinct

services. A service can also have many connections through different communication drivers. For example, multiple services for the same sensor can be registered with ZeroConf at the same time, one accessible over TCP and another over UDP. Client applications use ZeroConf to identify and locate services provided by sensors, and communicates with them through service instances in the middleware.

3.3 Implementation Details

Here, we describe the technical details of the SENSEWRAP middleware, starting with a description the main classes and interfaces. Furthermore, we describe how to add new sensors and communication protocols to the middleware. Concluding the section is an overview of the middleware protocol, describing how to interact with SENSEWRAP services.

3.3.1 Interfaces, Classes and Abstractions

Our middleware define two core abstractions: **SensorUnit** and **Service**. A *SensorUnit* is a virtual representation of the physical device hosting the actual sensors and actuators. Attributes include identity (typically a MAC address) and location. Sensor units are subclassed into sensor types such as Small Programmable Object Technology (SPOTs) [99], SquidBee [111], etc. Figure 3.2 illustrates the attributes and relationships of the *SensorUnit*-interface and its implementations in more detail.

The *SensorUnit* handles the communication between the middleware and the physical sensor, communicating directly with the sensor nodes. It translates application commands received through the communication driver and forwards these to the physical sensor, using the native communication protocol of the sensor. It is instances of this class that we refer to when using the term “virtual sensor”.

Each virtual sensor keeps track of the state of its associated physical sensor, which has the added benefit of being able to hide intermittent connection failures to the application layer, as it may cache values internally, and refresh these

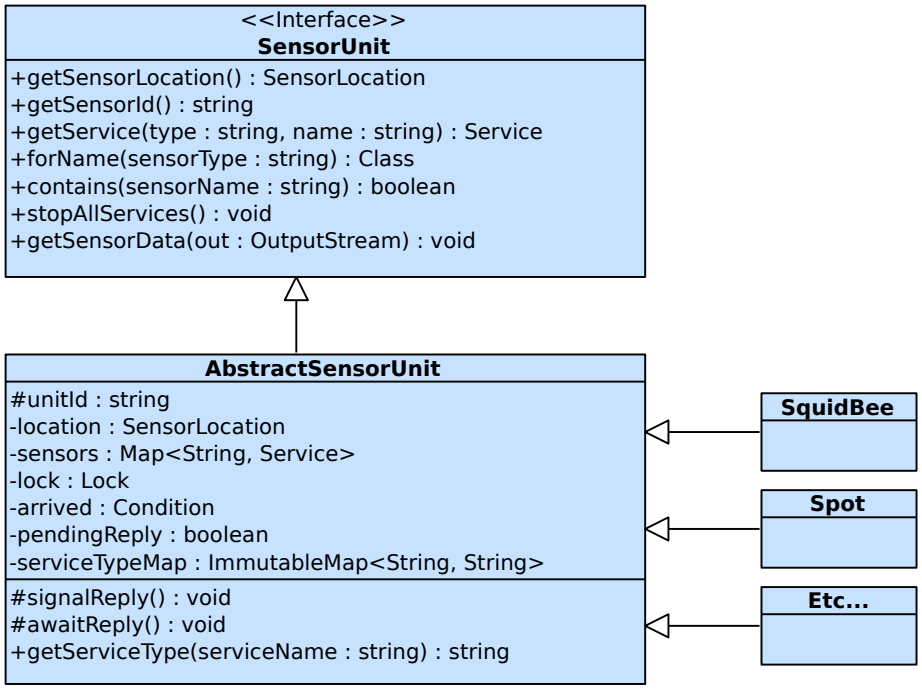

Figure 3.2: The *SensorUnit* interface and associated classes

when the connection is restored. A sensor is considered to have failed if an *IOException* is caught, e.g. due to a communications failure. If a sensor fails, the virtual sensor is responsible for *unregistering* the service from ZeroConf, removing itself from the list of sensors maintained by a *SensorFactory* instance (a factory class that will be introduced shortly), and terminate. Similarly, if an *IOException* is caught when clients are trying to access the service, the service will be unregistered from ZeroConf itself.

A *Service* is hosted on the physical unit, and can either be a sensor or an actuator. Examples of sensors includes temperature, humidity and luminosity sensors. Examples of actuators are power and light switches, thermostats and locking mechanisms. More examples of sensors and actuators are provided in Section 2.5.3 and 2.5.2. Both types are represented in the middleware through instances of the *Service* interface, which are subclassed into the classes *Sensor* and *Actuator* (Figure 3.3). These instances register the communication endpoint (host name and port number) of the service with ZeroConf and listen for con-

nection requests from clients. Upon receiving a connection request, the service instance creates a communication driver to handle the communication with the client.

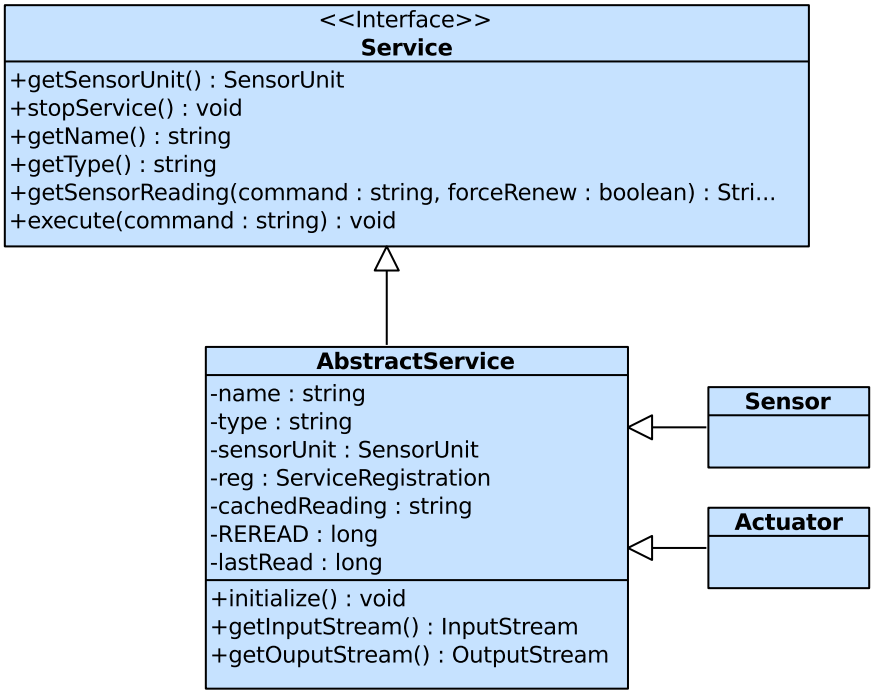

Figure 3.3: The *Service* interface and associated classes

Another important part of the SENSEWRAP middleware is the **SensorFactory** interface, illustrated in Figure 3.5. Subclasses implementing this interface listen on the network for new sensor devices, and create virtual representations of these. They also maintain a list of sensors that the middleware is capable of communicating with. The sequence diagram in Figure 3.4 illustrates how the *SensorFactory* listens for service advertisements broadcast by sensors in the network. After the service has been registered with ZeroConf, it listens for client requests on the corresponding TCP port, and spawns a *ClientHandler* thread for each connection request.

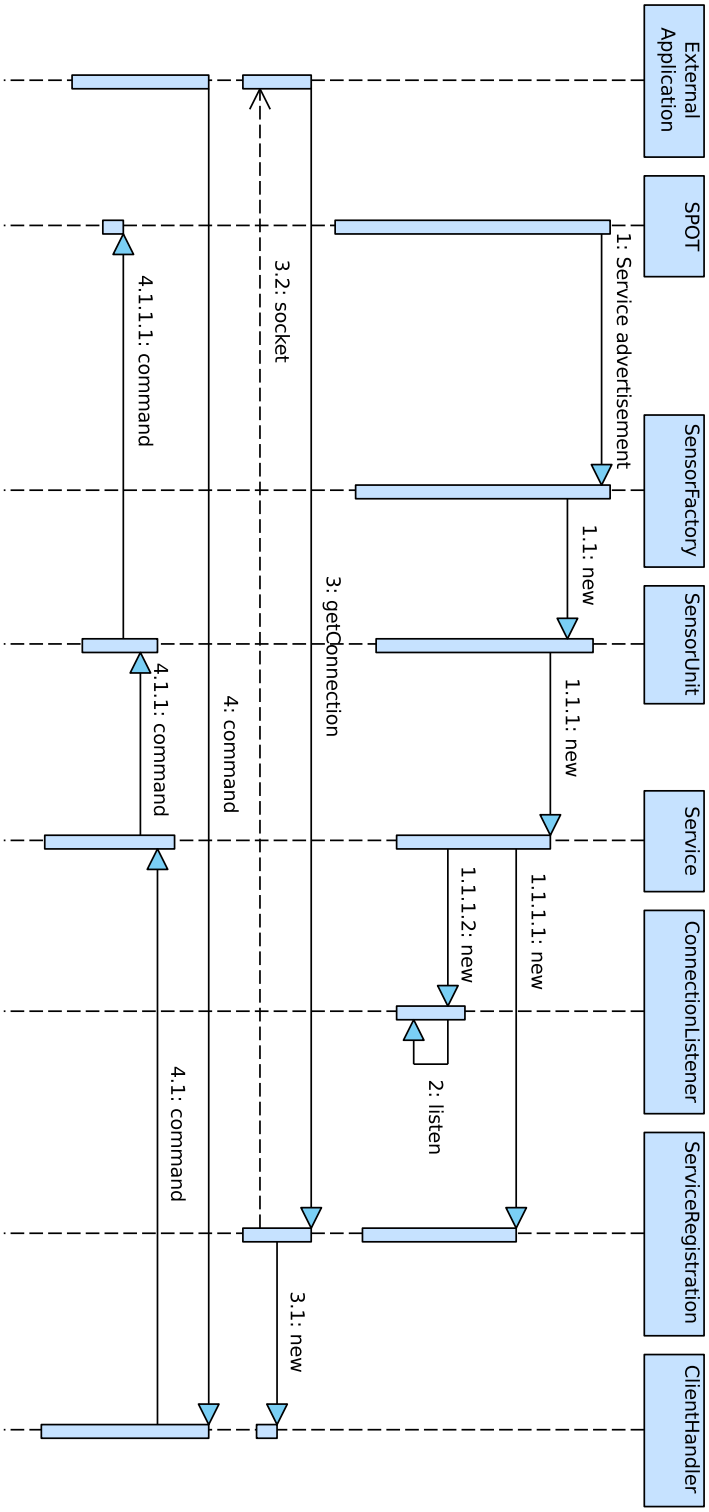

Figure 3.4: Service discovery and connection establishment

The *SensorFactory* and *SensorUnit* implementations are the only components in the SENSEWRAP middleware that needs customization to support new a new sensor device. That is, they are both comprised of a generic part, and a custom part that needs to be tailored specifically for each supported sensor type. Our current implementation have support for Sun SPOTs and SquidBee sensor types.

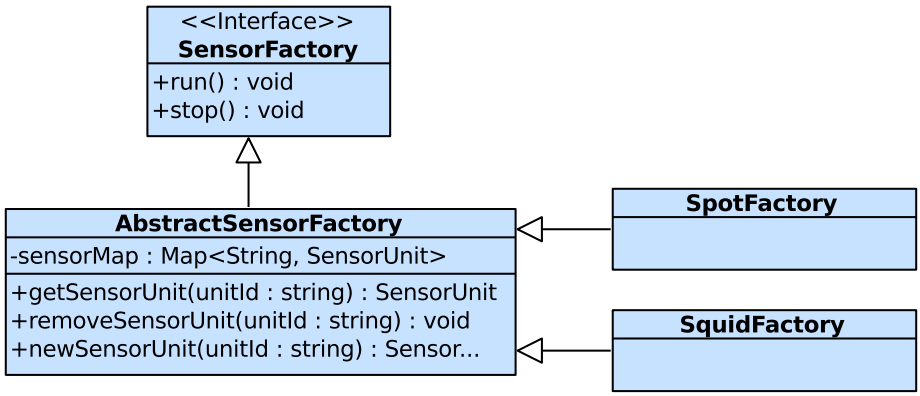

Figure 3.5: The *SensorFactory* interface and associated classes

Listing 3.1 The ServiceRegistration interface

```
1 public interface ServiceRegistration {
2   void register(String serviceName, int port)
3     throws IOException;
4   void deregister();
5 }
```

Clients can multicast a DNSSD request for available services that resides on the same network and the ZeroConf framework will reply with the name of the host where the service is running, and the port number through which to connect. An application can then send a connection request and obtain a connection in response. Commands received by the client handler is forwarded to the virtual sensor, which translates these into the appropriate sensor-specific command. These, in turn, is transmitted to the physical sensor, using the device's native communication protocol, as represented by the sensor class for the given sensor type.

3.3.2 Adding New Sensor Types

Adding support for new types of sensors involves developing device-specific subclasses of the *SensorFactory* and *SensorUnit* interfaces. In order to simplify development, the middleware includes abstract classes, implementing common parts of these interfaces, as illustrated in Figures 3.2 and 3.5. This enable implementations to reuse common functionality, effectively providing developers with a blueprint of the required classes.

Essentially, the custom part of the sensor factory needs code for detecting connection requests from the physical sensors and for creating the appropriate virtual sensor. Obviously, the virtual sensor must also be able to communicate natively with the physical sensors.

3.3.3 Adding New Communication Protocols

The communication driver is a generic communication interface through which clients connect. Different applications might require different communication protocols, and the middleware supports adding new communication drivers. Currently, a TCP-based communication driver is supported, while support for UDP, HTTP, SOAP and RMI can easily be added. Once a driver has been developed, it can be reused without modification for all sensor types supported by the middleware. In addition to making the middleware flexible, this ensures future compatibility with new protocols as they emerge.

3.3.4 SENSEWRAP Middleware Protocol

SENSEWRAP supports both the request/reply and publish/subscribe interaction models. The default interaction model is request/reply, while subscriptions can be created by appending additional parameters to the basic service call.

After a service has been looked up through ZeroConf, and connection has been established, the client applications use generic commands to communicate with the services. Table 3.1 shows the available generic commands.

For instance, the way to do a simple temperature reading would be issuing the command `GET` to the service. This would return a single reading. If the

Command	Parameters	Description
GET		Retrieves a single value
PUT		Issues a command to an actuator
SUBSCRIBE	Interval (optional) Filter (optional)	Creates a subscription, where events are pushed at the specified rate
UNSUBSCRIBE		Cancels the subscription

Table 3.1: SENSEWRAP application protocol

client wants to subscribe to the temperature service, it can ask the middleware to feed it with periodic readings by appending the keyword `SUBSCRIBE` followed by optional arguments for desired interval in milliseconds, and an optional filter string. The middleware will keep sending readings at the specified interval until it receives an `UNSUBSCRIBE` message, or until the connection is closed. If no interval or filter string is specified, it will receive all values produced by the sensor.

3.4 Proof of Concept

To demonstrate the capabilities of the SENSEWRAP middleware, a simple temperature reading application was developed. The application uses programmable sensors from Sun Microsystems, called SPOTs [99].

These are J2ME programmable sensors that come with built in temperature sensors and accelerometers. We have implemented a driver for the SPOTs devices, and implemented virtual services for the temperature sensors and light emitting diodes (actuators) hosted on these units.

Native communication is done over the 2.4GHz band, using the wireless IEEE 802.15.4 standard, which the ZigBee protocol is built on top of. Since these particular sensors are highly programmable, they can be made to respond to any command we choose. An interface that simply maps commands to a byte-based protocol we implemented on the SPOTs was developed, and is implemented by both the client class running on the physical sensor and the virtual sensor class running within the middleware. The main reason for mapping com-

mands to bytes in this manner is to improve the readability of the application code, and to conserve power when transmitting data.

In addition to the SENSEWRAP middleware, the implementation consists of the following applications:

- The **Simple Sensor Client** runs on the SPOTss and communicates wirelessly with the middleware over IEEE 802.15.4 using a base station that is connected to the host computer via USB. It is a multithreaded application that supports both broadcast and unicast over a SPOT-specific datagram protocol.

- A **Service Browser** application allows the user to browse for services and request values from these through a web browser. It uses the functions provided by Apple's Bonjour API to browse and resolve services without requiring the user to perform any network configuration. The Apache Struts presentation framework is used to generate the web pages.

The service browser application uses the *BrowseListener* interface from the ZeroConf Java API to find services on the network with very little code. The constructor:

```
new BrowseDNSSD("_ishome._tcp");
```

starts a thread that finds services of type `_ishome` that speaks TCP and keeps the application updated with any changes. When the user clicks on a service, displayed as a link on the web page, the following method call is made:

```
DNSSD.resolve(0, DNSSD.ALL_INTERFACES, name,
"_ishome._tcp", domain, this);
```

The call returns a service name, host name and port, and by using this information the *ServiceBrowser* application can create a TCP connection to the service, which is listening on the advertised port. Once connection is established, the application can issue commands and get value readouts from the physical sensor.

We also implemented an actuator test application, controlling the LEDs on the SPOTs, using the SENSEWRAP middleware platform. However, this was not used during performance tests.

3.5 Performance

Because the middleware is intended to run on a dedicated machine within the home, we do not see scalability as a big concern. Typically, the number of sensors to be handled in such an environment are limited to less than 100, and as such, the demands for scalability is not critical. However, we have performed tests on this matter to reveal potential flaws of the architecture. Response time (the time it takes for clients to receive an answer to a request) under realistic client load was measured with clients

Regardless of the scalability of the middleware itself, the number of services running on the middleware is limited by the underlying ZeroConf framework, which becomes ineffective when the number of nodes approaches 1000 [25].

Tests were performed by running the middleware on a dedicated machine, while polling it for sensor readouts from other machines on the same local network. At the most, 19 computers, hosting eight clients each, was continuously polling the middleware. The "server" had 2GB of RAM, an Intel Core Duo 2 E8300 processor and was running Fedora Core 11 with Sun's Java version 1.6_14. Measurements were obtained using the request/reply model.

3.5.1 Results

Here, we present the results from an experiment that involved loading the SENSEWRAP middleware with continuous requests issued from an increasing number of clients. This experiment allows us to observe how the latency of SENSEWRAP is affected under heavy load, and to get some measurements of the system's performance.

The latency was measured as the time elapsed between query and response from a temperature sensor on a Sun SPOT through the SENSEWRAP middleware. It was measured at the client. A caching mechanism implemented in the

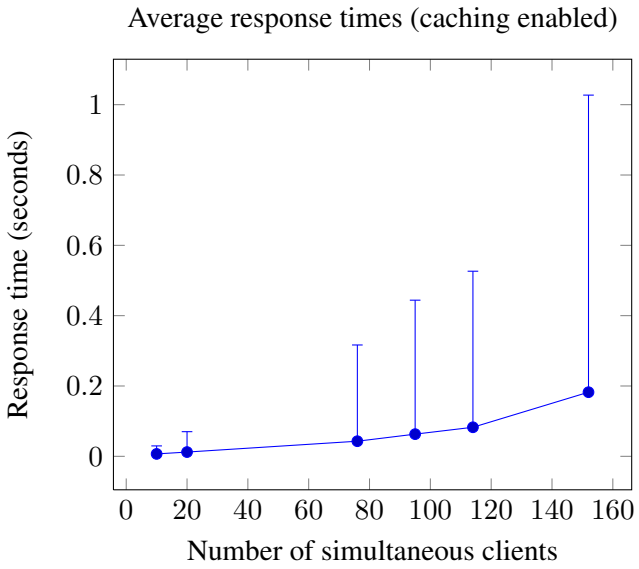

Figure 3.6: Average response times (measured at the clients)

middleware was set to reread values from the sensor only if the existing value was older than four seconds.

Figures 3.6, 3.7 and 3.8 illustrates the results of this experiment in different ways: Figure 3.6 shows the average latency per run, for six runs, with an increasing number of clients. Figures 3.7 and 3.8 shows the individual data points collected for a run of 95 simultaneous clients.

Performance were measured to an average of 6.8 milliseconds with a test run of ten simultaneous clients, each issuing 1000 requests (Figure 3.6), immediately sending a new query as soon as a reply is received. This amounts to an average capacity of handling about 147 queries per second under load. Predictably, the average response times rise as more clients are jamming the middleware with queries, and drops to a capacity of around 5.5 queries per second with 152 simultaneous clients. The error bars in Figure 3.6 illustrate the standard deviation from the average response time. As we can see from the figure, the standard deviation rises to 0.84 seconds with 152 simultaneous clients, due to a small number of outliers having high response times.

The first scatter plot (Figure 3.7) shows an excerpt of 14000 operations from

a run of 95 clients simultaneously querying the middleware a total number of 190000 times while caching of sensor readings is set to four seconds. The plot starts ten seconds into the experiment, to be sure that all clients have started, and to allow the JVM running the SENSEWRAP middleware to optimize its operation. The y-axis shows the round-trip time for each query, measured in seconds. The x-axis shows time elapsed, also measured in seconds.

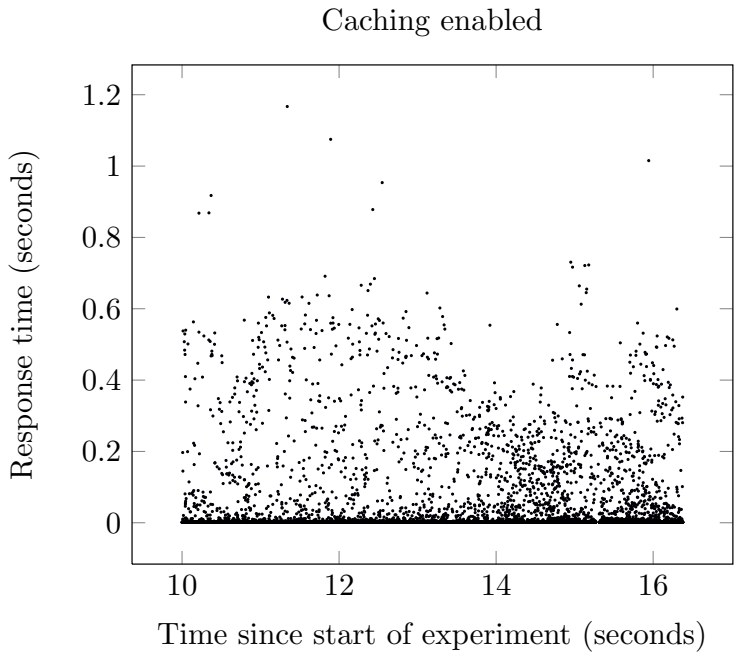

Figure 3.7: Response times for 95 simultaneous clients and 14000 requests

An observation that can be made from Figure 3.7 is that it takes only 6.38 seconds to finish 14000 operations, giving an average of 4.6 milliseconds per operation. Note that the average observed when measuring individual client operations is 6.8 milliseconds. This indicate that having the clients waiting for a response before issuing a new command does not load the middleware sufficiently to make it the performance bottleneck. In other words: the aggregated 4.6 millisecond average we observe when dividing the number of performed operations by time, suggests that parallel processing of requests are the reason behind the lower aggregated than individual average.

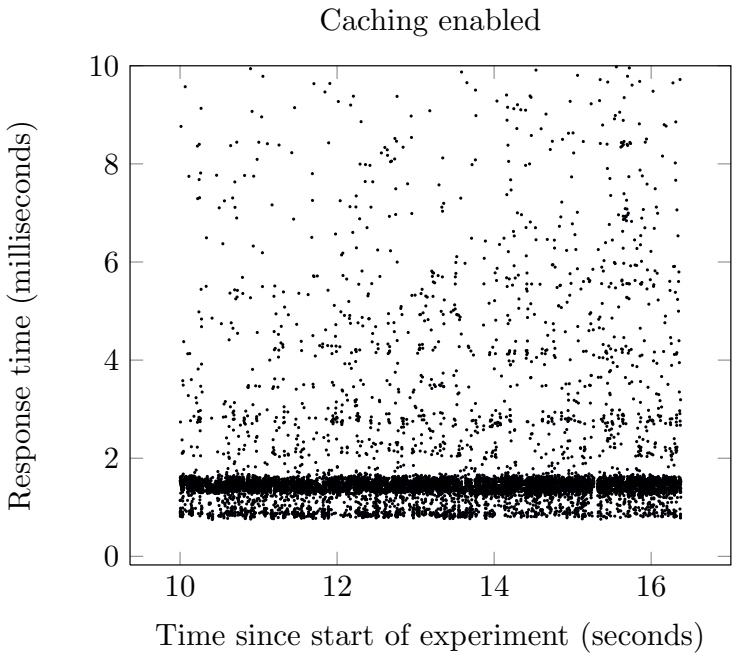

Figure 3.8: Response times under 10 ms for 95 simultaneous clients and 14000 requests

The scatter plot in Figure 3.8, was generated from the same data set as Figure 3.7, but here, response times higher than 10 milliseconds are filtered out. From this plot, we can observe that a majority of responses under heavy load are in the sub 2 millisecond-range. With this in mind, we can draw the conclusion that the 4.6 millisecond average is caused by a relatively few outliers having very high response times (the maximum observed value was 1166.959 milliseconds). These outliers also explain the high standard deviations of Figure 3.6.

3.5.2 Evaluation

In a smart home scenario, the middleware is likely to run on less powerful hardware than what was used in our tests, but our rationale is that even if one divide the performance by ten, it is still more than sufficient to handle the requirements of a typical smart home. We also measured the execution time for each renewal

of the cache at the server. A total of 56 sensor readings had an average value of 951.57 ms, which would give the middleware a capacity of just over one reading per second per sensor, thus illustrates the performance gain of caching.

3.6 Related Work

In previous work, **Construct** [44] offers a distributed middleware for pervasive systems and provides mechanisms for capturing sensor data and converting them into Resource Description Framework (RDF) formatted data for storage. Like SENSEWRAP, Construct employs ZeroConf to locate services, but does not allow discovery of sensor devices as in our middleware. While our focus is on finding a standardized way for applications to communicate with sensors, the main focus of Construct appears to be on data capture and the processing of information.

Hourglass [134] is an infrastructure for connecting sensor networks to applications. It provides a *data collection network*, that aggregate functionality from several disparate sensor networks, and offer this to Internet-based applications. Compared to our architecture, Hourglass focuses on the underlying network links and data streams more than the service aspect. The main effort is on handling unreliable connectivity by providing links between networks and applications that buffers data and retransmits these at a later point in cases of link loss. Neither Hourglass or **Global Sensor Network (GSN)** [1] focus on service discovery. Like our own middleware, GSN aims to solve the problem of hardware heterogeneity in sensor networks. GSN also use *adapters* to abstract physical devices into virtual sensors. With SENSEWRAP, we take the abstraction one step further by virtualizing the services as well. With GSN the emphasis is to provide the ability to query all supported sensors using SQL, and to provide a homogeneous view of sensor data.

Open Services Gateway initiative (OSGi) provides a gateway for connecting different devices and services together through a central point, allowing applications to be composed from different, reusable service modules [43]. The framework is module based and only specifies the application programming in-

terface, not the underlying implementation, leaving it up to the developers to handle the actual communication with the sensors or actuators.

Using OSGi as a foundation, Gürgen et al take a "database approach" in their **SStreaMWare** middleware [67], offering a schema to represent sensor data in a generic manner. Interaction with the sensors is performed with declarative queries in a SQL-like relational language. Like SENSEWRAP, SStreaMWare uses *adapters* to transform generic commands into the necessary device-specific format, and it also provide both publish/subscribe and request/reply communication models. However, the scope of SStreaMware is quite different from SENSEWRAP, as SStreaMWare comes as a complete package, where sensor interaction is performed via a provided graphical user interface and not at application level. This makes the system difficult to adapt to third party applications, which it is clearly not intended for. The scope of our middleware is to facilitate integration between sensors and applications with minimal effort. Our approach to virtualizing sensors based on ZeroConf, using *communication drivers* to interface with applications is more lightweight and allows better application level adaptation.

Tenet [63] is more of a network architecture than middleware, dividing sensor networks into tiers, consisting of masters and motes. The argument for this architecture is that sensor motes are unreliable and underpowered, hence all but the simplest computing tasks are better left to more powerful master nodes. Furthermore, the authors claims that software re-usability is enhanced by having most of the application logic on master nodes, as device specific customization of the code is less likely to be needed. This is not unlike our approach, but instead of several masters, we use a single gateway to perform the heavy lifting in terms of computational tasks. The reason for not using several masters is simply that we don't see the need for more in a private smart home, although it would be relatively easy to include additional gateways if required (one way of achieving that would be to set up an additional gateways to listen for different types of services).

The **Hydra** framework [61] provides much of the same functionality as SENSEWRAP, and is built upon many of the same concepts, with virtualiza-

tion of hardware resources as the fundamental idea. It is also more mature than our solution. However, we argue that it is fundamentally flawed as a scalable middleware for sensor networks in a number of ways; For one, it uses "Big" Web Services to expose sensor services, which means that the sensor network must pass around heavyweight SOAP objects in order to use the services. Additionally, it uses the UDDI protocol for service discovery, a protocol which we in Section 2.6.1 argue is far to chatty for efficient service discovery in a sensor network. Additionally, it uses the obsoleted JXTA [118] protocol for P2P communication between sensors.

3.7 Conclusions and Future Work

By virtualizing the physical sensors in smart homes, we can provide client applications with a uniform communication interface. We have demonstrated how the important task of automating the discovery of services and devices as well as the networking between applications can be solved using ZeroConf.

While the middleware presented here makes the communication protocol between sensors and application generic, the application protocol is not. A logical next step would be to expand the SENSEWRAP middleware application to include support for other types of sensors beyond the Sun SPOTs supported in the current implementation.

Enabling remote access to the services in the home over wide area networks such as the Internet or GPRS can be useful for tasks like adjusting the heat before coming home, or turning off the alarm to let someone in. Remote accessibility brings up some security and privacy concerns that needs to be addressed at some point.

Having multiple higher-level applications competing for resources (actuators) introduces the issue of resource ownership and dependency management. For instance, two applications accessing the same actuators could potentially result in conflicts where one of them is constantly turning a switch off, while the other turns it back on. A priority concept, like the one outlined by Retkowitz and Kulle [125] could be worth looking into in future versions.

Chapter 4

EVENTCASTER: A Platform for Stateful Event Processing

This chapter introduces the EVENTCASTER platform, a general-purpose event processing platform that is built on the principles discussed in Section 2.5, addressing the EVENT PROCESSING ARCHITECTURE challenge. That is to provide a general architecture for the efficient processing of high volumes of events. EVENTCASTER is the underlying platform on which the implementations presented in Chapters 6 and 7 are built upon.

Section 4.1 provides a high-level overview of the EVENTCASTER software architecture, while Section 4.2 gives some implementation details and introduces the underlying technologies and the reasons for choosing these. Configuration of the system is described in Section 4.3, while deployment requirements and options are discussed in Section 4.4. Section 4.5 concludes the chapter with a brief summary.

4.1 Architectural Overview

The EVENTCASTER plaform follows the Event-Driven Architecture (EDA) paradigm introduced in Section 2.5.1. To reiterate; the main building blocks of an EDA are event *producers* and *consumers*, *Event Processing Agents (EPAs)* and *channels*, as illustrated in Figure 2.9. Event producers are software that

introduce raw events to an Event Processing Network (EPN). An EPN is composed from EPAs connected by event channels. After being processed by EPAs, events are distributed from the EPN to event consumers, which represent the endpoints in event-driven interactions. Event producers and consumers are located outside of the EPN.

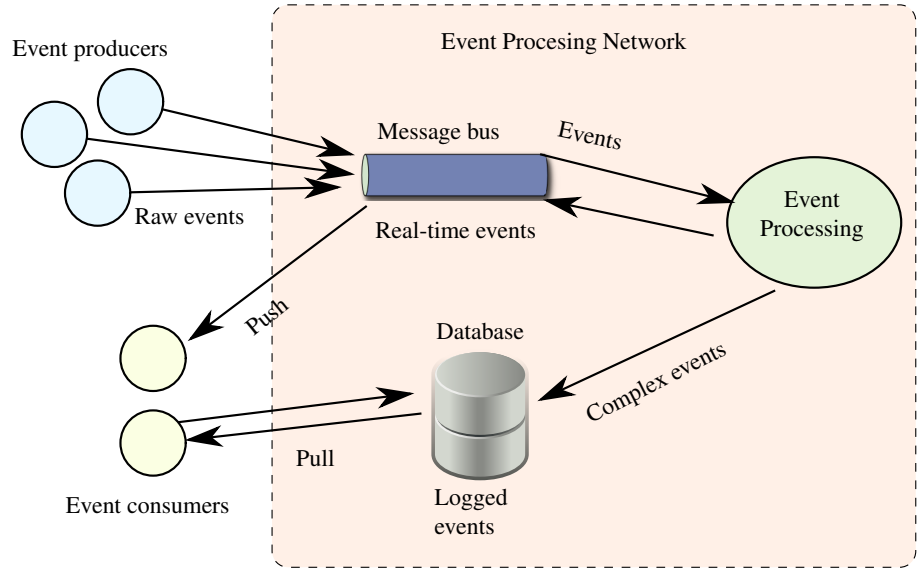

Figure 4.1: EVENTCASTER general architecture

Figure 4.1 shows the EVENTCASTER platform from a birds-eye perspective. In addition to the event producers and consumers, it includes a message bus, representing event channels. The message bus connects producers to EPAs, EPAs to EPAs, and EPAs to consumers. The EPAs are located within the event processing applications, represented by the green ellipse in Figure 4.1. EPAs are pieces of code that performs event processing in some form, including filtering and transformation. Additionally, it includes a database, which is not part of the EDA model presented in Section 2.5.1. It is used in the EVENTCASTER architecture to store derived and aggregated events, as well as snapshots of application state.

The EVENTCASTER platform facilitates general event processing by combining our own Java interfaces with XML, Java 2 Enterprise Edition (J2EE)

technologies and the Esper CEP engine. The Esper engine is programmed using the declarative EPL language, which builds on the SQL-92 syntax. An overview of Esper and EPL is provided in Section 4.2.1.

4.1.1 Package Organization

Here, a brief overview of the software packages included in the EVENTCASTER middleware are provided, along with a description of their role in the system. Figure 4.2 shows the relationships between the packages.

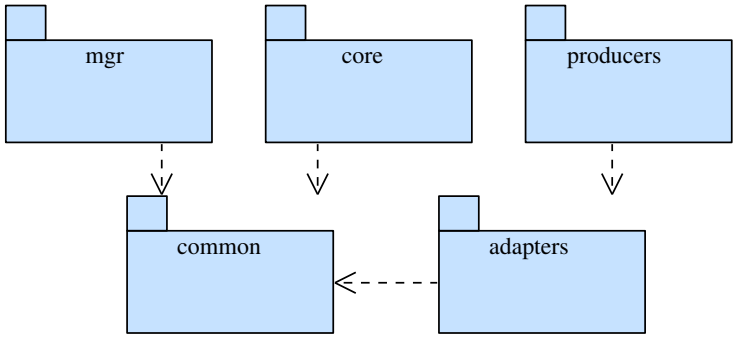

Figure 4.2: EVENTCASTER package organization

- **Common**: As the name implies, this package contains the common interfaces and utility methods used by all other packages in the EVENTCASTER middleware.
- **Core**: The core package contains the Esper event processing engine, and hosts the EPAs, in the form of EPL statements as well as EPAs written in Java. The core package is represented with a green ellipse in Figure 4.1.
- **Producers**: Events arrive over various protocols, and are piped onto the message queue by producers, which are standalone clients for handling events and putting them on a message queue. Producers are represented by the blue circles in Figure 4.1.
- **Adapters**: Adapters are used by producers to fetch data at the protocol-level. Currently, this package includes adapters for receiving data over

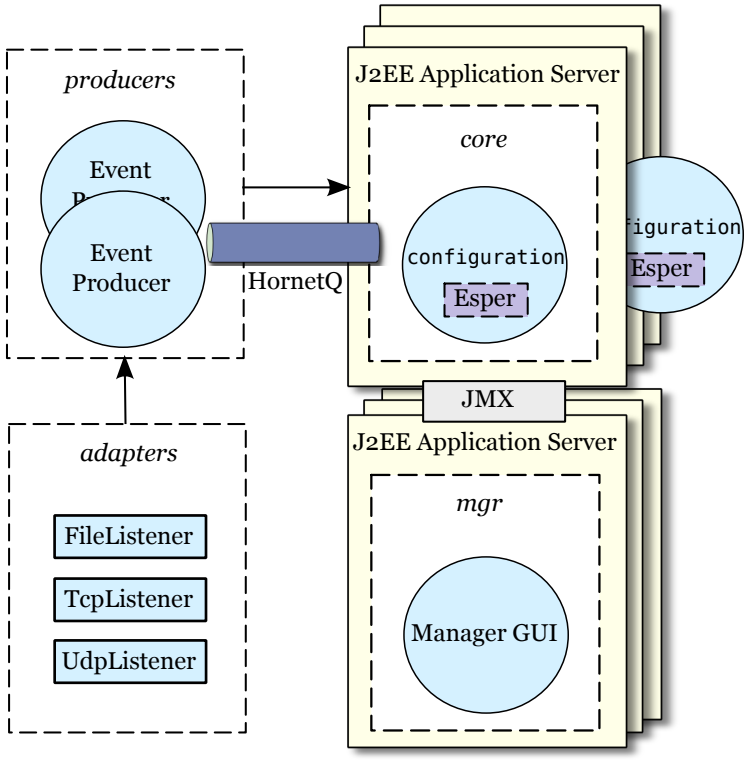

Figure 4.3: Overview of the EventCaster package structure and components

UDP and TCP, as well as tailing log files and monitoring folders for the arrival of new files.

- **Mgr**: The mgr package contains the EVENTCASTER manager web application, from which a screenshot is displayed in Figure 4.6, which enables users to install and remove EPL queries at runtime.

Figure 4.3 illustrates another view of the EVENTCASTER platform. The software packages described above are denoted with dotted lines, application instances are blue, and application server instances are denoted with yellow squares. In the lower left part, we have the *adapters*, which are general protocol adapters, used by *producers* to receive data.

The *core* package is deployed on several instances of the JBoss Application Server. When deployed, it is packaged as an Enterprise Archive (EAR), with

all instances being identical, except the main configuration file, which resides outside of the EAR package. This configuration file contains the EPL statements (EPAs), and defines the wiring between the message queues and EPAs, as well as between EPAs and event publishers. It is covered in detail in Section 4.3.

The primary interaction model of EVENTCASTER is publish/subscribe, which allows event producers and consumers to be changed without affecting other parts of the system, and at the same time eliminates some of the latency and processing overhead that typically comes with request/reply interactions. Management of the event processing engine (*mgr*), however, is exposed over a request/reply interface, due to the synchronous nature of applying configuration changes through a web-based user interface.

In addition to the push interface of the MOM for subscribing to live data, persisted historical data can be pulled from a relational database (Figure 4.1).

4.2 Implementation

This section presents some implementation details, including the underlying technologies used to build the platform. We start with describing the flow of events through the system, and a discussion on how EVENTCASTER applications are composed.

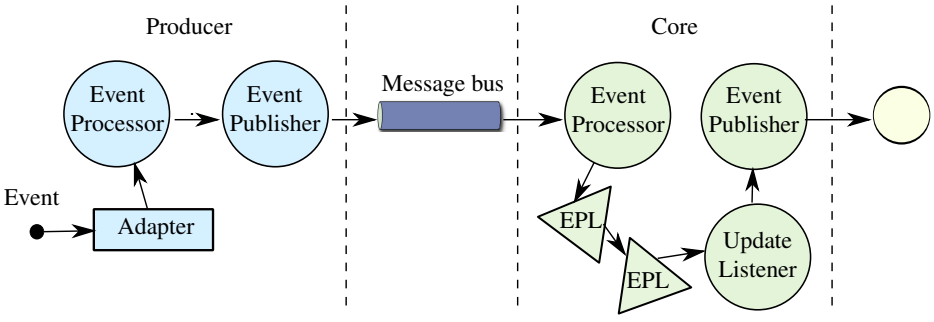

Figure 4.4: EVENTCASTER event flow

Essential to the implementation of the EVENTCASTER middleware is a pair of interfaces, named *EventProcessor* and *EventPublisher*, presented in Listings 4.1

and 4.2. Figure 4.4 illustrates the flow of events through the middleware, and shows how events transmitted over various protocols are received by *adapters* before being passed on to *EventProcessor* instances in the *producers* package. These instances perform simple event processing such as transformation and error-checking on the events, before introducing them to the EPN through an *EventPublisher*.

Listing 4.1 The EventProcessor interface

```
1 /**
2  * @param <T> class of events to process
3  */
4 public interface EventProcessor<T> {
5   /**
6    * @param event object to process
7    */
8   public void processEvent(T event);
9 }
```

EventProcessors are also used in the *core* application, where they convert events from various formats into Java objects before sending them to the Esper runtime. Here, they are processed by one or more EPAs, represented by EPL statements.

Access to the output of an EPL statement in Java code requires the registration of an *UpdateListener* with the statement. The *UpdateListener*, presented in Listing 4.3, is an Esper interface containing an `update()` method that is called whenever the EPL statement outputs events.

The *EventPublisher* interface are used both by *producers* to publish events to the message bus, and within the *core* application for publishing events received by an *UpdateListener* to external consumers, represented by the yellow circle in Figure 4.4. *EventPublishers* may also publish events back to the message queue, for further processing by other *core* application instances. However, this alternative flow of events was omitted from the illustration in order to preserve clarity.

Furthermore, an *UpdateListener* may also perform event processing in Java, before introducing a modified or derived event to the Esper engine, without publishing it to external consumers. This alternative event flow was also omitted

from Figure 4.4 to avoid clutter.

As can be seen from Figure 4.4, an EVENTCASTER *producer* is composed of three objects: An *Adapter*, an *EventProcessor* and an *EventPublisher*. Of these three components, only the *EventProcessor* needs to be developed in order to create a new producer, provided that an adapter for the underlying communication protocol already exists. Writing an adapter of this type is a one-time task, as it may be reused for all new producers receiving events over the same protocol.

Listing 4.2 The EventPublisher interface

```
1  /**
2   * @param <T> class of events to publish
3   */
4  public interface EventPublisher<T> {
5    /**
6     * @param event object to publish
7     */
8    void publishEvent(T event);
9    /**
10    * @return address of the published events
11    */
12   String getDestination();
13 }
```

Listing 4.3 The UpdateListener interface from the Esper distribution. Some comments and parts of the Javadoc have been removed for the sake of brevity

```
1  public interface UpdateListener {
2    /**
3     * @param newEvents is any new events. This will be
4     *  null or empty if the update is for old events only.
5     * @param oldEvents is any old events. This will be
6     *  null or empty if the update is for new events only.
7     */
8    public void update(EventBean[] newEvents,
9                       EventBean[] oldEvents);
10 }
```

With the EVENTCASTER middleware, a complete event processing application may be implemented by writing one or more *EventProcessors*, and setting up the appropriate channel connections in the main configuration file.

4.2.1 Underlying Technologies

Here, the technological building blocks of the EVENTCASTER platform is introduced. Because the platform is deployed in an industrial environment, as mentioned in Chapter 1, research prototypes were not considered.

We start off by presenting the message bus, which provides the *event channels* of the platform, before introducing the application server that hosts the *core* application. A J2EE application server comes with useful abstractions for handling things like database connections, access control and application management. Furthermore, it opens up possibilities for high availability through clustering and automatic failover [137]. Even though setting up high availability is fully achievable in other ways, much of the required functionality for this is already included with most application servers.

The section is concluded with a presentation of the event processing engine. Using a specialized event processing language gives us access to some useful abstractions that simplifies the development of more advanced event processing applications. The benefits and drawbacks of this approach are discussed in detail in Chapter 6.

HornetQ Message Bus

HornetQ [75] was chosen as the distribution layer for events due to its high performance and ease of use. It comes with the usual functionality included in MOM, such as filtering, transformation, persistence and delivery guarantees, as presented in Section 2.4.2.

Getting started with HornetQ was fairly simple, and simply a matter of downloading the distribution, making some small changes to the main configuration file and executing the startup script.

Interacting with the HornetQ server was straightforward as well, and required only the inclusion of two JAR libraries on the client side (three, if one needs to use the JMS overlay). Even though the HornetQ distribution includes a JMS overlay, we opted to use HornetQ's native *core* API instead. The core API only offers two abstractions; *queue* and *address*, and the documentation sug-

gests that one can build any interaction included in the JMS specification from these.

There was, however, a learning curve regarding the behavior of the middleware, most notably in the behavior of message acknowledgments and having messages removed from the server after client delivery, and implementing the *publish/subscribe* interaction pattern.

HornetQ provides a JMS API in the form of an overlay library that can be used instead of the native API. HornetQ's JMS api does not offer any functionality that cannot be achieved with the native API. However, some of the challenges in understanding the behaviour and interactions of the HornetQ MOM, could probably have been avoided by using the JMS API, as it offers a more intuitive implementation of messaging and *publish/subscribe* interactions. The reasons for going with the native API was that it meant one less JAR to depend on and the documentation suggesting slightly better performance and a simpler abstraction. It seems clear now, however, that using the JMS overlay would have been the easier route to take, at least for the *publish/subscribe* interactions.

HornetQ offers excellent performance, and its STOMP interface has proved a convenient way of providing push-style interactions to non-Java client devices.

Events are distributed on multiple queues, according to event type, which allows for greater flexibility and cleaner code on the consumer part. That is, there is no need to set up a filter for extracting different event types. In the current deployment, the 5 minute average CPU load typically hovers around 1% for the HornetQ server during normal operation.

JBoss Application Server

HornetQ is built and maintained by the JBoss community, and the JBoss application server [100] comes with HornetQ support out of the box, and as such, was a natural choice of application server.

The use of an application server allows for the convenience of packaging the *core* application together with all its dependencies in a single EAR, deployable to the application server. The building and packaging of EARs can be automated using a build tool like Maven [92], and is an elegant way of handling

dependencies.

The JBoss application server also handles database connections, and includes ready-made functionality for connection pooling. Furthermore, it facilitates redundancy through automatic failover, as well as load-balancing across multiple instances. However, the load balancing part in particular is not very straightforward to implement, and as such, was not used in the EVENTCASTER system.

With this in mind, it is not a given that the benefits outweighs the added complexity of running the EVENTCASTER *core* component in an application server, and it is something that may well be reconsidered at a later stage.

Event Processing Language

Here we briefly survey the capabilities of the EPL [46] language and its runtime environment, the Esper event processing engine. Esper provides an open-source implementation of its processing engine, and the necessary Java libraries for interacting with it. Our choice of Esper and EPL is primarily motivated by its focus on stream processing.

EPL is a declarative query language derived from SQL; it shares much of its syntax and functionality with SQL, such as select, insert, update, and aggregation functions for summation, averaging, and join operators. However, instead of operating on relational database tables, EPL operates on streams of data. Using these operators, one can construct a wide variety of online queries used to process data from event streams, such as the stream of channel changes (zaps) from customer STBs. An EPL query will process one or more event streams, looking for event patterns that match the query, and produce an output event.

Moreover, since streams are continuous, i.e. not temporally restricted, EPL introduces a sliding window concept to be able to construct queries that operate over limited, but sliding time intervals. This is a very useful construct that was introduced in Section 2.5.5, and are used extensively in the applications presented in Chapters 6 and 7. Figure 4.5 illustrates the sliding time-window concept. In this example where the window only keeps the events received in the last 10 seconds.

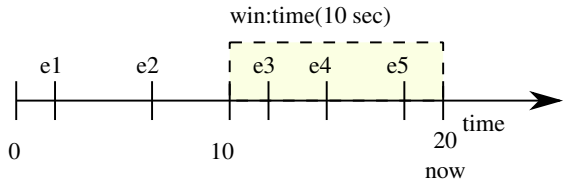

Figure 4.5: Sliding time-window

Esper can handle events represented in a variety of ways, e.g., as Java or C# objects that provide getter and setter methods to access its attributes, and is the approach used in the EVENTCASTER platform. Events can also be represented by Java objects that implement the `java.util.Map` interface, using the map's get method to access the attributes. Event objects in Esper are assumed to be immutable.

Deploying an Esper server typically involves the following steps:

1. Start the Esper processing engine.
2. Install EPL queries.
3. Establish subscriptions by registering *UpdateListeners* with the Esper processing engine. These subscriptions are connected with the installed queries, acting as handlers for output events generated by the queries.
4. Construct Java objects to be passed to the processing engine to be processed by the previously installed queries.
5. Receive and parse events from the data stream.

4.3 Configuration

To ease the deployment of EVENTCASTER applications, the EPL statements and the wiring of these to other components of the platform are kept in a configuration file, separated from the rest of the application.

We now describe how to configure an EVENTCASTER deployment, giving special attention to the main configuration file, that controls the EPL statements, and the general operation of the application. The configuration file defines the

set of EPL queries to install and which event processors to load at startup. Listing 4.5 shows a complete example configuration. We explain its structure below.

Changes to the configuration can be made in several ways: One approach is to use the Java Management Extensions (JMX) interface accessible through the web interface of the application server or the JConsole tool bundled with the Java SDK. JMX is a management protocol for Java applications. Alternatively, the *mgr* interface, shown in Figure 4.6, enables the installation and wiring of EPL statements to *EventPublishers* at runtime through a web browser, as well as providing an overview of the currently deployed EVENTCASTER configuration.

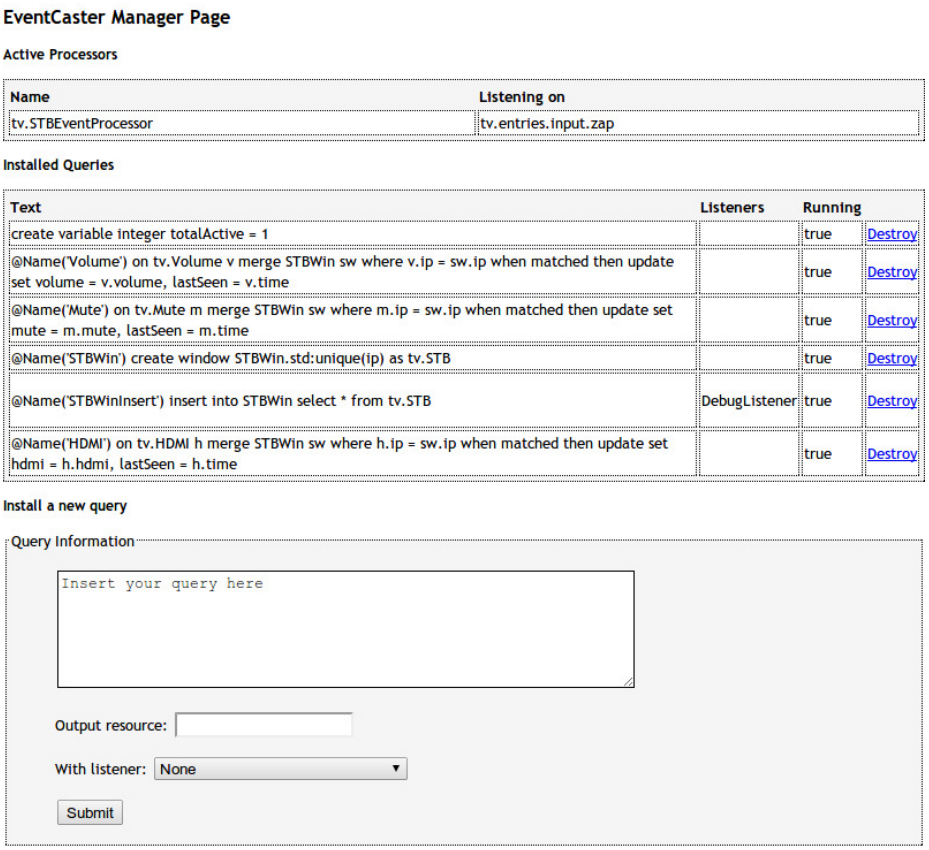

Figure 4.6: EventCaster Manager GUI

Another approach is to modify the configuration file Figure 4.5 directly. This is currently the only way to make persistent changes to the configuration of

an EVENTCASTER deployment. Because it is loaded on startup, the latter approach requires a restart of the application server for the changes to be loaded. Listing 4.5 is from the configuration of an EVENTCASTER instance, and shows how publishers are connected to EPL queries. The structure of the file is:

- `external-hq-server`: The URL of the message broker.
- `processors`: List of Java processors to be loaded at startup. Processors subscribe to message topics, process the messages, and sends them to the Esper processing engine. The `input-resource` attribute tells the processor where to look for messages, e.g. a HornetQ queue name.
- `epqueries`: List of EPL queries in the given configuration. The output of these may be directed to other queries for additional processing, or to custom-made Java update listeners. Listeners perform functions such as persisting state snapshots to a database, or publishing derived events back onto the message bus. An `epquery` represent an EPA in the *core* application. A `@Name` attribute at the start of the statement can be used as a key to look up and modify the execution status (pause, destroy, etc) of statements after they have been installed in the Esper engine.
- Optional `listeners` may be attached to an `epquery`. These are Java classes that implements the *UpdateListener* interface from Esper, and are loaded dynamically at using startup, using reflection to instantiate a class object from the class name, specified in the `name` attribute of the `listener` entry (see Listing 4.5 for examples). As mentioned in Section 4.2, an *UpdateListener* receives the events emitted by the EPL query it is attached to.
- The optional `output-resource` attribute of a `listener` is used if the *UpdateListener* forwards its received events to an *EventPublisher*, as illustrated in Figure 4.4, and specifies the destination of the events. The destination can be anything from a text file to a database table.
- `statehandler` contains two attributes: `name` and `stmts-to-isolate`. The `name` attribute specifies the name of the class responsible

for restoring state on startup, and persisting state on shutdown. `stmts-to-isolate` is a list of EPL statements to suspend when restoring state on startup to ensure that the ordering of events is correct. This is necessary because EPL statements started before state is restored, introduces the risk of live events being superseded by older events, retrieved from persistent storage.

Listing 4.4 EVENTCASTER Processor configuration

```
<!--List of processors to be loaded at startup-->
<processors>
  <processor>
    <name>tv.ChannelStatProcessor</name>
    <input-resource>lvq.tv.stats.viewerstats</input-resource>
    <enabled>true</enabled>
  </processor>
</processors>
```

Listing 4.4 is an excerpt from the configuration of the ADSCORER system, presented in Chapter 7, and illustrates how an *EventProcessor* instance are connected to a message queue.

Listing 4.5 EVENTCASTER Example configuration

```
<configuration>
 <external-hq-server><!--Address of HornetQ server-->
  <uri>tvmq1</uri>
 </external-hq-server>
 <processors><!--List of processors to be loaded-->
  <processor>
   <name>tv.STBEventProcessor</name>
   <input-resource>tv.entries.input.zap</input-resource>
  </processor>
 </processors>
 <epqueries>
  <epquery><!--Window containing the state of STBs-->
   <statement>
    @Name('STBWin')
    create window STBWin.std:unique(ip) as tv.STB
   </statement>
  </epquery>
  <epquery><!--Decorating channelstats with percentage-->
   <statement>
    @Name('ChannelStatPublish')
    select cs.*, percent(cs.viewers, totalActive) as share,
    current_timestamp as time,
    (select count(*) from STBWin(mute = true)
     where channel = cs.channel) as muted
    from ChannelStats cs
   </statement>
   <listeners>
    <listener>
     <name>tv.TvStatDataMiner</name>
     <output-resource>viewer_stats</output-resource>
    </listener>
    <listener>
     <name>tv.TvStatSnapshotPublisher</name>
     <output-resource>lvq.tv.stats.viewers</output-resource>
    </listener>
   </listeners>
  </epquery>
 </epqueries>
 <statehandler>
  <name>StateHandlerViewStats</name>
  <stmts-to-isolate>
   <name>STBWin</name>
  </stmts-to-isolate>
 </statehandler>
</configuration>
```

4.4 Deployment

Due to its modular and flexible architecture, a complete EVENTCASTER application may be deployed on either a single host, or on multiple hosts.

For optimal performance, it is recommended that a dedicated host is assigned for each *producer*, an additional host for the message bus, as well as to one or more hosts assigned to the *core* application. These requirements depends on the complexity and size of the event processing task at hand.

Like the *core* application, the *mgr* application also runs on the JBoss J2EE server, and can either be deployed on the same JBoss instance as the *core* application, or on a separate server. Since the manager application is not resource intensive, it makes sense to install it on the same server as the *core* application.

If the application requires persistent logging of events, it is recommended to use a database, which should be assigned a dedicated host as well. A relational database is the most likely component of an EVENTCASTER application to become a bottleneck. Consequently, the structure of the database schema along with the rate of writes must be carefully considered.

Figure 4.7 illustrates an example of how the EVENTCASTER system can be distributed among multiple hosts. The following subsection describes the network setup in detail.

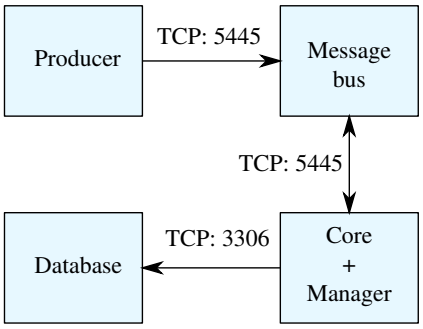

Figure 4.7: Recommended minimal deployment

4.4.1 Network Setup

The message queues on the HornetQ server are exposed over several APIs that all use TCP for transport. Java clients may use either the native HornetQ API, or a JMS API over the same TCP port, which is configurable on the server, and defaults to 5445.

For the EVENTCASTER system, we only use the native HornetQ API for the producers, meaning that a single open TCP port is sufficient for the communication between producers and the message bus. This also applies for the communication between the message bus and the *core* application.

Event consumers written in Java also subscribe to the message bus using the same port. Non-Java consumers may use the STOMP protocol, and web applications can use STOMP over WebSockets, which enable push interactions directly to a web browser. STOMP is a platform independent text-based protocol for publish/subscribe interactions that runs over TCP. STOMP and STOMP over WebSockets require an additional TCP port each. If a database is included in the system, it requires an open TCP port from the *core* application to the database server as well.

4.5 Summary

In this chapter, we have introduced the EVENTCASTER general-purpose event processing middleware, an industry-ready platform built from state-of-the-art middleware technologies combined with a custom-written layer of application logic. The key components of the middleware are the *producers*, *adapters* and the *core* software packages. Event processing is performed within the *core* application, using the Esper CEP engine, and events are distributed over the HornetQ message bus.

The components of the EVENTCASTER middleware are loosely coupled, which allows them to be distributed across multiple servers. This aids scalability and flexibility of deployment.

The *core* application can be managed via a web browser interface, or via a configuration file. This eases the deployment of EVENTCASTER applications,

as changes in configuration may be performed at runtime, eliminating the need for recompilation and redeployment, which would have been necessary if the EPL statements and their wiring to other components were hardcoded.

The EVENTCASTER middleware enables developers to focus on implementing the logic of the actual event processing task at hand, instead of details unrelated to the primary goal, such as integration tasks and system architecture.

Chapter 5

Television Viewership Ratings

The current approach used to obtain official television channel statistics is based on surveys combined with specialized reporting hardware. These are deployed only on a small scale and batch processed every 24 hours. With the enhanced capabilities of present-day STBs for Internet Protocol Television (IPTV), network operators can track channel popularity and usage patterns with a degree of precision and sophistication not possible with existing methods.

Altibox has more than 320,000 STBs deployed in their customer network, and as such, provides us with an ideal scenario for the analysis of viewer behaviour. By using event processing technologies, this analysis can be performed in near real-time. Additionally, it is also an excellent use case for performing an event processing paradigm comparison.

In Chapter 6, we develop a viewership statistics application in two distinct programming and event processing paradigms, comparing the performance and complexity of the resulting applications.

We build on this application to develop another novel application for scoring advertisements in Chapter 7. The AdScorer application scores the performance of televised advertisements in near real-time, presenting a detailed evaluation on a per-advertisement basis within a second after the advertisement has finished.

While the previous chapter gave a technical overview of the underlying EventCaster platform, which the event processing applications presented

in Chapters 6 and 7 are built upon, this chapter provides background and motivation for this work.

We begin by describing the current state-of-the-art in measuring viewership and program rating, focusing on the Norwegian television market. This initial section also covers a brief historical overview of the measurement methods used, and argues why these needs to change. Section 5.2 describes the deployment scenario of the work presented in Chapters 6 and 7, while Section 5.3 covers related work. Finally, some thoughts on the future of the media measurement industry is presented in Section 5.4. A summary concludes this chapter.

5.1 The Current State of Media Measurement

The media measurement industry is in turmoil, with the old prediction-based models being challenged by more accurate measurement-based techniques, based on actual viewer behaviour drawn from much larger sample selections. Currently, media are divided between online and offline media, where media distributed on the internet are categorized as online, while media distributed on the traditional channels, such as print, radio and broadcast television, are categorized as offline. The consumption of media distributed on the internet can be measured with great precision, and in near real-time, which is not the case with offline media.

As measurement methods converge across different types of media, the online versus offline measurement divide will diminish. Television is one such medium that has traditionally required offline measurements because of its inherent one-way communication style. Dorai et al. [45] predicts that the online versus offline division we have today will soon be replaced by measured versus unmeasured, as measurement methods converges between different types of media. This view is shared by others as well; Internet and television is predicted to be measured as one by 2015 in a report [29] published by Forrester Research.

Advertisers are, for the most part, still accepting predictions based on historical data rather than current facts. Despite the limitations of purely statistical evidence, yearly spendings on television advertisements are still much higher

than for any other medium, and rising. Despite the fact that viewing habits and media delivery methods has changed drastically over the last decade, the basic methodology for measuring the impact of television content is still the same as in the early 1990s [90], with some aspects, like the survey part, dating back to the 1950s.

While the public's response to web advertisements can be analyzed and evaluated in near real-time with reasonable precision and confidence, advertisers are generally limited to base their evaluation on surveys and the daily logs of a small sample of selected households (7500 in the US as of 2010 [29]) when it comes to advertisements presented on television. A detailed survey of the current state of television audience measurement is provided in [51].

As the traditional method of television content distribution (one-way broadcast) is replaced by full-duplex distribution capabilities made possible by IPTV, the traditional survey approach used to estimate television viewership will be replaced by more accurate methods, such as the analysis of measurements obtained from STBs [29, 80, 45, 51]. We argue that there is no longer any reason to base the analysis on surveys, other than it being the established *currency* that the industry knows, referred to as *entrenched practices* [29, 62] by some.

In the 1950s, the Nielsen company [116] invented the rating system that dominates the television industry today. In Norway, the main provider of viewership data to the official television networks is TNS Gallup [142], a company that specializes in polls and ratings. The measurement methods used by TNS Gallup are still the same as those pioneered by Nielsen.

To measure the viewership, a device dubbed the *mediameter* is used to record and log inaudible sonic signatures emitted from the audio part of television and radio programs that the device is exposed to. The participants in this continuous poll are required to *carry the device with them*, and to keep the sound audible in order for the *mediameter* to record appropriately. The device must then be placed in a docking station overnight, to transmit the recorded data to TNS Gallup. In addition to the *mediameters*, TNS Gallup also collect data from 1,000 selected households whom have a specialized logging device attached to their television, but still requires operating a special remote to record

changes. The device records viewer data and transmits these every night. Viewership is computed from the collected data, where each household supposedly represents 2,000 households from the same district. This type of continuous polling represents the state-of-the-art in obtaining viewership, and similar systems are deployed in many other countries, including the US [116]. Anecdotally, non-technological approaches like *viewer diaries* are apparently also still being used [116].

The status quo in the media measurement industry is maintained by contracts that make it very difficult for competitors to unseat the existing players. Yearly spendings on long term contracts are estimated to be around $50 million per year [62] between networks such as NBC Universal and Nielsen. And the contracts runs for a long time; the current contract between TNS Gallup and the Norwegian networks runs from 2008 until 2015.

5.2 The Altibox IPTV Deployment Scenario

Here, we provide an overview of the deployment scenario in which our viewer statistics and ADSCORER applications are situated.

Altibox [6] is the largest distributor of television over a pure IP-based network in Norway, with a deployment of over 320,000 STBs, distributed amongst approximately 250,000 households. Customers are connected to two main distribution centers by fiber-to-the-home, giving customers a unique bandwidth capacity to support a variety of services, including internet, burglar and fire alarm, Voice-over-IP (VoIP), IPTV, Video-on-Demand (VoD), and Personal Video Recorders (PVRs). The STB is the host device for IPTV, VoD, and PVR services, and to simplify interacting with these services, an Electronic Program Guide (EPG) is also available to users. Technically, the EPG is essentially a database accessible through a web service interface that associates channel name to information about the programming of that channel. Currently, Altibox offers a total of 253 TV channels, accessible through the STB by way of IP multicast (over their fiber-based broadband network).

The software on STB devices are updated with new service offerings, bug

fixes, and quality-of-experience monitoring and diagnostics applications [83, 3] on a regular basis. Moreover, since STBs also have two-way communication capabilities, network operators can update their functionality to track program popularity and usage patterns by recording channel change (zap) events. This can take place without any observable changes to current user behavior, such as using a special remote or carrying a *mediameter*.

Thus, data collection is transparent to the user, and the reported data is expected to be more accurate, since users cannot forget to record the change. Moreover, we also avoid the embarrassment factor sometimes present in surveys, where users report an idealized version of their habits due to embarrassment over their factual habits. It is not unthinkable that this factor can play a role when viewers decide whether to use the special remote when viewing programming that is perceived as of lesser quality. Finally, it also allows for a much more accurate understanding of viewer behavior than existing methods, as the sample size is more than 300 times larger, representing approximately 11 % of the Norwegian television population.

In comparison, satellite television amounts for 36.6%, coaxial cable 35,2%, and the digital national distribution network has a 13.6% share. The total share of television subscriptions over fiber was 14.2% for the same period, which implies that Altibox has the majority of subscriptions in this category. These numbers were published in an official report [105], issued by the Norwegian Post and Telecommunications authority [107], and applies for the first half of 2012. The report also indicates a steady growth of fiber subscriptions, combined with a decline in satellite subscriptions (fiber is up from 8.6% two years earlier, while satellite is down from 42.7% in the same interval).

5.3 Related Work

This section gives an overview of existing work that use channel change events obtained from STBs, or in other ways challenges the existing measurement regime of television viewership.

In their 2008 study, Cha et al. [20] captured the channel changes of some

250,000 households over a period of six months in an IPTV network where channels are sent via multicast, which is the same technique used by Altibox. By thoroughly analyzing this massive data set, they were able to create more accurate statistics of user behavior than with traditional sampling methods like the ones utilized by Nielsen Media Research [116]. According to the authors, this is the first large-scale measurement of viewer channel-change behaviour in an IPTV network. In their paper, they presented some general observations, such as overall viewing patterns correlated to the time of day, identifying spikes in arrivals and departures. The analysis also revealed great local variations in channel preferences across Digital Subscriber Line Access Multiplexers (DSLAMs), which acts as switches in DSL networks. In one instance, 42% of the total viewing time spent by the subscribers connected to one particular DSLAM was spent on a specific channel during one day, while the subscribers connected to another DSLAM only spent 10% of their total viewing time on that channel during the same period.

Another observation was that over 60% of the channel changes was related to channel surfing. With this in mind, they argue that optimizing the channel selection process is especially important in IPTV networks, as the switching delay in IPTV systems is generally higher than the 100-200 millisecond delay that is the limit for what is perceived as instantaneous by viewers. Suggestions for optimizing the channel selection process include dynamic reorganization of the channel list, either based on channel popularity for each user, or by overall program popularity. Even though they do not present a solution for calculating viewer statistics in real-time, the point is made that real-time statistics can be achieved by exploiting the bidirectional capabilities of STBs in an IPTV network, and used to assist users in selecting channels. This work is closely related to ours, in that it analyzes channel changes from a large IPTV network, however, it is strictly a statistical analysis of IGMP logs, and does not consider any real-time applications. Since the study is based on IGMP logs, the only events containing information about viewing behaviour are IGMP *join* and *leave* entries.

The viewer statistics applications that will be presented later in this thesis,

on the other hand, takes a slighly different approach, as they are based on a software agent deployed on each STB. This agent, which we have dubbed the ZAPREPORTER, reports a number of additional user actions, including HDMI status (indicating whether the television set is powered on or off), mute/unmute and volume, which facilitates a deeper understanding of viewer behavior than channel changes alone. Furthermore, these attributes are all analyzed in near real-time, with the results being made available through a push-interface. This opens up a wide range of possible applications, such as annoyance detection (discussed in Chapter 6) that would be impossible to implement if restricted to historical channel change data.

Commercial vendors like JDS Uniphase [131], Mariner [91] and Agama [3] delivers agent-based solutions for monitoring QoS that also provides channel usage statistics. However, the interaction model of these solutions are all pull-based, and typically stored in a relational database, either wrapped in a SOAP API, graphical view from within the application, or through export functions that allows users to export historical data to a file. The long query delays of such systems makes them unsuitable for the purpose of computing channel statistics and presenting them in near real-time. Thus, none of the commercially available solutions today have interaction models that is suitable for incorporating their functionality into a larger event-driven architecture. Moreover, they cannot be used to develop specialized applications like annoyance detection. The reasons for this can probably be attributed to business protectionism, attempting to lock IPTV operators to their solutions as much as possible, coupled with limited knowledge of the push-based interaction model that is vital in developing event-driven architectures and real-time functionality.

What separates our work from previous work is that none of the aforementioned solutions leverage event stream processing to compute online channel usage statistics, limiting their use to identifying historical usage trends. Furthermore, we provide added insight through monitoring more parameters in addition to channel changes, such as HDMI and mute status. By performing the computations online in near real-time, we are able to provide the users and operators with the added value of having instant access to emerging usage trends.

5.3.1 Methods For Measuring Advertisement Response

In this section, we discuss existing methods for measuring how the audience responds to televised advertisements, providing background information and motivation for the ADSCORER advertisement scoring application.

Kempe et al. [80] argues that audience response should be reflected in the pricing, ordering and selection of ads within a commercial break. The current pricing structure, where advertisers are not held directly accountable for causing viewers to change the channel, does not carry enough incentive to keep viewers watching. The goals of advertisement effectiveness and avoiding viewer annoyance are often conflicting; though a loud and silly jingle might annoy many viewers, and cause them to switch channels, its popularity as a stimuli in televised advertisement speaks to its effectiveness. Recent technological advances facilitating time-shifted viewing[1] and easily accessible on-demand content, combined with an ever-increasing selection of channels has made it easier than ever before to avoid advertisements, by fast-forwarding over these. With this in mind, it is becoming increasingly important for television networks to hold on to their viewers.

A number of algorithms to measure viewer behavior in response to commercials are presented by Kempe et al. [80], concluding with a more sophisticated algorithm that builds upon the insight gained from these, named the Audience Value Maximization Algorithm (AVMA).

Dorai-Raj et al. [45] also advocates a business model that to a greater extent considers the audience response to advertisements in television. The idea is to apply many of the techniques used in online advertising to television. One of their proposed units of viewer response measurement is the Initial Audience Retained (IAR) metric, which is included in the scoring results of the ADSCORER system, presented later in Section 7.3. The IAR metrics is a very simple one, and computes the fraction of viewers retained for the duration of an advertisement. It is calculated as follows:

$$IAR = \frac{\epsilon}{\alpha}$$

[1] The recording or pausing of a television program, to be viewed or resumed at a more convenient time.

where ϵ represent the number of viewers that stayed on the channel throughout the advertisement and α represent the number of viewers at the start of the advertisement. By excluding the viewers that were not present at the start of the advertisement we eliminate most channel surfers (viewers that constantly flicks between channels during the commercial break).

We implement the IAR metric in ADSCORER because it is easy to implement, and gives an intuitive understanding of viewer behaviour in response to an advertismement. It is easy to implement alternative scoring metrics and in ADSCORER, but we leave this as future work.

An obvious, but important insight presented by Kempe et al. [80], is that while online ads can be measured through the positive action of a click, viewers are primarily limited to the negative action of changing the channel in response to televised commercials.

However, the actions of muting/unmuting the audio or turning off the TV or STB are not mentioned by Kempe, or in any of the other papers we reviewed, but also belongs to the current repertoire of viewer responses. The inclusion of the aforementioned user actions is one of the features of ADSCORER that sets it apart from other systems.

At the commercial end of the spectrum, Rentrak [64] appears to be the market leader for STB data aggregation, and is already collecting usage data from millions of STBs deployed by AT&T, Charter, Dish Network and Midcontinent Communications [64]. One of their products; *AdEssentials* includes total ad impressions, average viewing time and unduplicated unique views per advertisement among its metrics.

UK satellite operator BSkyB is another actor in the STB data market that are already collecting STB usage data from over 30 000 devices, correlating these with brand purchasing history from many of the same homes [18].

TRA [143] also combines STB data with credit card transactions, using a third-party blind matching method, where TRA never sees any addresses or names involved in the transactions, in order to measure the effectiveness of advertising campaigns, as well as profiling viewer groups. Data generated from the ADSCORER system could also be correlated with purchase activity in the

same manner as TRA and BSkyB, and to target ads, like BSkyB intend to do in 2013, provided that privacy laws permits it.

CasterStats [19] provides audience measurement for streaming media, in the form of reports that can be generated through a web interface. However, this appears to be limited to media distributed over the internet and not broadcast television media, unlike the work presented in the following chapters.

Coalition for Innovative Media Measurement (CIMM) [26] is a coalition of television content providers, media agencies and advertisers intent on finding new and better ways to measure television media consumption, in the changing media landscape. A main objective of this effort is finding values and applications of STB data, and their contributions include an analysis for the STB viewer measurement data landscape, as well creating and maintaining metrics and an ontology relating to STB data [27]. If CIMM were to succeed in establishing a common standard for STB viewer measurement data, it would greatly benefit all actors who have access to STB data, including Altibox.

5.4 The Future of Media Measurement

Because the competition for audience attention has become increasingly intense through the digitization of media, the advertising industry need to continuously improve and re-evaluate its measurement and targeting methods. The reasoning behind this is that information consumes the attention of its audience, and while telecommunication bandwidth is practically infinite, human bandwidth is becoming increasingly scarce [62]. A logical conclusion that can be extracted from this insight, is that television networks should change their business model from selling audience exposure in the form of network time to selling viewer attention, as argued by Kempe et al [80].

Where the traditional mass media channels have established currencies for audience measurement, no such standard currency exists for Internet audiences. In 2002 the Interactive Advertising Bureau attempted to establish a standard set of guidelines on how to count impressions. However, this ended up “extremely confusing and ultimately a compromise” [62]. The main reason for this is the

sheer complexity involved in delivering the content.

The difficulties of standardization, however, is unlikely to prevent new models of media measurement from emerging in the near future, as the advertisers and content providers becomes aware of the opportunities of more accurate audience targeting afforded by technologies.

As mentioned earlier in this chapter, many predict that the current online/offline division of media will be soon replaced by measured versus unmeasured [45, 29]. Despite these predictions, the established currencies in mass media audience measurement are unlikely to go away anytime soon, simply because advertisers and media channels needs to agree on a common measure for pricing, even if this is inaccurate [62]. However, the increasing expectations for accountability will create a market for additional, more accurate measurements that can supplement the standard currencies, and may eventually replace them.

The repertoire of viewer actions is likely to grow in the near future, as the media of television gains more interactivity. Examples include interactive links to buy a product, rating possibilities, like/dislike buttons and games.

5.5 Summary

This chapter has provided background and motivation for the work presented in Chapters 6 and 7, discussing the current state of the media measurement industry and its main actors, as well as providing a brief account of its history. We also discussed how the status quo is being challenged by more accurate measurement methods, facilitated by the two-way communication capabilities of modern-day STBs, by companies such as Rentrak and TRA.

By ignoring volume changes, muting/unmuting and HDMI status information, current academic research and commercial offerings does not take advantage of the full repertoire of viewer response actions. Furthermore, while current offerings are restricted to historical viewer statistics, the work presented in the following chapters, demonstrates how viewer statistics can be made available in near real-time, using event processing techniques.

Chapter 6

Computing TV Channel Statistics from Channel Zap Event Streams: A Paradigm Comparison

In this chapter, we present a paradigm comparison that addresses Research Challenge 5. To perform this paradigm comparison, we develop a television viewer statistics application that is novel in its own right. It addresses the current limitations of television viewer measurement, caused by small sample selections, one-way communication and offline reporting, as outlined in Chapter 5. Most of the work presented here was published at the 5th ACM International Conference on Distributed Event-Based Systems (DEBS) in July 2011 [51].

Chapter organization: Section 6.1 gives a brief introduction to the work presented here, and its context. Section 6.2 describe the architecture of our current deployment, and outline plans for improving the accuracy of statistics and provide new services to customers. Focusing on the event processing logic, Section 6.3 describe the details of our viewer statistics application, and its implementation in the two programming paradigms that we compare. Section 6.4 describe our annoyance detection algorithm, which is also implemented in both

paradigms.

In Section 6.5 we give a brief analysis of the viewer statistics obtained from our current deployment. Additionally, we evaluate our implementations in terms of throughput and memory usage, and program complexity. Finally, Section 6.6 concludes the paper with an overall discussion of the merits of the two paradigms.

6.1 Introduction

Viewer statistics is the most important metric used by television broadcasters to plan their programming, and for many broadcasters, to rate their advertisement time slots. Gaining an improved understanding of viewer behavior and responses to the current programming is essential to a successful TV channel, going forward in a highly competitive TV market. The state-of-the-art approach to obtain the viewership of a program is to *sample* a very small *selected, but hopefully representative* portion of the population. In Norway, the sample size is 1,000 out of 2,000,000 television households (0.05 %) [59], while in the US, only 25,000 out of 114,500,000 households are sampled (0.02183 %) [116]. Such a small sample size is often criticized as being statistically insignificant [116], and may lead to incorrect conclusions about actual viewer interests in a specific program, and viewer exposure to advertisements.

With the enhanced capabilities of present-day STBs, network operators can track channel/program popularity and usage patterns with a degree of precision and sophistication not possible with existing methods. This can be done by recording or aggregating channel change events (also called ZAP events) from customer STBs. Hence, assuming that the network operator's customers represents a statistically significant portion of the population, collecting statistics based on ZAP events is likely to provide a much more accurate statistic compared to state-of-the-art.

There are generally two approaches to compute accurate viewership. One is to store every ZAP event for later bulk processing, e.g. using transactional databases or techniques based on Map-Reduce [39, 86], or aggregate statistics

can be computed on-the-fly, based on in-memory state. We take the latter approach, as we are mainly interested in aggregate information from these events and want to avoid storing huge volumes of data generated by channel surfers. Only aggregate numbers are stored on disk or forwarded to interested parties.

In this chapter, we describe the architecture in which STBs are deployed, and how channel ZAP events are propagated to an aggregation cluster for online incremental event processing. Based on this architecture, our goal is to analyze the capabilities and trade-offs between two programming paradigms for building our application to obtain viewership statistics. Hence, we have implemented two applications that compute two different statistics based on received ZAP events:

1. The number of viewers for all channels.

2. Detecting a 15 % rise/drop in the viewership for a channel.

The first is used to generate a *top-ten* list of the most popular channels and programs in near real-time. The second application reveals useful information about which programs are luring people away from other channels. An important characteristic of both these applications is that they are stateful and demand significant computational resources to ensure timely processing. Although the applications that we cover here are fairly simple, their statistical measures might be interesting to network operators and broadcasters. Furthermore the calculation of viewer numbers per channel provides the foundation for the more sophisticated ADSCORER application, presented in Chapter 7.

The two applications have been implemented in two very different programming paradigms. One based on the general-purpose object-oriented programming language Java, and the other based on the EPL language [28, 98], introduced in Section 4.2.1. We are interested in exploring the trade-offs between these two paradigms, to determine their suitability for our applications. Specifically, we are interested in the performance trade-off and the program complexity of each implementation.

Java is expected to have higher program complexity than EPL, since EPL is specifically designed for processing events. We compare our implementations

using on several metrics for analyzing code complexity, including lines of code and Halstead's complexity measure [73, 36, 71], in addition to a more subjective discussion based on our experience developing these applications. The results indicate that EPL might yield easier reuse compared to Java. Because EPL facilitates event stream reuse and includes a number of constructs for expressing event patterns with fewer lines of code, it is easy to prototype.

Previous, but simple, benchmarks conducted with the EPL benchmark kit [46], have indicated that it would offer competitive performance. However, at the outset of this work it was not clear if EPL would offer competitive performance for our somewhat involved applications. Our performance evaluation involves data obtained from more than 250,000 STBs. We conduct both memory profiling and throughput analysis, and find that our implementations of these applications have very different performance characteristics in the two programming paradigms.

6.2 Architecture

In this section, we present the architecture of the deployment at the time when our experiments were conducted. We then discuss some changes that were implemented after the measurements were obtained. These changes will hopefully significantly improve the accuracy of future measurements, at the expense of more demanding processing and network overhead. We also outline a few applications that become possible with more accurate measurements.

6.2.1 Deployment Used During Experiments

The STB devices deployed in customer residences for supporting IPTV, VoD, and PVR, are fully capable of two-way communication, and have been augmented with a software agent to keep track of and report ZAP events to a centralized server. We call these the ZAPREPORTER client and ZAPCOLLECTOR server, respectively. A simplified architecture is illustrated in Figure 6.1.

The ZAPREPORTER monitors channel changes performed by the user of the STB, and generates ZAP events containing the following information:

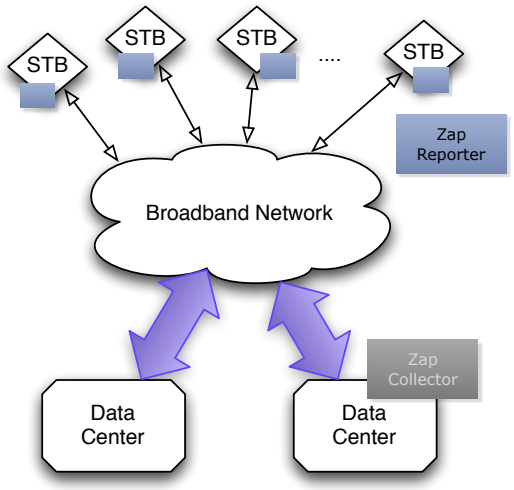

Figure 6.1: Network architecture illustrating the Altibox IPTV network.

⟨STB-IP, TIMESTAMP, TOCHANNEL, FROMCHANNEL⟩

The event is encoded as text, and one event is typically less than 60 bytes, hence approximately 25 events can be sent in one 1500 byte long UDP/IP packet. The clocks on the STBs are synchronized using the Network Time Protocol (NTP), and thus provide higher accuracy than what is needed for our purposes. The ZAPREPORTER records ZAP events with a per second granularity. Events are generated according to Algorithm 1, and described informally as follows: When a user changes channel, the ZAPREPORTER record this change event locally on the STB and may batch together multiple change events in the same UDP/IP packet. T is the number of seconds a viewer has to stay on the same channel in order for the channel change to be recorded. S is the timeout period in minutes between sends.

At the time of the experiments, the ZAPREPORTER was configured in the following way: On the first channel change, a timer is started. If the user stay on the same channel longer than 60 seconds, the event is saved away in *unsent*. If the user changes channel again before the 60 second timer expires, the event is overwritten (i.e. not recorded in *unsent*). Periodically, the events stored in *unsent* is sent off to the ZAPCOLLECTOR and emptied.

This strategy ensured that the total number of messages sent was kept to approximately two messages per hour per STB, and at most 30 events needs to be kept in STB memory. With this configuration, we expected that we would rarely see more than 50 events generated by the same STB in one hour, requiring more than two 1500 byte messages to be sent. Hence, in the worst case, when all 320,000 STBs were active, we might see a total of 640,000 messages per hour, or just over 2 Mbps (on average). Moreover, if the hourly total message count per STB was 50 ZAP events, the processing rate would have to be about 4.4k events/second.

On the server side, the ZAPCOLLECTOR collect events from all the STBs and store them in log files that are rotated daily. The events are stored in the order they are received from the STBs. However, since each message contained about 30 minutes worth of ZAP events, the log files are not initially sorted by the timestamp. Therefore, the event logs must be sorted before they can be used to produce incremental statistics. At the time this work was conducted, there were no service offerings at Altibox that took advantage of these log files. With the viewer statistics applications presented in this and the next chapter, the data generated by the STBs are put to use.

Algorithm 1 ZAPREPORTER pseudo code

```
1:  Initialization:
2:  T ← 60                                   {Timeout period (seconds)}
3:  S ← 30                                   {Period between sends (minutes)}
4:  event ← ⊥                                {Most recent event, not yet recorded in unsent}
5:  unsent ← ∅                               {Set of unsent zap events}
6:  startPeriodicTimer(⟨SENDTIMEOUT⟩, S)
7:  on ⟨CHANNELCHANGE, toCh, fromCh⟩
8:      event ← prepareEvent(toCh, fromCh)   {Update event}
9:      restartTimer(⟨RECORDTIMEOUT⟩, T)
10: on ⟨RECORDTIMEOUT⟩
11:     unsent ← unsent ∪ event              {Record event}
12: on ⟨SENDTIMEOUT⟩
13:     ∀e ∈ unsent : send ⟨ZAPEVENT, e⟩ to ZAPCOLLECTOR
14:     unsent ← ∅
```

6.2.2 Current Deployment

The one-minute granularity of the ZAPREPORTER deployed at the time of the experiments is too low to capture all the nuances of the viewers' behaviour. Furthermore, the 30-minute batching of events, described in Algorithm 1, made any near real-time functionality impossible to implement.

Some time after the measurements presented in this and the following chapter, we were able to deploy a new ZAPREPORTER in the Altibox network. Note that, the ZAPREPORTER functionality implemented in STBs is beyond the direct control of the authors of this work. However, we were able to influence and request implementation changes to the STBs. The reasons for this is corporate policies relating to accountability for changes that can potentially cause problems for customers. Moreover, the STB can only be updated two times a year, during a relatively short time window. Hence, this poses some challenges for us in implementing the desired functionality.

There are several reasons why we are interested in increasing the accuracy of these statistics. First of all, we want to be able to provide a ranking (top-10 list) of programs in near real-time to both viewers and broadcasters. Also, we are interested in detecting flash crowds, i.e. when a large number of viewers change to or from the same channel within a short period of time. This might be expected either when a new (popular) program is beginning, or during commercial breaks. The former we have seen evidence of from our current datasets. However, to understand better the user behavior in commercial breaks, we need more accurate information from the ZAPREPORTER. Also to provide a real-time ranking, we must to revise the ZAPREPORTER.

In the current deployment we report channel changes (lasting 10 seconds or more) within a 10 second interval. In Algorithm 1, S represents the minimum time a viewer has to stay on a channel, in order for it to be stored in the *unsent* buffer, while T represents the timeout for reporting the stored events back to the ZAPCOLLECTOR.

Thus, in Algorithm 1, we set $S = 10$ seconds, and $T = 10$ seconds. This, in effect means that the *unsent* buffer will never contain more than a single event. Technical limitations in the ZAPREPORTER which are beyond our control are

the reason for not setting these attributes to even lower values. Obviously, no message will be sent if there are no channel changes. This also means that no packet will ever contain more than one ZAP event. Limitations in the ZAPREPORTER prevented lower values than 10 seconds for S and T, though ideally, we would like T to be smaller, as it would enable us to capture zapping behaviour at an even finer resolution than what is currently possible.

To determine the worst case network resources necessary with this sampling frequency, assume all 320,000 STBs generate 1 event every 10 seconds. Assuming every event takes 60 bytes, the packet size should be roughly 60+70 bytes (including headers). Under these assumptions, the worst case network load would be around 33 Mbps overall, and 32,000 events/second would have to be processed. These numbers are obviously above what is expected in the normal case, but we would like to be able handle flash crowds that might reach towards such numbers.

On the ZAPCOLLECTOR end, we introduced a ZAPPROCESSOR to process events incrementally to compute statistics for program ranking in near real-time, and for detecting flash crowds and other similar statistics. We have implemented these services and in Section 6.5 we evaluate our ZAPPROCESSOR implementations in both Java and EPL, based on real data obtained from our log files.

6.3 Viewer Statistics

In this section we present two implementations of an application for obtaining viewership statistics, one implemented in Java, and the other implemented using EPL. We describe both implementations in detail, specifically focusing on the event processing aspect.

In order to obtain statistics for the different channels, we simply count the occurrences of ZAP events changing to the different channels. Moreover, we also have to reduce the count for the channel the STB is moving away from (or the previously recorded channel of that STB). We do not reduce the count of any channel if the event originate at an STB from which we have no recorded events. With this approach it will take some time to build up the data needed to

compute statistics since not all STBs have its channel state recorded.

6.3.1 Java Implementation

Algorithm 2 shows Java-like pseudocode for the ZAPPROCESSOR implementation used to obtain viewer statistics. The core of the algorithm (lines 9-18) deals with processing ZAP events received from STBs. Lines 10-13 of Algorithm 2 checks to see if the STB have been active in the past, and if so replaces the *fromCh* field of the message with the last recorded previous channel. This is necessary because not all channel changes are propagated to the ZAPPROCESSOR, due to the 1 minute rule imposed by the ZAPREPORTER. Otherwise, our counting in the last part would not be correct.

In the Java implementation, we implement the counting of ZAP event occurrences using a multiset, where each entry (the channel) is associated with a count value representing the number viewers on that channel.

Periodically, output events are generated by first determining which channels have the most viewers, and for each channel query the EPG to determine which program is currently being broadcast on that channel. To avoid frequent database queries, we cache program information in memory. From this we construct the top-10 list of programs to be sent to interested subscribers, providing near real-time viewership information. One such subscriber that we have implemented is the EPG itself. In this case, we integrate the top-10 list within the program guide interface on the STB device, enabling users to see statistics and choose program from the list.

An important improvement that these real-time viewer statistics provide over batched statistics is that broadcasters could potentially adjust their advertisement programming based on actual viewer numbers.

6.3.2 EPL Implementation

Both the Java and EPL implementations share a common logic in how events are handled (see Figure 6.2 and Algorithm 2). However, the EPL implementation requires a slightly different understanding of how events are related, and hence

Algorithm 2 ZAPPROCESSOR pseudo code

```
1:  Initialization:
2:  R                                              {Subscribers of output events}
3:  EPG                                            {Electronic Program Guide database}
4:  S ← 10                                         {Period of between output events (seconds)}
5:  STBs ← ∅                                       {Set of known STB-IP addresses}
6:  viewers ← ∅                                    {Multiset: viewer count for each channel}
7:  prevCh ← ∅                                     {Map from STB-IP to previous channel}
8:  startPeriodicTimer(⟨OUTPUTTIMEOUT⟩, S)

9:  on ⟨ZAP, ip, timestamp, toCh, fromCh⟩
10:   prev ← prevCh.get(ip)                        {Get previous channel of ip}
11:   if prev ≠ null then
12:     fromCh ← prev
13:   prevCh.put(ip, toCh)                         {Update previous channel of ip}
14:   viewers.add(toCh)                            {Increase count of toCh}
15:   if ip ∈ STBs then                            {Have we seen STB before?}
16:     viewers.remove(fromCh)                     {Reduce count of fromCh.}
17:   else
18:     STBs.add(ip)                               {New STB, record ip}

19: on ⟨OUTPUTTIMEOUT⟩
20:   topProgList ← ∅
21:   topCh ← viewers.mostFrequent(10)             {Top-10 channel}
22:   for ch ∈ topCh
23:     prog ← EPG.getProgram(ch)                  {Query EPG}
24:     topProgList.add(prog)                      {Create top-10 program list}
25:   send ⟨TOP10LIST, topProgList⟩ to R
```

the following gives a more succinct description from the EPL perspective, while the algorithmic descriptions closely match the Java implementation.

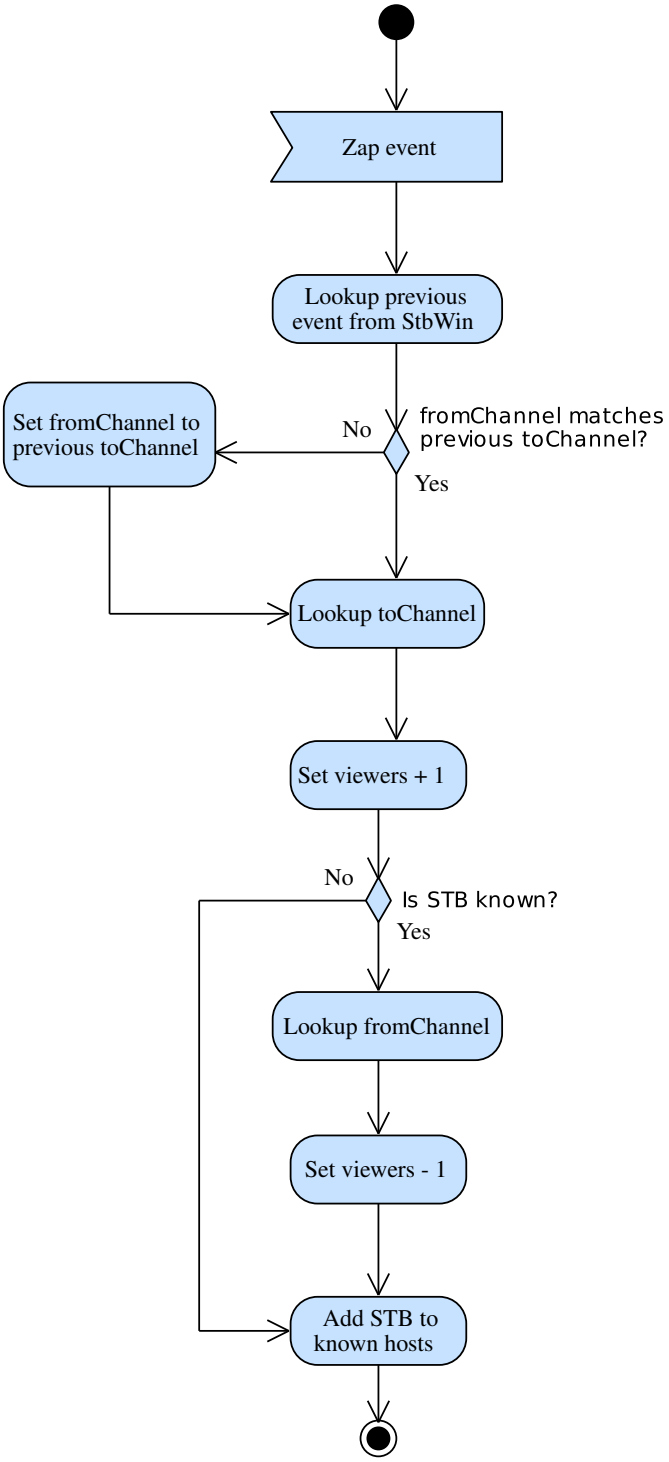

Figure 6.2: Viewer Statistics Activity Diagram

As shown in Figure 6.2, incoming events are matched against the previous event received from the same STB, comparing its *fromChannel* field with the *toChannel* field of the STB's previous ZAP event. Different values might be observed at this stage, either due to packet loss, or more likely due to the way that the ZAPREPORTER generate events (not all events are actually sent). To compensate for this, we set the *fromChannel* of the incoming event to the *toChannel* of the previous event. If no previous event exist, no action is taken.

The next step is to update the number of viewers by adding one to the channel matching the *toChannel* field of the ZAP event and subtracting one from the channel matching the *fromChannel* field. If the event received is from a previously unknown STB, the *fromChannel* is not subtracted, as it is the first event received from this particular STB. This is appropriate since the STB has not attributed to any channel counts. Finally, the STB is added to the list of known devices.

The EPL queries contains all of the logic illustrated in Figure 6.2, while the EPL implementation also requires some Java code to handle parsing and object creation for incoming events. Also, listener objects must implement a callback interface in Java to receive output events generated by the Esper engine. We have not included the Java code.

Listing 6.1 shows the complete EPL code for generating viewer statistics. It consists of 10 statements, and makes extensive use of *windows*, which is an Esper abstraction that provides a view of an event stream.

The first statement (lines 1 and 2) defines a simple datatype; *ChannelTotViewers*, while the next statement (lines 3 and 4) creates a data window that operates on a stream of objects of this type. The *std.unique()*-operator tells the window to only keep the last object based on a given key attribute. In this case, the key attribute is the *channelName* of a *ChannelTotViewers* object.

Next, we define the window *StbWin*. This window collects the first ZAP event from each unique STB, using the *firstunique()*-operator. It is used to keep track of known STBs. The *tv.Zap* variable refers to the Java object created when parsing incoming ZAP events. To keep track of the last ZAP event from each STB, we create the *ZapWin*, beginning on line 7.

The *update istream* query is necessary to compensate for any discrepancy in from/to channel values, as described above, and operates on the ZAP event before it enters any stream.

Lines 18-30 of Listing 6.1 implements the core logic of the viewer statistics application, as illustrated in Figure 6.2 and listed in lines 10-18 of Algorithm 2. This part of the code updates *ChannelWin*, containing viewer numbers, as well as adding the STB to the *StbWin*, containing known STBs.

The *insert* statements in lines 10 and 32 of Listing 6.1 populates the previously defined windows. Esper performs the statements in the order of which they are installed, and for the logic to be correct, the *StbWin* containing the list of known STBs must not be updated before the core logic has been performed. If the insert operation listed on line 32 in Listing 6.1 is performed before lines 10-30, the *exists* condition listed on line 28 will always return true.

The final statement in Listing 6.1 outputs an ordered snapshot every 15 seconds into an event stream named *ZapSnap*. The snapshot is taken from *ChannelWin*, which contains channel name and the total number of viewers on that channel. The statement augments these attributes with a percentage value, providing the share of the total viewers, which is calculated by a custom method implemented in Java.

The reason for using the whole *tv.Zap* object most of the time, instead of extracting only the necessary values is that according to the Esper documentation [28], selecting individual properties from an underlying event object comes with a performance penalty, as the engine must then generate a new output event containing exactly the selected properties. Additionally, it simplifies the syntax.

We ran both the Java and EPL implementations with the same datasets as input, and after a few rounds of debugging, we observed identical output for both implementations.

Listing 6.1 EPL Viewer Statistics

```
1  create schema ChannelTotViewers
2    as (channelName string, viewers int)
3  create window ChannelWin.std:unique(channelName)
4    as ChannelTotViewers
5  create window StbWin.std:firstunique(ip)
6    as tv.Zap
7  create window ZapWin.std:unique(ip)
8    as tv.Zap
9
10 insert into ZapWin select * from tv.Zap
11
12 update istream tv.Zap as zap
13   set fromChannel =
14     (select toChannel from ZapWin where ip = zap.ip)
15   where fromChannel !=
16     (select toChannel from ZapWin where ip = zap.ip)
17
18 on tv.Zap zap merge ChannelWin cw
19   where zap.toChannel = cw.channelName
20   when matched
21     then update set viewers = viewers + 1
22   when not matched
23     then insert
24       select toChannel as channelName, 1 as viewers
25
26 on tv.Zap zap merge ChannelWin cw
27   where zap.fromChannel = cw.channelName and
28     exists (select * from StbWin where ip = zap.ip)
29   when matched
30     then update set viewers = viewers - 1
31
32 insert into StbWin select * from tv.Zap
33
34 insert into ZapSnap
35   select *, percent(viewers, sum(viewers)) as activity
36     from ChannelWin
37     output snapshot every 15 sec
38     order by viewers desc
```

6.4 Annoyance Detection

Here, we discuss the second application which is aimed at detecting if a particular ad is causing viewers to change channel. Broadcasters would most likely want to know about this, in order to remove or charge more for ads that annoy or upset viewers. To support such ad annoyance detection, we must detect changes in the viewership beyond some threshold, e.g. measured as a fraction, P, of the total number of viewers on that channel.

6.4.1 Java Implementation

Here, we present the Java implementation of the annoyance detection algorithm. Algorithm 3 shows the additional code necessary for such annoyance detection. To implement this, we again rely on a multiset to keep a count of the number of ZAP events seen in the current interval. The interval width used in this case is 60 seconds, but this can easily be adjusted for more fine grained intervals.

Note that *ival* is an integer, and the $\oplus$ symbol represents concatenation. Hence, the element of the multiset is the concatenation of channel name and an integer representing an interval. To ensure that memory usage is kept low, we immediately expunge data from a previous interval, and if an output event is generated within one interval, we reset the counting for that interval. This allows multiple output events to be generated for the same interval, if the fraction of viewers changing channel in that interval is $\geq 2P$.

6.4.2 EPL Implementation

The EPL implementation of the annoyance detection algorithm builds on the viewer statistics application from Section 6.3.2, and demonstrates how effortlessly EPL implementations can be augmented with new functionality.

The annoyance detector in Listing 6.2 looks at the average viewer number over the last minute, constantly comparing the most recent number with the average. If the viewer number drops with 15 % compared with the last minute average, an output event is triggered.

Algorithm 3 Annoyance detection pseudo code

```
1:  Initialization:
2:  F                          {Multiset: counts viewers moving from channel in intervals}
3:  M ← 2000                   {Minimal # of viewers to consider for detection}
4:  P ← 0.15                   {Fraction of viewers moving from channel in interval}
5:  W ← 60                     {Interval width}
6:  prevIval ← ⊥               {The previous interval}
7:  on ⟨ZapEvent, date, time, ip, toCh, fromCh⟩
8:    ival ← time/W            {Get interval of this event (sec)}
9:    if ival ≠ prevIval then
10:     F.clear()              {New interval begun; expunge old entries}
11:   prevIval ← ival
12:   F.add(fromCh⊕ival)       {Increase count changing from channel in ival}
13:   F.remove(toCh⊕ival)      {Reduce count for channel moved to in ival}
14:   v ← viewers.count(fromCh) {#Viewers on fromCh}
15:   if v > M ∧ F.count(fromCh⊕ival) ≤ P · v then
16:     Generate output
17:     F.setCount(fromCh⊕ival, 0) {Reset count for ival}
```

To construct the annoyance detector query, we make use of the *ZapSnap* event stream from the viewer statistics code in Listing 6.1. For this case, the power of sliding time windows is illustrated: The query selects properties from the *ZapSnap* stream of viewer statistics, exposed as a sliding time window.

Every 15 seconds, a snapshot of the viewers on each channel combined with additional statistics are published to the *ZapSnap* event stream. As in the Java implementation, channels having less than 2000 viewers are filtered out before they enter the window in order to prevent channels with only a few or no viewers from triggering annoyance events. The average number of viewers is calculated from the events kept in the 1-minute window, while events older than this leave the window.

Listing 6.2 EPL Annoyance Detector

```
1 select channelName, viewers, avg(viewers)
2   from ZapSnap(viewers > 2000).win:time(1 min)
3   group by channelName
4   having viewers < avg(viewers) * 0.85
```

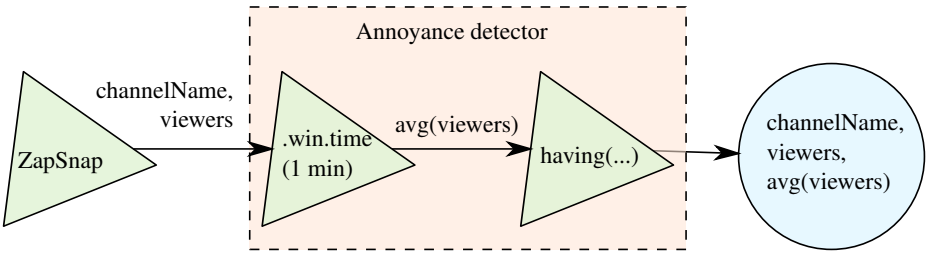

Figure 6.3: Annoyance detector illustrated

Figure 6.3 illustrates the EPL statement from Listing 6.2, and shows how aggregated events, containing channel statistics from the *ZapSnap* event stream, are fed into a sliding window. If the average number of viewers in this sliding window drops by more than 15% in less than a minute, an output event is generated.

6.5 Evaluation

The main goal of this chapter is to evaluate two paradigms for developing event-based systems, and specifically if it can be applied to our enhanced high-volume use case. Moreover, in this section we first give a brief analysis of the data obtained from the initial deployment of ZAPREPORTER. This will be followed by a performance benchmark and software complexity evaluation.

6.5.1 Brief Data Analysis

To be able to predict the kind of traffic one might expect, when scaling up the number of events that will be generated, we examine the current trend of channel zapping. Hence, we selected a 15-day period (January 31 – February 14) from our logged datasets obtained using the current infrastructure at the time, as described in Section 6.2.1. This period constitutes approximately 1.7G bytes of data, or about 118M bytes per day. The sampled dataset contains events from 253,985 unique STBs, and 183 different channels were visited at least once during the period.

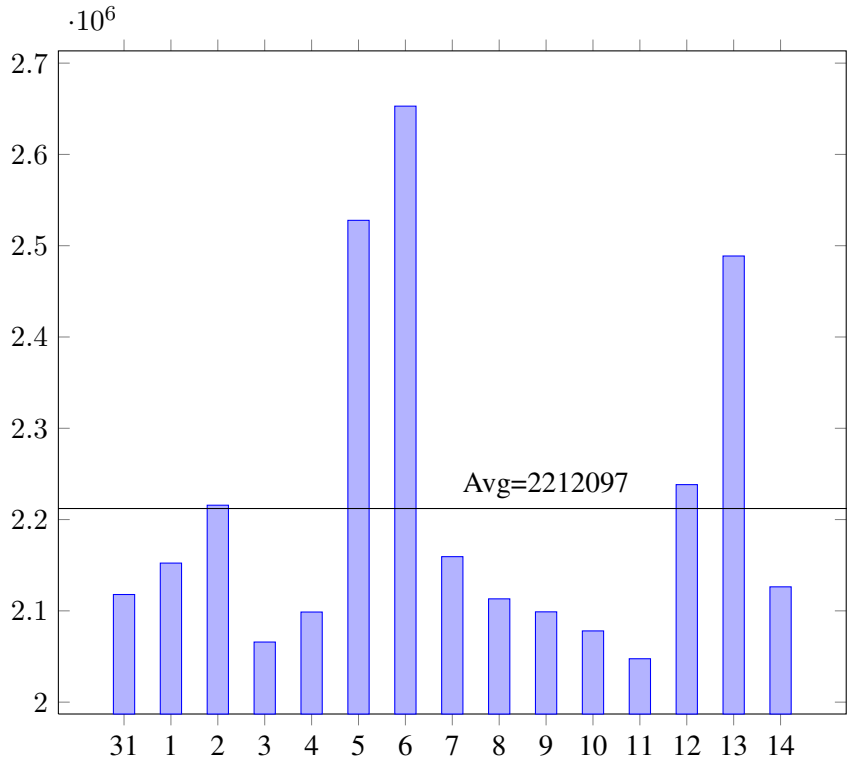

Figure 6.4: Number of zap events/day over a 15-day period.

The number of events generated each day is shown in Figure 6.4, and the same data is also shown in Figure 6.5, sampled at hour intervals. An interesting observation from Figure 6.4 is that Wednesdays (5,12) and Thursdays (6,13) represent a significant deviation from average zapping activity. We speculate that this might be due to poor programming on these days across the board among broadcasters. In Figure 6.6, we show the distribution of zap events over a 24-hour period based on data from January 31. The plot confirms what is expected from habitual patterns, with a peak in zapping activity around 20:00. We leave it for future work to provide an in-depth analysis of these data, when we have better accuracy.

The data for the statistics presented in Figure 6.7 was sampled at Saturday the 21st of May, 2011. The graphs shows the viewing trends for different regions for TV2, the second largest channel in Norway, and illustrates viewer numbers,

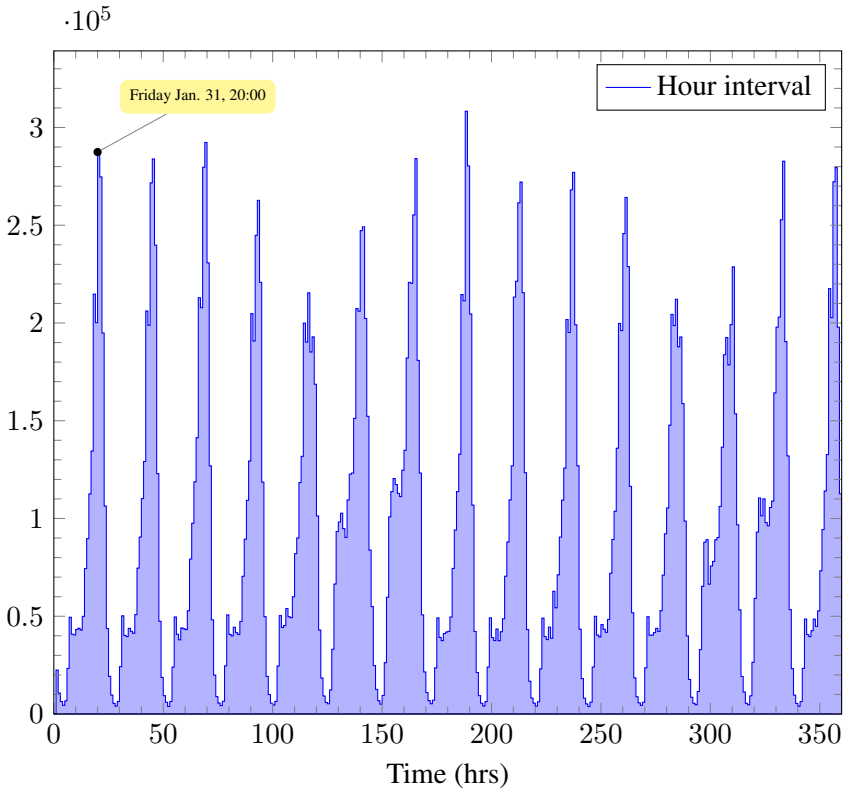

Figure 6.5: Number of zap events/hour over a 15-day period.

calculated from ZAP events. We used the EPL implementation from Listing 6.1 to calculate these numbers.

Placing these trends on top of a canvas containing the channel programming for the same time period makes the relationship between the number of viewers and the programming quite clear. For instance, one can see that the most popular program on that particular evening was the nine o'clock news, which should come as no surprise to those familiar with Norwegian viewing habits. Another interesting observation that can be made from this graph is the spike in viewer numbers at around 17:40, which correlates with a summary of highlights from the bicycle tournament "Giro d'Italia".

In addition to presenting the total viewer numbers for TV2, Figure 6.7 also divides the viewers into four regions: North, Center, East and South/West, which is a common division of Norway. Since these graphs reflects actual

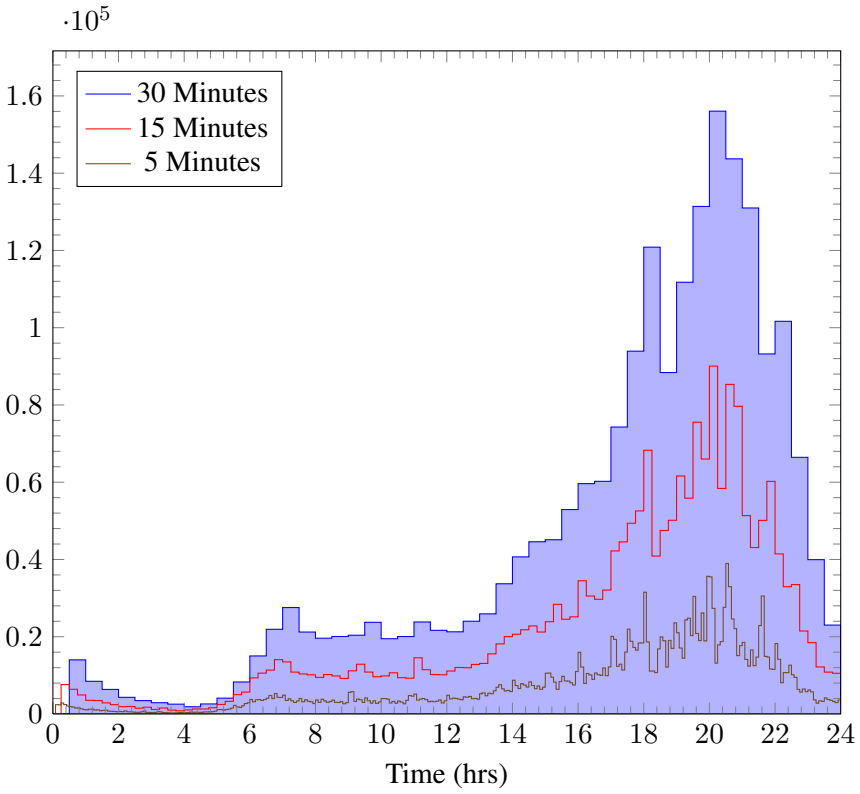

Figure 6.6: Number of zap events over a 24-hour period for different sampling intervals: 5, 15, and 30 minutes.

viewer numbers, they do not indicate relative share. Instead, they give an indication of the number of deployed STBs in the different regions. As can be observed from the plot, viewer number fluctuations are fairly identical for all the regions, suggesting that viewing patterns are quite similar across the whole country. There are some minor differences, suggesting that the viewers in the eastern part of Norway leaves the channel sooner during a commercial break, when compared to viewers in the south and west. This can be observed in the commercial break after the nine o'clock news, where we see a slightly steeper drop curve for the viewers in the eastern part of the country.

Figure 6.8 combines the top graph from Figure 6.7 with viewer numbers from NRK1, which is the largest television channel in Norway. Note the inverse correlation between the two graphs, suggesting that a majority of viewers on

both channels were switching internally between the two channels. This graph also illustrates that the interval between 18:00 and 22:00 is the prime time viewing hours on a Saturday evening. NRK is government owned and commercial free, while TV2 is a commercial network, which might explain why the TV2 graph contains more fluctuations, like the sawtooth pattern that can be observed around eight o'clock.

6.5.2 Performance Evaluation

Here we provide a brief performance evaluation of both our implementations for the viewer statistics application. The annoyance detector application was difficult to test with Esper due to lack of real data, and for this reason was not benchmarked.

Environment and Experiment Setup

To benchmark our applications, we used a server with RedHat Enterprise Linux 6, 64-bit, with 14GB RAM, and a single Intel Xeon E5530 (8MB cache) Quad Core 2.4GHz CPU. Java version OpenJDK 1.7.0-ea and Esper version 4.0.0 was used to conduct the experiments.

Since the ZAPREPORTER was not generating events at the desired rate at the time of experimentation, we wanted to verify if our implementations could sustain the expected traffic volume. Therefore, we built a test framework, in which we process a log file containing zap data from one day (January 31), carrying a total of 2,117,897 zap events. We measure the throughput obtained and memory usage while processing this file. The throughput is measured in four ways:

1. By reading the entire file into memory before processing it from memory
2. By reading the file line-by-line from disk
3. By receiving the events over UDP
4. By receiving the events via the HornetQ message bus

The reason for running both experiment 1 and 2 was to reveal whether the performance bottleneck is I/O or CPU bound.

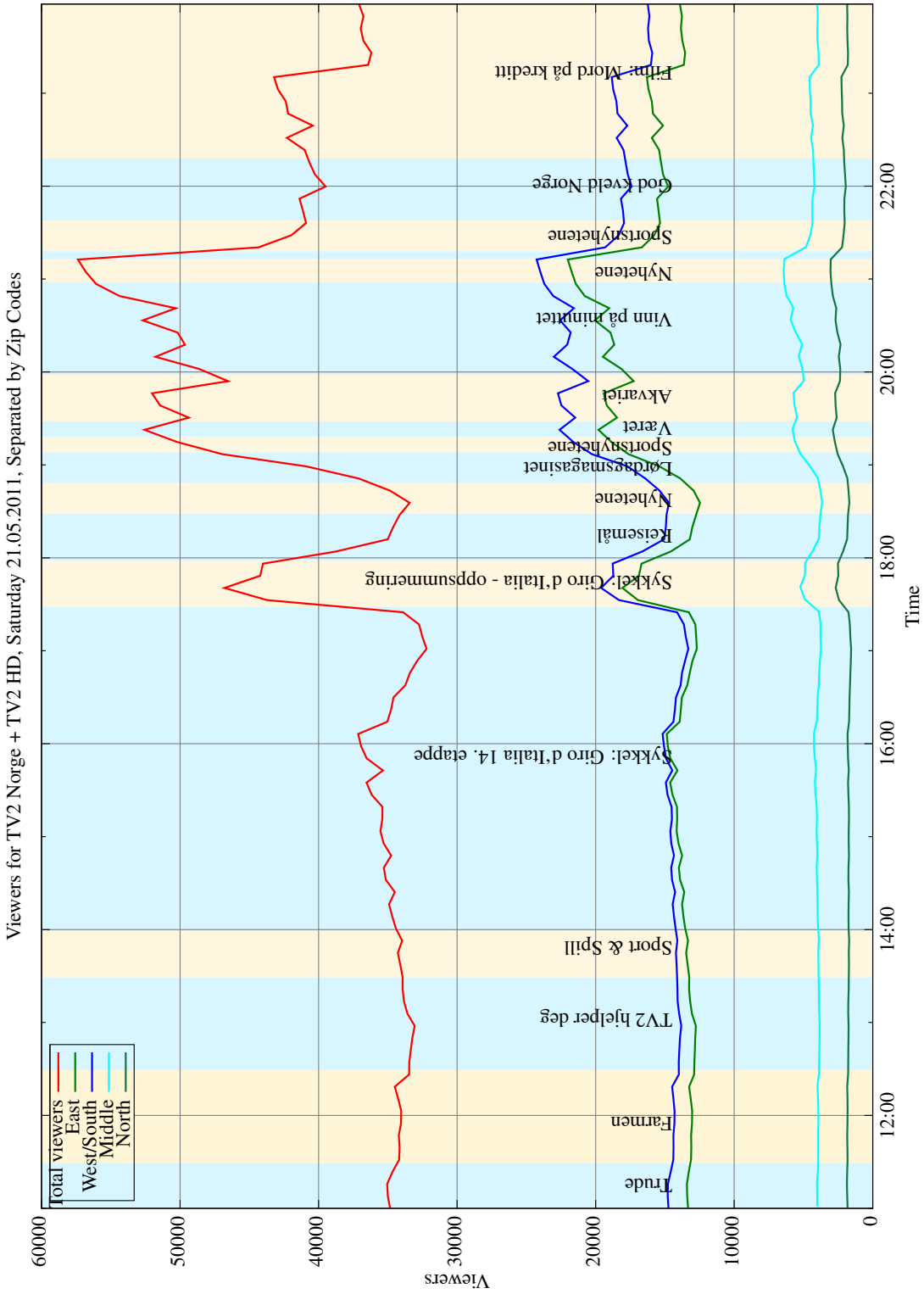

Figure 6.7: Viewer numbers for TV2, with program schedule.

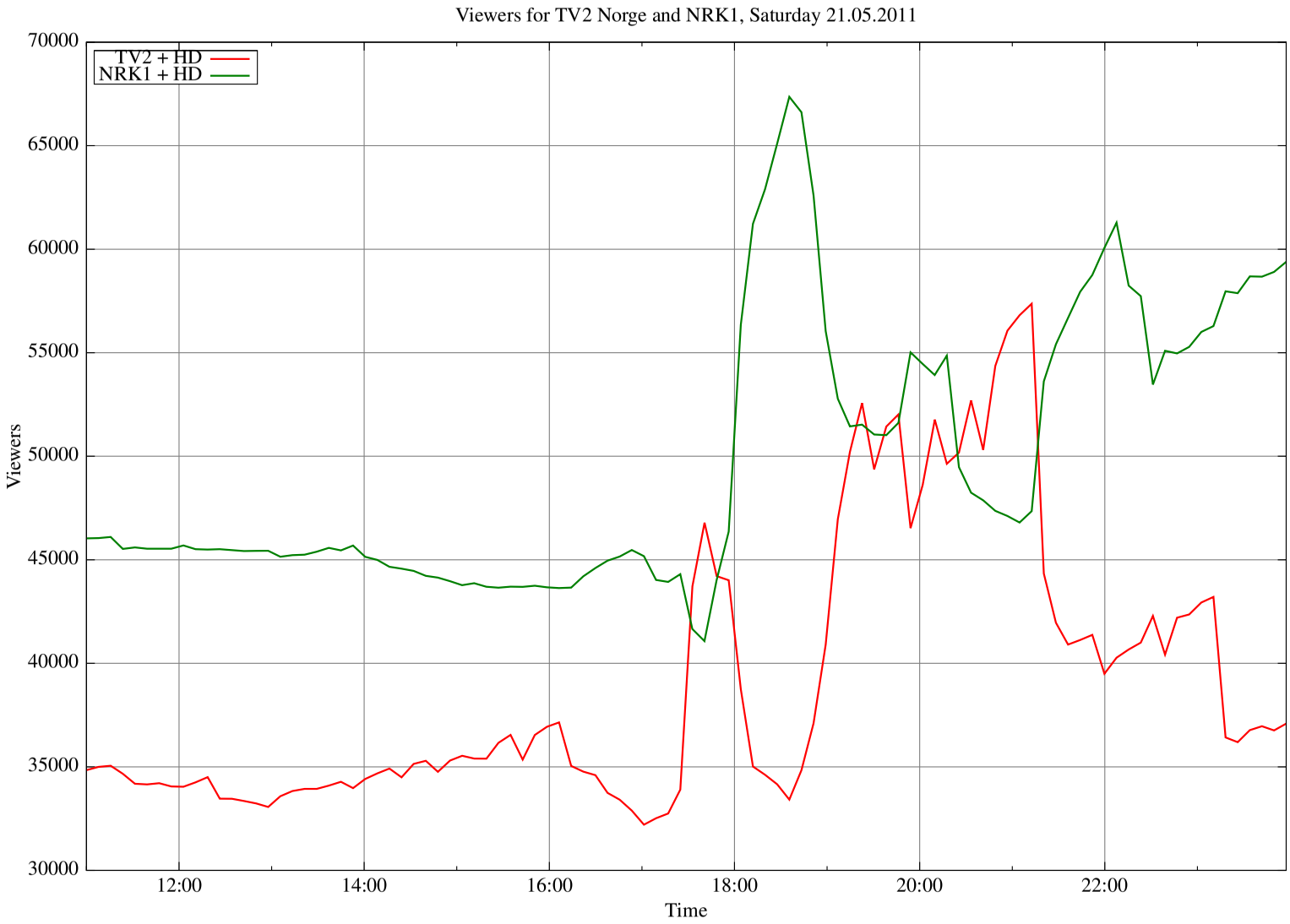

Figure 6.8: Viewer numbers for TV2 and NRK1.

Each experiment was repeated 11 times, allowing one iteration for the Java hotspot compiler to optimize the code. The experiment results are presented as the average over ten iterations of each test, as shown in Figures 6.9 and 6.10. The results were validated by comparing the final state of both Java and EPL implementations, as they should end up with the exact same number of viewers per channel, and number of STBs observed after a completed run.

VisualVM v1.3.2 with a tracer plugin for collecting heap memory usage, was used to measure memory consumption. VisualVM only supported a sample rate of 1 Hz, so the precision was limited, but nonetheless gave an overall impression of the memory consumption of the two implementations.

Results

As seen in Figure 6.9, the native Java implementation outperforms the EPL implementation by a very large margin, with an average throughput surpassing 700,000 events per second compared to an average of only 64,275 events for the EPL version with the in-memory tests. Similar results are observed for the from-disk tests. We believe this can be accredited to the flexibility offered by a general-purpose language like Java to express and optimize data structures for the specific problem at hand. Relying only on pure EPL code to express complicated queries seems to hurt performance in a significant way. This can probably attributed to the fact that EPL provide general constructs for event processing, while Java can cut corners and optimize.

Another interesting observation is the negligible performance hit on both implementations introduced by reading the events from disk instead of memory, indicating that the performance bottleneck is CPU-bound. By looking at Figure 6.9, it is also clear that receiving events over UDP introduces a significant performance penalty, reducing throughput by approximately 90 % for the Java implementation, from an average of 641,112 events per second (from-disk) to 63,515 events per second (UDP). Using the HornetQ message bus for event passing, a further performance hit is observed, to 22,546 events per second, or only 3.5 % of the throughput compared to reading the events from disk. For the EPL version, the throughput drops from 62,846 to 34,146 events per sec-

ond (46 % reduction) over UDP, and to 12,623 events per second when using HornetQ.

Although the performance hit on the Java application seems significant for the UDP and HornetQ cases, it still offers roughly 45 % higher throughput compared to the corresponding EPL versions. Moreover, the observed CPU load during the experiments was significantly lower with the Java version.

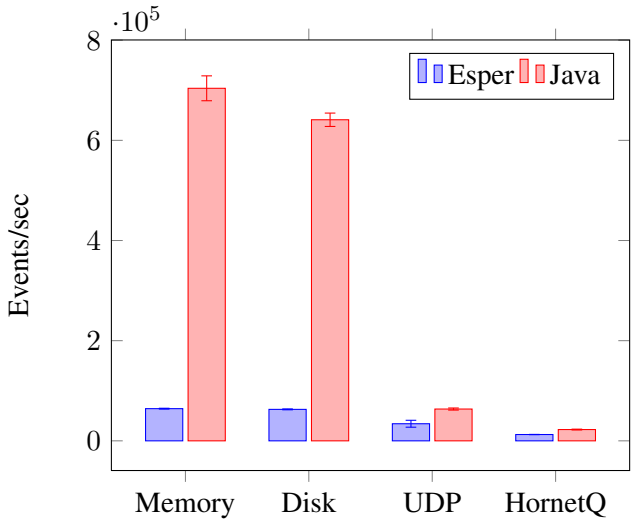

Figure 6.9: Average throughput over 10 runs.

The error bars in Figure 6.9 represent the standard deviation for each experiment. In the UDP experiments, the average packet drop for the Java version was 0.16 %, and 0.3 % for Esper. No packets were dropped by HornetQ.

Figure 6.10 once again shows the efficiency of the Java implementation. The average heap memory consumption of the Esper implementation is almost three times more than its Java counterpart, while it seems to confirm the negligible difference in performance between reading the events from disk versus loading them into RAM before processing. The negligible difference in performance between reading the events from disk suggests that performance is limited by the processing of events, and not by I/O.

Error bars, indicating standard deviation was omitted from this figure, because of the way the experiment was conducted: The heap memory usage was

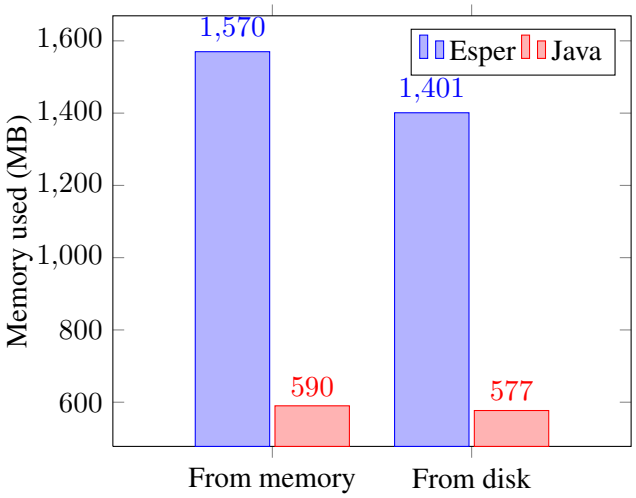

Figure 6.10: Average heap memory consumption over 10 runs.

measured on a per-second basis over the course of ten repetitions, with the numbers presented in Figure 6.10 referring to the average value of all of the samples. Because of the way the memory is handled in the JVM, where a garbage collector frees up memory at irregular intervals, we will normally observe significant fluctuations in heap memory usage that is not directly related to the operation of the application itself. Hence, the standard deviation for this type of measurement will be adversely affected by the Java garbage collector.

6.5.3 Software Complexity

Software complexity is in general an equally important evaluation criteria to performance, when comparing the different approaches. Simpler code amounts to more robust and maintainable software [73], while the performance of hardware increases steadily. Therefore, we also evaluate our rather simple code examples using Halstead's software complexity metric along with a subjective discussion.

Complexity is measured using Halstead's formula [73, 133, 148, 4], that, when applied to the number of operators and operands in a program, is said to predict the following attributes of the program:

- Length, volume, difficulty, and level of abstraction

- Effort and time required for development
- Number of faults

Predicting something that has already occurred is obviously self-contradictory, as the program must be developed before the number of operands and operators can be counted. The first two bullets are therefore in practice only used to validate the theory, and to give a metric of the complexity of a program, which is how it will be used in this evaluation.

There has been some dispute [71] regarding the usefulness and predictive powers of the Halstead metrics, and it could also be argued that the validity of these metrics are limited when applied to modern day object-oriented programming languages like Java, as they were conceptualized in an era of procedural languages. Nevertheless, we will include the non-predictive metrics, since these, together with total lines of source code, hopefully can give us some objective insight regarding the scope and complexity level of the implementations.

Originally, we used a software tool to automatically compute the Halstead metrics of the Java implementation. However, since we were unable find a tool that can compute the metrics for both Java and EPL, and because there are no universal consensus on the exact way of counting operators and operands in a given block of code [5], it was decided to calculate them manually instead, in order to ensure that the counting strategy is consistent between the two implementations.

Li et al. [37] addresses some of the challenges involved in applying Halstead to object-oriented languages, and the essence of their findings is implemented in our own strategy for counting operators and operands. This includes ignoring import statements and package declarations, but counting everything that is necessary to express the program. Operators that are syntactical identical, but semantically different through context, are counted as different operators. Examples include the parenthesis '`()`' operator, which is counted as an operator in the case of grouping expressions, e.g. `(2+2)*4` and type casting, but not when used in methods. Furthermore, the dot operator '`.`' were ignored in package names when referring to objects, such as

`tv.ChannelZap`, and included when delimiting an operator from an operand, as in `ZapWindow.std:unique()`. The colon operator ':' is also ignored in cases like this, when used to reference methods from package names, but included in statements like: `fields.hasNext() ? fields.next() : "OFF";`

Because Halstead's metric is designed to measure algorithms as opposed to complete programs [133], the metrics were calculated on class level in the Java implementation and subsequently summed together.

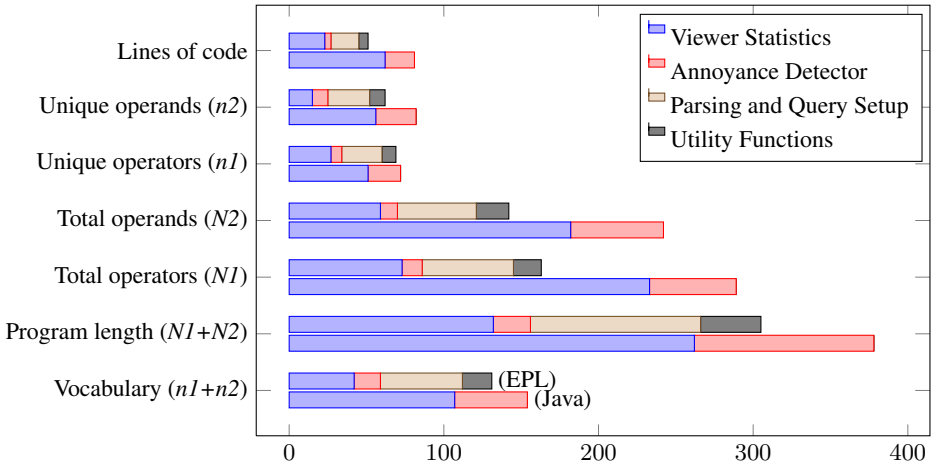

Figure 6.11: Complexity metrics breakdown per function for EPL (upper bar) and Java (lower bar).

	Java		Esper + Parsing and setup (Java)		
Metric	**Viewers**	**Annoy**	**Viewers**	**Annoy**	**Parsing**
Source lines of code	62	19	23	4	24
Prog. length (*N1*+*N2*)	262	116	132	24	149
Unique operators (*n1*)	51	21	27	7	35
Unique operands (*n2*)	56	26	15	10	37
Total operators (*N1*)	233	56	73	13	77
Total operands (*N2*)	182	60	59	11	72
Vocabulary (*n1*+*n2*)	107	47	42	17	72

Figure 6.12: The underlying complexity metrics for Figure 6.11

Figure 6.11 gives a break down per function for both EPL (upper bar) and Java (lower bar) implementations, while Figure 6.12 provides the underlying numbers. The bar chart should be read as follows: The metric for the viewer statistics is shown to the left, followed by the metric for the annoyance detector application. In the case of Java, these are the only metrics necessary to represent both applications; event parsing is included in the code for the viewer statistics application. For the EPL implementation, we also include metrics for the additional Java code necessary for parsing, Esper setup, and a custom utility function for calculating percentage. These are in addition to the query language itself. For both Java and EPL, the annoyance detector application builds upon the viewer statistics application, thus the numbers for the former includes the code from the latter.

On reading these metrics, it should be noted that the EPL implementation was done by a novice EPL programmer, and more efficient implementations might be possible.

The EPL implementation scores slightly better in all of the complexity metrics for the viewer statistics application, and significantly better for the annoyance detector. We do not find the difference in score between the two viewer statistics implementations wide enough to draw the conclusion that one is easier to develop than the other. However, upon expanding the basic viewer statistics application with annoyance detection capabilities, the additional programming effort required for expanding the Esper implementation (four lines of EPL) is significantly smaller than for the Java version (19 additional lines of code). The observed program length numbers points in the same direction, with an added program length of 116 versus only 24 for the EPL version. These observations are supported by the findings presented in Appendix A, where the complexity of two similar event processing applications implemented in Cayuga Event Language (CEL) and Java is compared.

One aspect of complexity, not covered by the software metrics, is the challenge of learning and understanding a new query language such as EPL or CEL. Although prior knowledge of SQL, possessed by many programmers, will be of great aid to this task. One concern in terms of using EPL for our applica-

tions is that we still had to write Java code to interface with other application code. Although, this interface code was minor in our case, it is easy to imagine having to write substantial amounts of wrapper/interface code outside of EPL for a variety reasons. Hence, it is obviously a disadvantage having to know and use two languages in order to develop an application. And another disadvantage with any declarative language is that we lose type-safety, an important software engineering principle for building robust applications.

Based on these observations, it is tempting to draw the conclusion that a general-purpose language is the most efficient tool for doing event stream processing. However, although it is the most effective implementation for the presented application in this case, dedicated event processing languages seems to gain efficiency relative to general-purpose languages upon expansion of the processing tasks, as indicated by the lesser effort required to add annoyance detection capabilities. This however, assume that streams can be reused across applications.

It should also be mentioned that the Java implementation is highly tuned and optimized using specialized data structures desgined for our purpose. This optimization took a fairly long time to achieve, despite the relatively few lines of code. Even though the Java implementation has better performance, the performance of the Esper implementation is more than sufficient for the application presented here. As long as the performance requirements are met by both implementations, the additional performance offered by the Java version does not translate into any real practical value. Thus, maintainability and speed of prototyping are likely to be the deciding factors when choosing which paradigm to use. As indicated by the complexity measures presented here, EPL performs better in this regard. Finally, it could be argued that a declarative programming language offers a more natural and intuitive way of expressing event patterns than what is possible with an imperative programming language.

6.6 Conclusions

In this chapter, we have demonstrated that we are able to get much more accurate viewer statistics than with traditional methods by capitalizing on the two-way communication capabilities of IP-enabled STBs. By operating on the stream of zap events from STBs, we have been able to generate viewer statistics in two very different programming paradigms. Furthermore, our results show that the general programming paradigm outperforms the query language approach by a surprisingly wide margin for this fairly simple application scenario, while at the same time being fairly similar to its counterpart in terms of total lines of code (taking the additional required lines of Java code into account).

The debate over which paradigm to choose for a specific implementation should be about choosing the right tool for the job. If the application complexity is modest and performance requirements are high, it is probably more efficient to use a general-purpose language in most cases. If however the processing task at hand is very complex, and performance requirements are met with a more specialized language, going the query language route opens up possibilities for more effortless maintenance and expansion of the application at a later stage. It is probably wise to keep a generous performance margin in such cases, as our tests indicated that added complexity hurts performance of EPL more than its Java counterpart in applications like this, because of the limited flexibility in selecting appropriate data structures.

Chapter 7

ADSCORER: Near Real-Time Impact Analysis of Television Advertisements

Building on the viewer statistics application, presented in the previous chapter, the work presented here has been extended since it was first published at the 6th ACM International Conference on Distributed Event-Based Systems (DEBS) in July 2012 [52].

In this chapter, we present ADSCORER, a scoring system for television advertisements. Our system is based on event stream processing techniques, and can compute scores for advertisements in near real-time based on channel change events from viewer set-top boxes. Our results show that ADSCORER is capable of delivering detailed scores on a per-advertisement spot basis for a whole block of commercials, immediately after the commercial break has ended. The scores include regional breakdowns with viewer numbers and shares for each geographical region of Norway as well as national scores.

Our evaluation of ADSCORER demonstrates that it is capable of scoring numerous channels simultaneously. In our experiments, we used one machine to analyze five channels, but our system can easily scale to support hundres of channels by adding more machines.

7.1 Introduction

As described in Chapter 5, significant inaccuracies can be expected with the traditional method of collecting broadcast television viewer statistics. This inaccuracy mainly exists because of the broadcast nature of the traditional mass media model, in which media consumers are secluded from providing feedback to the broadcaster [62].

In recent years we have been shifting away from this traditional model to an Internet-based model in which media consumers are empowered with numerous additional capabilities. With this model, the audience is no longer a passive crowd of media receivers, but increasingly active participants, uploading videos on YouTube, blogging, and interacting with each other on social media platforms such as Twitter and Facebook.

Additionally, the pervasiveness of devices such as STBs with recording capabilities, smart phones and tablets has enabled people to create their own daily media schedule, where they can choose what media to consume, where and when. Thus, with this changing media landscape comes new opportunities for more accurate prediction and analysis of audience behavior and responses. However, despite the advantages of online advertisement in terms of accountability and targeting, yearly spendings on traditional television commercials is rising [95, 38].

In this chapter, we perform an online analysis of the impact of advertisements on channel change behaviors among a large population of viewers. The analysis provides a score for each individual advertisement spot. The resulting scores can be useful for numerous parties, such as TV networks, advertisers, and cable network operators, as well as the general public.

To facilitate online analysis, we have developed ADSCORER, which leverage numerous advanced technologies, including CEP, video stream content recognition, and message-oriented middleware, in order to generate an instantaneous evaluation for each advertisement spot. ADSCORER is deployed in the Altibox network, which covers more than 11% of Norway's 2.2 million households [105, 136], which is a sufficiently large and diverse sample to be sta-

tistically significant. As such, this gives us an excellent opportunity to observe how the system performs in a large-scale, real-world setting.

Algorithms for evaluating the impact of television advertisements do exist [45, 80], but to our knowledge, none of these works carries the near real-time aspect that our system provides, nor have they been deployed in a live IPTV network at any scale. Furthermore, none of these covers the complete value chain necessary to perform such calculations, which include STB clients, channel change event collection, distribution and aggregation layers for translating channel change events into statistics, detection of advertisements from the TV channel stream, and finally provide a score for each advertisement.

In Chapter 6, we have demonstrated how viewer statistics can be generated in near real-time from processing STB ZAP events, both by using a specialized event processing language (EPL) and a general purpose programming language (Java). In this chapter we focus only on extending the EPL version. The system presented here builds on the previous implementation in the following ways: It has been extended to score advertisements, and it has been embedded in a generalized CEP architecture, presented in Chapter 4.

Section 7.2 describes the overall architecture of the ADSCORER system, and how EVENTCASTER is configured to provide a *success score* for each individual advertisement during commercial breaks. In Section 7.3 we discuss the attributes that make up this success score. Section 7.4 describes the deployment scenario of the application, Section 7.5 provides a walkthrough of the EPL statements used to evaluate advertisements, and in Section 7.6 we evaluate our implementation. Finally, we present our conclusions in Section 7.7.

7.2 System Architecture

This section gives a high-level overview of the ADSCORER system architecture, which consists of the following components:

- Broadcast television network
- STB client software
- Video stream content recognition software

- Message-oriented middleware (HornetQ)
- EVENTCASTER event processing middleware (presented in Chapter 4)
- Database

Figure 7.1 illustrates how these components interact at a high level. On the left side of the figure, we have the two main event producers, the AdDetector located in our data center and a large number of STBs located in cable customers homes. The AdDetector component automatically identifies advertisement spots in the television video stream, and subsequently publishes an event. An *advertisement* is defined as a single unit of presentation and is typically 10-30 seconds in length. We define a *commercial break* to consist of one or more advertisements, and it may vary in length from 30 seconds to 6 minutes. An advertisement event, from now on referred to as *AdIdentified*, contains the following attributes:

- An identifier for the advertisement
- The channel name
- The length of the advertisement
- The time of detection
- Begin or end status for the advertisement

These events are published to a message queue, and subsequently picked up by another component, as we explain in more detail later. Events indicating the start and end of commercial breaks are inferred from the stream of advertisement events. For the rest of this chapter, we refer to these as *CommBreak* events. Additionally, the STB clients generates several event types:

- Channel change event (also called a zap event)
- HDMI status event: TV set on/off
- STB audio on/off event (mute)
- STB volume change event

These STB events are transmitted over UDP to a ZAPCOLLECTOR in our data center. The ZAPCOLLECTOR decodes the packets and places them on a message queue.

ADSCORER uses one instance of the EVENTCASTER core application, presented in Chapter 4. This instance subscribes to events generated by STBs as well as *AdIdentified* events emitted by the AdDetector, as described above. It is responsible for scoring the advertisements, according to the criteria presented in Section 7.3.

Components of the EVENTCASTER middleware are colored yellow in Figure 7.1.

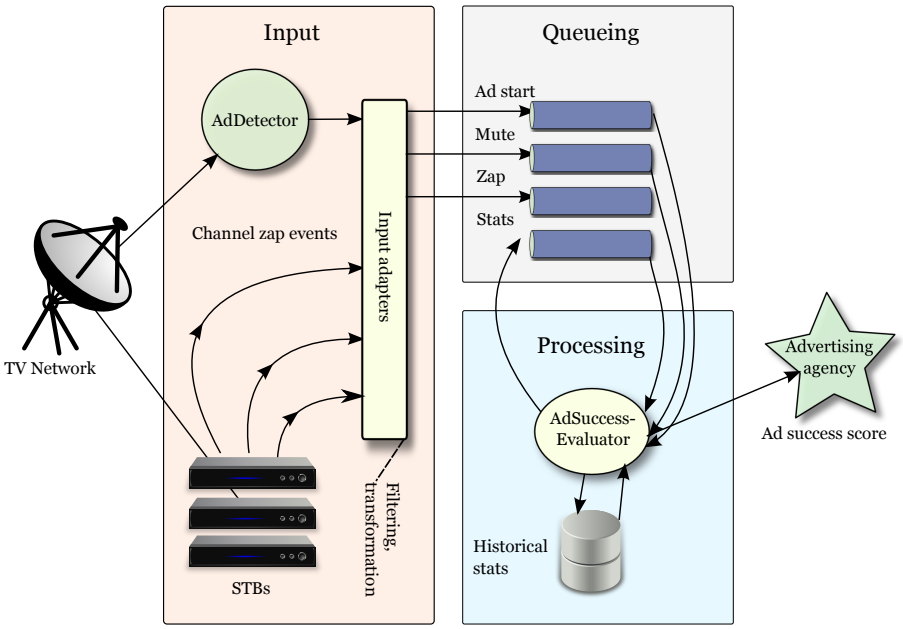

Figure 7.1: ADSCORER Architecture Overview

ADSCORER is an event-driven architecture, in which event producers and event consumers are decoupled [119]. Figure 7.2 illustrates the connection between components of the ADSCORER system and the conceptual building blocks of an EDA, presented in Section 2.5. To reiterate, the four main building blocks of an EDA are *producers*, *consumers*, *agents* and *channels* [22].

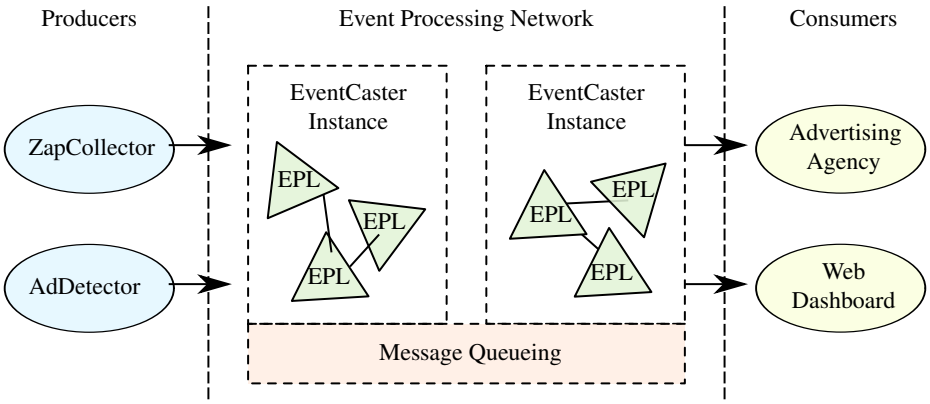

Figure 7.2: AdScorer as an EDA

In the figure, the ZAPCOLLECTOR and AdDetector entities are categorized as event producers, even though they are not the actual generators of events. The explanation for this can be found in Section 2.5.2, where we define producers as the entities that introduce raw events to the EPN.

The green triangles in Figure 7.2 represent processing agents in the form of EPL queries, using the same notation as in Figure 2.9. The channels connecting the agents are either internal inter-process communication, or message queues, depending on whether the EPL queries are located on the same EVENTCASTER instance or not.

7.3 Scoring Criteria

This section discusses the scoring criteria used in in the ADSCORER system. In our implementation, an advertisement spot is evaluated according to a wide range of criteria, as listed in Table 7.1. Examples of usage for all of the criteria listed in this table can be found in the code examples provided in Section 7.5.

These scoring criteria may be represented in both actual numbers, and additionally in percentage form. In the cases where they are related to other numbers, such as interval between number of viewers at the start and end of the advertisement, a percentage representation is more intuitive than the underlying

Criteria	Symbol
Viewers at the start of the advertisement	α
Viewers at the end of the advertisement	θ
The interval between α and θ	τ
Viewers that stayed on the channel throughout the advertisement	ϵ
Viewers that muted the sound during the advertisement	Δ
Viewers that was in mute mode when the advertisement began	γ
Viewers that unmuted the sound during the advertisement	δ
Viewers that turned on TV during the advertisement	Ω
Viewers that turned off TV during the advertisement	ω
Average volume at the start of the advertisement	Λ
Average volume at the end of the advertisement	λ
Average volume during the advertisement	κ
Initial Audience Retained (ϵ/α)	IAR

Table 7.1: Scoring Criteria

numbers. As described in Chapter 5, the IAR metric presents the fraction of viewers retained for the duration of an advertisement.

Another metric that we expect to be of interest, is the numbers of viewers that pressed the mute button during the advertisement (Δ). This metric should give further indication to whether viewers are actually watching the advertisement, as it is likely that viewers that have muted the sound, does not pay attention to the advertisement.

Furthermore, the Δ trend of individual advertisements over time, combined with the interval between the average volume at the start (Λ) and end (λ) of the advertisement can potentially reveal advertisements where the audio contains particularly high RMS levels. RMS is an abbreviation for Root-Mean-Square, and when used in relation to audio, refers to the average loudness over time. The way humans perceive loudness is to a much higher degree related to average levels than peak levels. Advertisers take advantage of this, as the technical limitations imposed upon broadcast media applies to peak levels, and not RMS levels [113].

By reducing the dynamic range of the audio, advertisers are able to increase the perceived loudness of their advertisements without exceeding the maximum

permissible peak level. In audio terms, a reduction of dynamic range is referred to as *compression*, and is the reason why the volume in commercial breaks generally appears to be louder than in the programs that preceded them. However, severe limiting of the dynamic range also introduces distortion in the audio signal as a side-effect, which further serves to annoy listeners. In fact, loud commercials has been number one complaint by television viewers to the Federal Communications Commission (FCC) in the United States over the past decade [94].

A study of RMS levels in televised advertisements, combined with scores from the ADSCORER system could help the understanding of how viewers respond to highly compressed audio in advertisements.

The criteria listed here could be combined to make up a final score, represented as a numerical value between 0 and 10. This score would say something about the impact of the advertisement, and could give advertisers an unprecedented opportunity to measure the impact for each individual advertisement spot, based on factual observations, as opposed to claimed attitudes and numbers.

7.4 Deployment

We now describe the deployment scenario at the time the measurements of this chapter were performed, as well as some additional features of the updated ZAPREPORTER, presented in Section 6.2.2. The features of the updated ZAPREPORTER is expected to facilitate significantly more accurate statistics and enable us to conduct more interesting behavioral analysis of television viewers. There are two ZAPREPORTER deployments – One that was deployed when conducting the experiments presented in this thesis, with the exception of the experiment presented in Section 7.6.2, and an improved version, which was deployed after the experiments were performed. Due to time constraints, we have not been able to repeat the experiments using data from the updated ZAPREPORTER.

The ZAPREPORTER deployed at the time the measurements in this chapter

were performed was the same as for the experiments presented in Chapter 6. Thus, it only reported channel change events where the viewer remained on the same channel for more than one minute. Unfortunately, this sampling mechanism prevented us from capturing some interesting behaviors of television viewers, such as the channel surfing behavior during commercial breaks.

A new ZAPREPORTER has been deployed, in which channel changes and other STB events are forwarded much more rapidly. The new ZAPREPORTER also forwards STB events related to mute, volume, and HDMI status on or off. The latter enables us to determine if the TV connected to the STB has been turned off. We provide further details on how the new ZAPREPORTER will be used in the current deployment below.

The AdDetector part of the scoring system, illustrated in Figure 7.1, is commercial software from a vendor that also delivers content-recognition technology to some of the major players in the media measurement industry. It is used by content creators to detect violations of copyright, as well as advertisers to measure that they are getting the exposure they have paid for.

7.4.1 Some Initial Findings with the Enhanced ZAPREPORTER

A better understanding of viewer behavior will hopefully be gained from analyzing the output of the new ZAPREPORTER, as it captures most of the channel surfers, and also expands the viewer action repertoire by including mute and volume events.

The use of HDMI status monitoring addresses the main criticism against STB-based viewer statistics, namely that most people do not turn off their STB, even though their TV is off. As such, it is impossible to determine whether there are people watching unless there is STB event activity. Being able to detect whether the TV is turned on or off, enables us to establish with great confidence whether someone is watching, as virtually everyone turns off the TV when going to bed or leaving the house.In some households, the TV may still be running in the background, while people are doing other things, but then again, the traditional methods are no better in this regard.

It is presently unclear what the impact of this flaw in our previous statis-

tics [51] and other IPTV measurements [20] will be. But we expect it to be significant, as we discuss next.

The new ZAPREPORTER was deployed to customers as a silent upgrade, which means that only those that power cycled their STB device was upgraded. Those that left their STBs on, was upgraded at a later stage, during a forced upgrade. One day after deploying the new STB software, approximately 15,000 STBs had upgraded, and after a week 80,000 STBs had upgraded. Out of a total of 320,000 STBs, these numbers seems to indicate that a large fraction of STBs are rarely powered off when the customer is not watching TV. Thus, for this reason we expect that STB-based statistics may see significant discrepancies between those that only monitor zap events and our approach that also captures HDMI status.

7.5 Implementation

The ADSCORER implementation scores televised advertisements according to the metrics presented in Section 7.3. In this section, we provide implementation details such as how components of the ADSCORER system interact, as well as code examples. We start with the component interactions, before going into a detailed code description.

7.5.1 Component Interactions

The sequence diagram in Figure 7.3 shows the interactions between the AdDetector, AdSuccessEvaluator and message queue components during the evaluation of an advertisement: An advertisement is identified by the AdDetector, which generates an *AdIdentified* event, containing an identifier, duration, channel name, and a boolean begin-property for the advertisement identified. A *producer* listens for these *AdIdentified* events, and puts them on the message queue, where it is picked up by an EVENTCASTER instance, configured to calculate advertisement scores. This EVENTCASTER instance is named AdSuccessEvaluator in Figure 7.1.

When an *AdIdentified* event is received by the AdSuccessEvaluator, it starts

to collect statistics about the STBs tuned to the channel of the advertisement. These statistics includes viewer numbers, number of muted STBs, and average volume among the STBs. The AdSuccessEvaluator collects these events until it receives another *AdIdentified* event containing the same advertisement identifier and channel name, with the begin property set to false to indicate the end of the advertisement. Alternatively, if the duration of the advertisement, included in the first *AdIdentified* event elapses, an *AdIdentified* event with the begin property set to false is generated by the AdSuccessEvaluator itself. When one of these two conditions are met, AdSuccessEvaluator calculates a final score, containing the scoring criteria presented in Section 7.3.

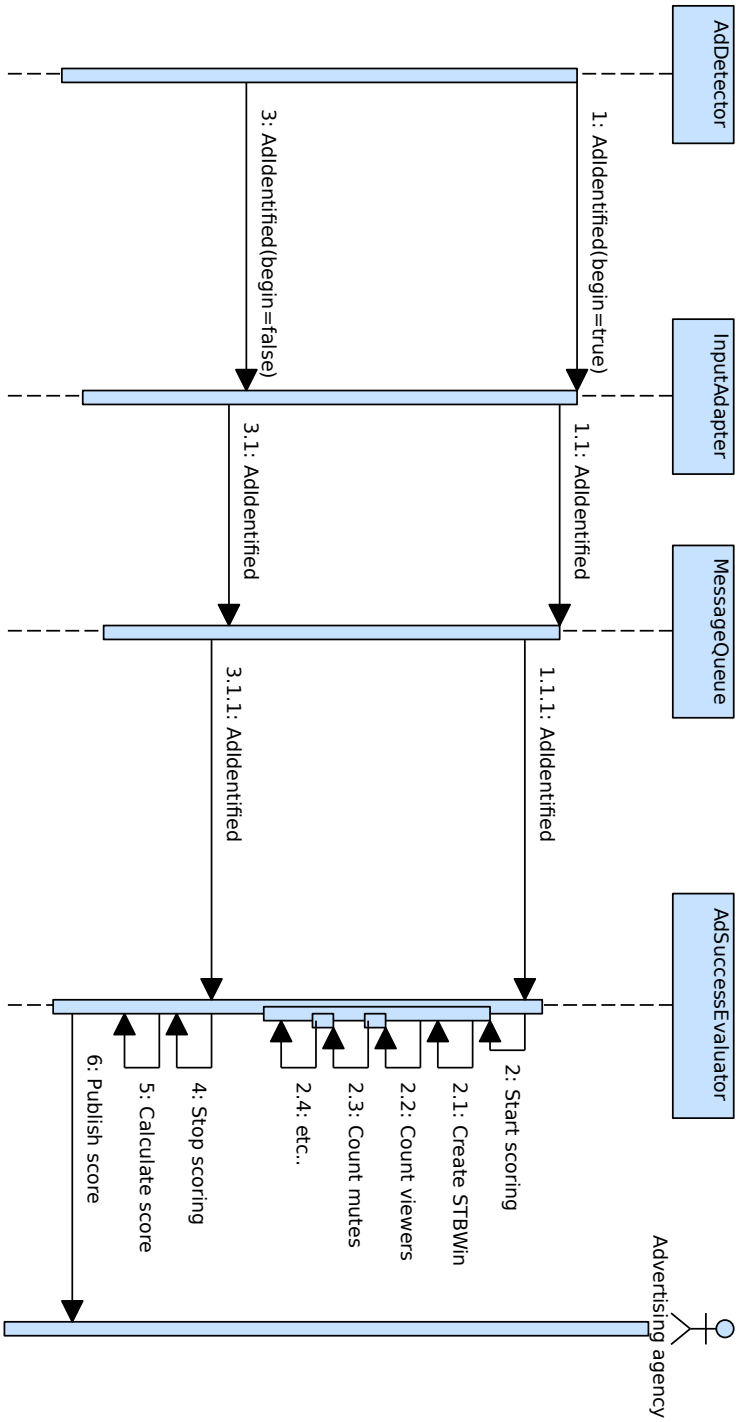

Figure 7.3: Ad Scoring Sequence Diagram

7.5.2 ADSCORER EPL Code

We now describe the EPL statements essential to the ADSCORER application. The code examples listed here, runs in the Esper engine of the AdSuccess-Evaluator instance, illustrated in Figure 7.1. It receives events generated by the AdDetector system.

Listing 7.1 defines a *context* named *AdBreakCtx*, that lasts for the duration of an advertisement. A context is an EPL abstraction that is created when one or more conditions are met, and destroyed when another set of conditions are met. Memory used by objects associated with the context are immediately and automatically released when the context ends.

Listing 7.1 EPL AdScorer per-ad context definition

```
1 create context AdBreakCtx as
2   initiated by
3     tv.AdIdentified(begin=true) as ad
4   terminated by
5     tv.AdIdentified(detectionId=ad.detectionId,
6       begin=false) as endAd
```

In this context, we create a data window, containing all the STBs tuned to the channel of the advertisement when the advertisement starts. Listing 7.2 creates and populates this window, named *STBWin*. As can be observed from the query, only STBs tuned to the channel of the advertisement, having an active HDMI connection in unmuted mode are included, as we can assume that the remaining STBs are not associated with any viewers.

Listing 7.2 EPL AdScorer per-ad STB window

```
1 context AdBreakCtx
2 create window STBsnapshots.win:keepall() as
3   STBWin
4    insert where
5      channel in (context.ad.channel)
6        and hdmi=true and mute=false
```

Viewers that leave the channel during the advertisement are inserted into an event stream named *Dropout* by the statement included in Listing 7.3. The

statement included in Listing 7.4 subscribes to the *Dropout* event stream, and removes these STBs from the *STBWin* data window. There is no corresponding insert statement for keeping track of viewers that arrives at the channel after the advertisement has started. This is because we are only interested in the details of those viewers that watched the whole advertisement.

Listing 7.3 EPL AdScorer per-ad dropout insert

```
1 context AdBreakCtx
2 insert into Dropout
3   select * from tv.ChannelZap(fromChannel
4     in (context.ad.channel)
```

Listing 7.4 EPL AdScorer per-ad dropout remove from main STB window

```
1 context AdBreakCtx
2 on Dropout d
3   delete from STBsnapshots s
4 where s.ip = d.ip
```

Every time the number of STBs in *STBWin* changes, the statement in Listing 7.5 inserts the updated number into an event stream named *ViewersCount*. As previously mentioned, the number of STBs in *STBWin* are only reduced during an advertisement: Viewers that arrives after the start of the advertisement are ignored. The first and last value from *ViewersCount* in Listing 7.6 to calculate the total number of viewers lost during the advertisement.

Listing 7.5 EPL AdScorer per-ad viewer dropout count

```
1 context AdBreakCtx
2 insert into ViewersCount
3   select count(*) as n
4 from STBsnapshots
```

The statement shown in Listing 7.7 keeps track of viewers that mutes the sound during the advertisement. The content of this statement is only published on termination of its associated context. Listing 7.8 continuously calculates the average volume for the STBs in the *STBWin*, and inserts these values into an event stream named *AvgVol*. *AvgVol* is subscribed to by the statement shown in

Listing 7.6 EPL AdScorer per-ad viewers lost

```
1  context AdBreakCtx
2  insert into ViewersLost
3  select
4    first(n) as vBegin,
5    last(n) as vRetained,
6    context.ad.adId as adId,
7    context.ad.channel as channel,
8    context.ad.time as startTime,
9    context.endAd.time as stopTime
10 from ViewersCount.win:keepall()
11   output snapshot when terminated
```

Listing 7.9, which extracts the first and last events from *AvgVol*, and calculates the overall average for all the *AvgVol* events published during the advertisement.

Listing 7.7 EPL AdScorer per-ad mute count

```
1  context AdBreakCtx
2  insert into MuteCount
3    select count(*) as mutes
4      from tv.Mute(mute=true) m
5    where
6      exists(select * from STBsnapshots
7        where ip = m.ip)
8  output snapshot when terminated
```

Listing 7.8 EPL AdScorer per-ad average volume calculation

```
1  context AdBreakCtx
2  insert into AvgVol
3    select avg(volume) as avgVol
4      from STBsnapshots
```

Listing 7.10 aggregates the various statistics collected during the advertisement, and forwards them into an event stream named *AdSummary*. The content of *AdSummary* is processed by the statement shown in Listing 7.11, which shows the EPL query for collecting and generating statistics on a per-advertisement basis.

The simplicity of the EPL language is shown in Listing 7.12, which is the query for collecting all *AdStat* events for a television channel between two

CommBreak events. *CommBreak* events indicate the start and end of commercial breaks. *AdIdentified* events can only occur between a *CommBreak (begin=true)* and *CommBreak (begin=false)* event for the same channel. An *UpdateListener*, implemented in Java, then publishes the result back to the message queue. The query in Listing 7.12 is also subscribed to by another *UpdateListener*, named *EmailPublisher*, that publishes events via email.

Listing 7.9 EPL AdScorer per-ad volume summary

```
1  context AdBreakCtx
2  insert into VolSummary
3    select
4      avg(avgVol) as average,
5      first(avgVol) as startVol,
6      last(avgVol) as endVol
7    from AvgVol.win:keepall()
8  output snapshot when terminated
```

Listing 7.10 EPL AdScorer ad summary

```
1  insert into AdSummary
2    select * from pattern [ every
3      (a=ViewersLost and b=MuteCount and c=VolSummary)]
```

Listing 7.11 EPL AdScorer per-ad query

```
1  insert into tv.AdStat
2    select
3      s.a.adId as adId,
4      s.a.channel as channel,
5      s.a.startTime as startTime,
6      s.a.stopTime as stopTime,
7      s.a.vBegin as viewersBegin,
8      s.a.vRetained as retained,
9      s.b.mutes as mutes,
10     percent(s.a.vRetained, s.a.vBegin) as iar,
11     (s.a.vBegin - s.a.vRetained) as lost,
12     roundDouble(s.c.average) as vol,
13     roundDouble(s.c.startVol) as startVol,
14     roundDouble(s.c.endVol) as endVol
15   from AdSummary s
```

Listing 7.12 EPL AdScorer whole break query

```
1 select * from pattern [
2  every
3   a=tv.CommBreak(begin=true)
4    -> b=tv.AdStat(channel=a.channel)
5   until c=tv.CommBreak(begin=false, channel=a.channel)
6 ]
```

7.6 Evaluation

In this section we describe some of the experiments that we have conducted with ADSCORER. We present both a performance evaluation and some interesting observations of viewer behavior derived from the ADSCORER system.

7.6.1 Environment and Experiment Setup

For the experiments, we obtained 1.5 hours of prime time broadcast television sampled from the largest commercial networks, starting at 18:45 on a Thursday evening. Before running the experiments, STB data from the 23 preceding days were used to initialize the system. This ensured that the Esper engine had the correct state when conducting the experiment.

Recorded video streams and STB data was used in order to be able to debug and verify system correctness, rather than operating on live video streams and STB data. Moreover, due to time constraints and lack of appropriate video editing tools, the experiments was conducted by simulating the output from the AdDetector system, using manually recorded timestamps and advertisement IDs. However, fingerprints were made from each commercial in one of the commercial breaks of the recordings, using the AdDetector system, and it was verified that the system successfully detected each of them within two seconds when streaming the broadcast recording to the AdDetector system.

The experiments involved three servers in addition to the database server keeping track of the state of the channel statistics. One server was designated event producer, simulating channel zaps obtained from STBs, and advertisement identifications obtained from the AdDetector system. Another server were running the message bus, and a third server were running the *core* application,

configured to score advertisements, as illustrated in the processing section of Figure 7.1.

Performing these experiments was a time-consuming task, and for this reason, the experiments were only repeated three times. When reviewing the results, it became apparent that there had been made a timing-error in the last iteration, leaving us with the results of only two iterations. Even though this is not a statistically significant amount of repetitions, it nonetheless serves as a proof of concept of the scoring capabilities of the ADSCORER system.

Consistency of advertisement scoring results were verified by comparing the results of the two iterations, ensuring that the values were not significantly different from one run to another. The biggest observed deviation was a difference of 12 viewers for the first advertisement starting at 18:48:30 on TV2 Norge: 34351 retained viewers was logged for the first run, while this number had increased to 34363 viewers in the second run – a 0.000349335% variation. For the rest of the advertisements, the results were for the most part identical between runs. The few additional variations observed were smaller than the one mentioned here.

The variations in scores can be attributed to the distributed nature of the system, combined with the event stream processing techniques used, where variations in network and processing latency might lead to slightly different states. Because *AdIdentified* events are generated on a different machine than the one generating ZAP events, small variations in viewer numbers, as described above are likely to occur. Small variations in timing of arrived *AdIdentified* events between runs can easily produce the kind of variations observed in this experiment, considering the high rate of events sent to the AdSuccessEvaluator server, with ZAP events arriving every millisecond.

7.6.2 Viewer Statistics During Commercial Breaks

We now discuss the results of our advertisement scoring experiments, and present some general viewer statistics for the measurement period.

Table 7.2 lists the time and duration for the commercial breaks that occurred on the commercial channels TV2 Norge and TVN during the sample period for

Time	Channel	Duration
18:48:30 - 18:50:30	TV2 Norge	2 minutes
18:53:00 - 18:59:00	TVN	6 minutes
18:56:30 - 18:59:30	TV2 Norge	3 minutes
19:23:00 - 19:29:00	TVN	6 minutes
19:23:30 - 19:29:30	TV2 Norge	6 minutes
19:51:00 - 19:57:00	TVN	6 minutes
19:54:30 - 19:59:30	TV2 Norge	5 minutes

Table 7.2: Commercial breaks

Figures 7.4, 7.5, 7.6, 7.7 and 7.8, described in the previous section. Figure 7.4 shows the stacked numbers of viewers for the channels NRK1, TV2 Norge and TVN. Figure 7.5 shows the actual viewer numbers for the same data set, while Figure 7.6 shows the actual viewer numbers for TVN, the smallest of the three channels.

Although most of the commercial breaks listed in Table 7.2 are clearly visible in some of the viewer plots, such as Figures 7.5 and 7.6, the drops in viewer numbers are not nearly as significant as we had expected. Some of this may be attributed to the lack of resolution on STB data, preventing us from accurately capturing channel surfers, as explained in Section 7.4. Moreover, we can clearly see from Figure 7.5 that there is a close to linear growth in viewer numbers over the entire measured interval, except for the significant drop on TV2 at 19:54, and similarly on NRK1 at roughly 20:00. We note that NRK1 is a non-commercial TV channel, and the largest in Norway. The steady growth of viewers during the measured interval means that the actual number of lost viewers during the commercial breaks are higher than it might appear from viewing Figures 7.4, 7.5 and 7.6, which only shows actual viewership.

Figure 7.7 illustrates the retained number of viewers for each advertisement in the commercial break that started at 18:56:30 on TV2 Norge, divided into regions. The difference in viewer numbers between regions, for the most part, reflects geographical variations in the number of deployed STBs in the Altibox network. However, there are some relative differences as well, illustrated in Figure 7.8, where the regional shares of one of the advertisements presented in

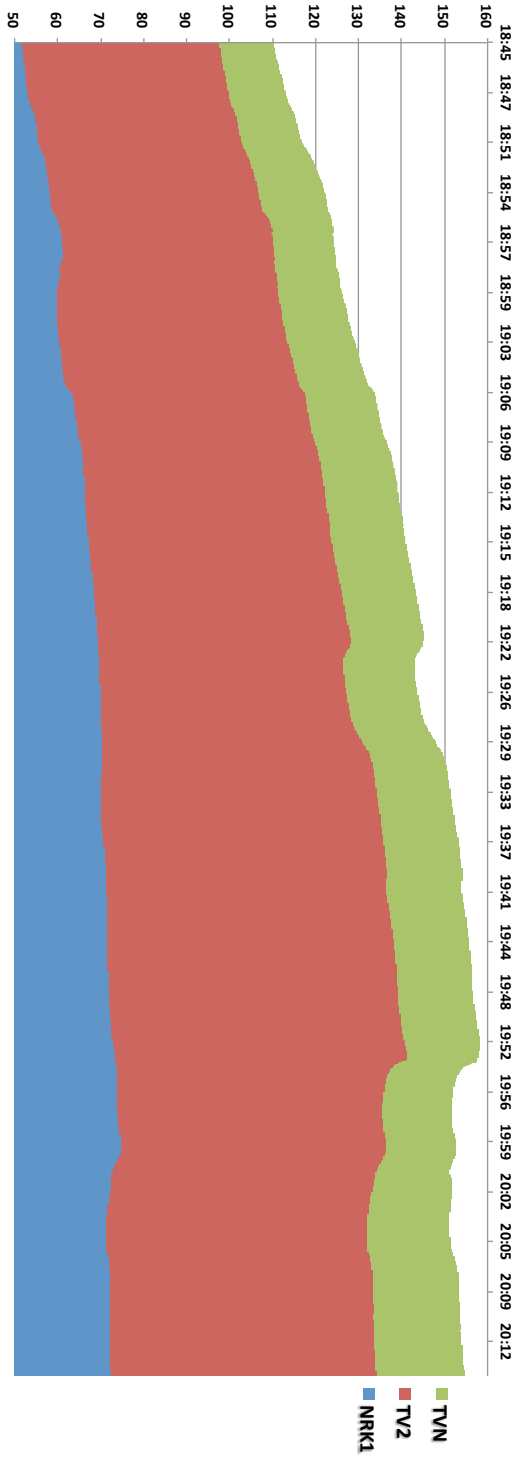

Figure 7.4: Stacked viewership (in thousands) for the three largest channels, NRK1, TV2, and TVN.

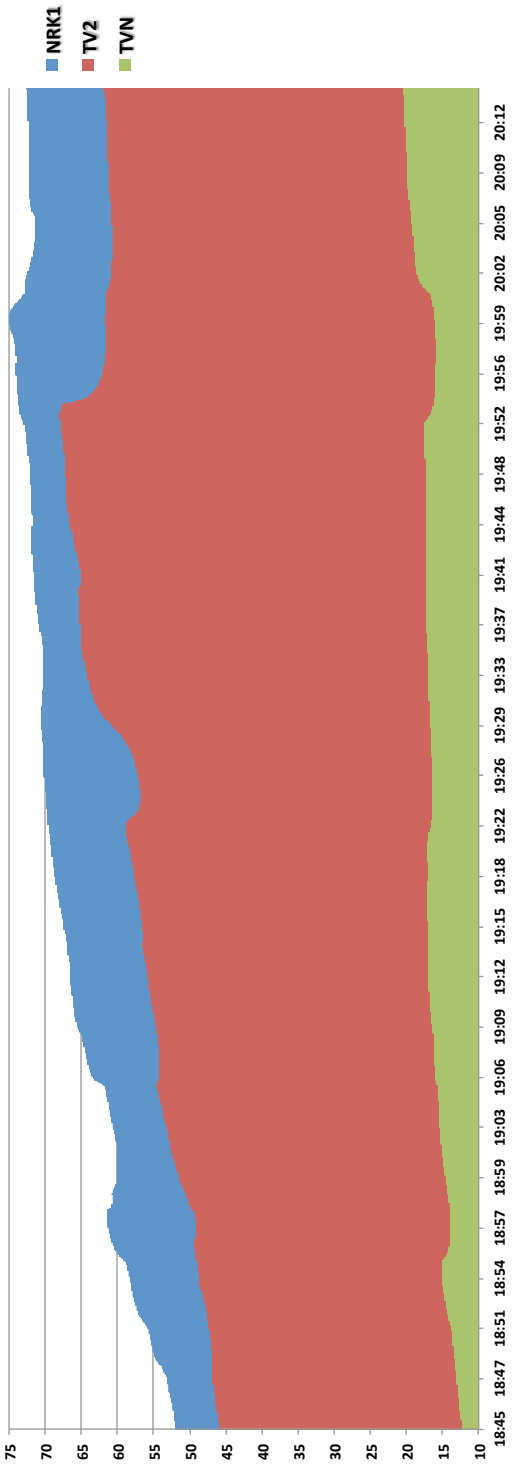

Figure 7.5: Actual viewership (in thousands) for each of the three largest channels, NRK1, TV2, and TVN.

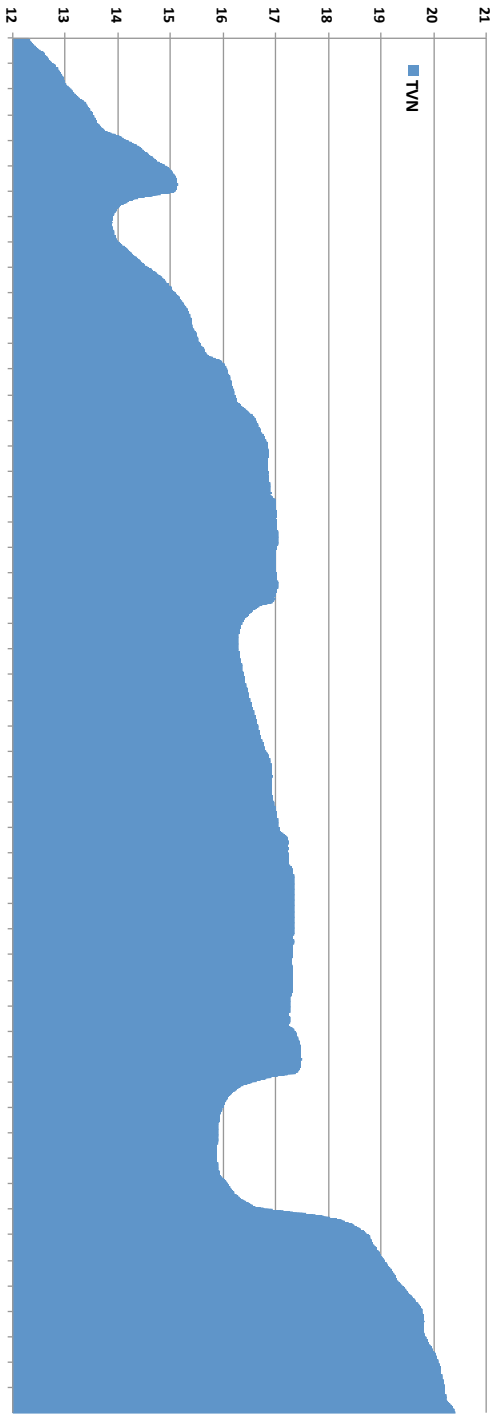

Figure 7.6: Actual viewership (in thousands) for TVN annotated with Start and End of commercial break.

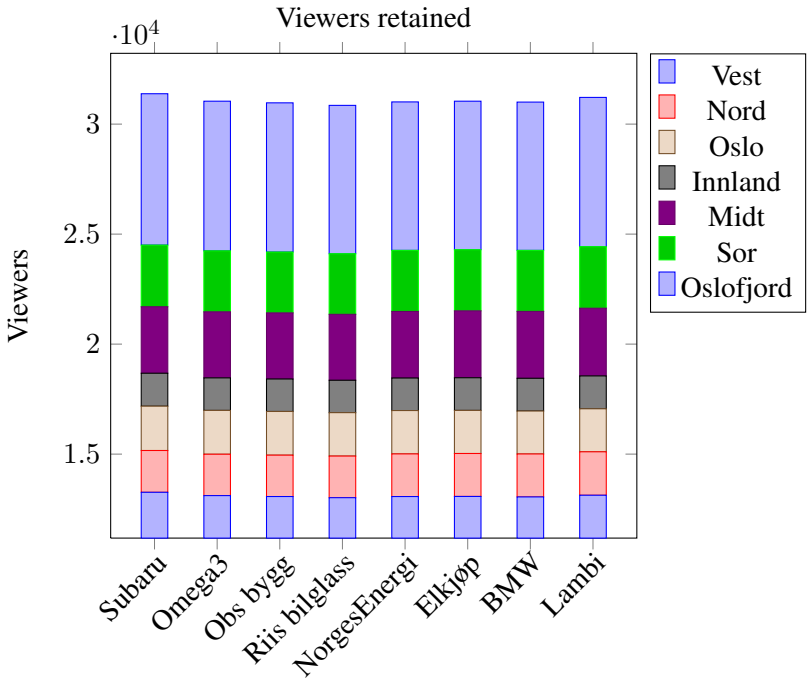

Figure 7.7: Viewers retained for several commercial spots, split into regions

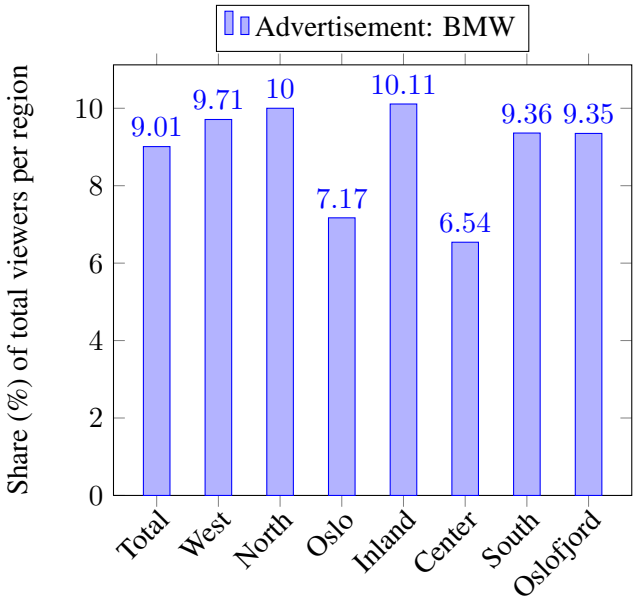

Figure 7.8: Regional viewer shares for a single advertisement

Figure 7.7 are displayed. From this particular BMW advertisement, we can see that there is a relatively low viewer share of retained viewers for Oslo and the center part of Norway than the rest of the country.

We can think of a number of reasons for this: One explanation could be that TV2 Norge generally has a smaller viewer share in these parts of Norway, another reason could be that the BMW car manufacturer has less brand recognition here. A third reason may be that people in Oslo and the center part of Norway are more likely to change the channel during a commercial break. These are just speculations, but could the subject of further research.

Live-Data Scoring Results

The following experiment was performed at a later time than the previous ones, and is the only experiment presented in this thesis that was performed after the deployment of the improved ZAPREPORTER, discussed in Section 7.4.1. It was performed with live STB data, in a prime-time commercial break on TV2 Norge, which is the largest commercial channel in Norway.

Figure 7.9 shows how many viewers were lost and retained for a commercial break between the evening news and sport news on TV2 Norge. The advertisements are listed on the x-axis in chronological order, as they appeared, not reflecting the duration of each advertisement. As is evident from the chart, viewer drop is quite significant in the beginning of the break, and it continues to drop well beyond the middle of the break. We imagine that advertisers would be very interested in having access to this type of data, in order to influence the ordering of the advertisement spots within a commercial break. What Figure 7.9 does not show is the viewer numbers after the commercial break. In this case, the commercial break was followed by two minutes of advertisements for scheduled programs on the same network.

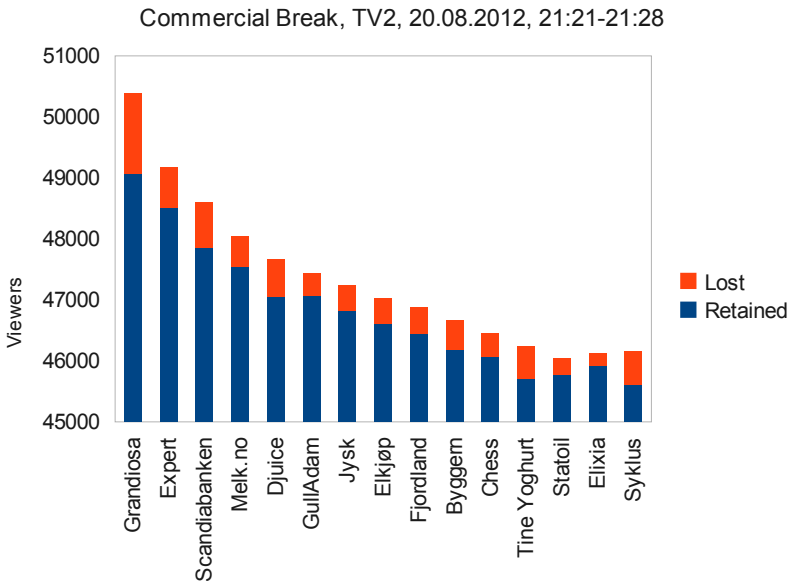

Figure 7.9: Retained viewers per ad-spot for a prime-time commercial break.

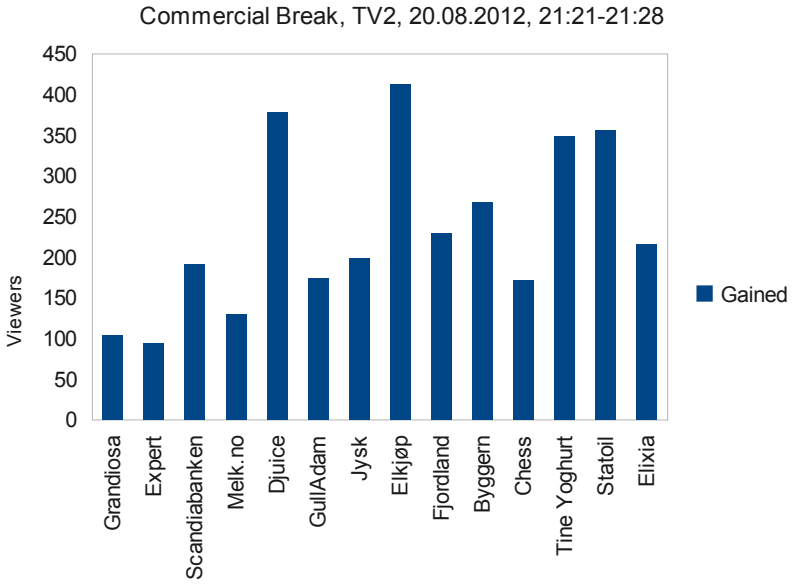

Figure 7.10: New viewers per ad-spot for a prime-time commercial break.

Figure 7.10 shows the number of viewers that arrived at the channel during each advertisement. The numbers were calculated by subtracting the number of retained viewers of the preceding advertisement from the starting number of viewers for the following advertisement. The number of new arrivals include channel surfers that stayed on the channel for 10 seconds or more, before leaving the channel again.

With this method, there is a natural tendency for more viewers to arrive during longer advertisements, such as “Elkjøp”, which lasted for 30 seconds, compared to short advertisements such as “Expert”, having a duration of 15 seconds. However, the difference between the two advertisements mentioned is more than double, with 94 arrivals for the “Expert“ advertisement, versus 412 arrivals for ”Elkjøp“. We believe this can be partly attributed to the fact that the ”Expert“ advertisement is located at the beginning of the break, while the ”Elkjøp“ advertisement is located in the middle, as will be discussed in the following paragraph.

A plausible explanation for the low number of arrived viewers in the first advertisements in the break relative to the later ones is that people switch away from the channel when the break starts, channel surfing on other channels in the beginning, and switches back after a while, to check if the next program has started. This theory is supported by the rising trend of arriving viewers as the commercial break progresses, which can be observed in Figure 7.10. Note that, even though there is an increasing number of arriving viewers as the break progresses, there is still more viewers leaving the channel, so the net result is still a trend of viewers leaving the channel.

The last advertisement in the commercial break is omitted from this chart, as we did not record viewer data after the break had ended for this experiment, which makes it impossible to calculate the number of arrived viewers for this last advertisement.

7.6.3 Advertisement Scoring Capacity

To understand the system’s ability to handle multiple channels simultaneously, we ran tests by synthetically generating *CommBreak* and *AdIdentified* events

for five different channels at the same time, while the system was receiving live channel zap events.

System load was not significantly affected during this experiment. The resulting output files appeared to be correct for the time period sampled, although it was not possible to repeat the experiment with identical values, as the system was operating with live STB data for this particular experiment.

7.7 Conclusions

We have demonstrated a new way of scoring television advertisements that is more in line with current measurement methods for online media than what is the current practice in the media industry, and more suited for the new models of media consumption.

Furthermore, the ADSCORER system provides a proof-of-concept for the EVENTCASTER middleware, proving its capabilities as a platform for building event processing applications.

Our results indicate that our implementation is capable of scoring advertisements on multiple channels simultaneously in near real-time with consistent results, and that event processing is an effective tool for achieving this. Furthermore, ADSCORER is capable of delivering an unprecedented level of detail, not possible through the current measurement regime.

We are already in the process of developing a graphical front end that operates on live data and displays the scoring results in near real-time. With this we intend to conduct more detailed viewer behavior analysis in order to derive an improved understanding of the media and to use this understanding to devise new service offerings.

Chapter 8

Conclusions

The exponential increase of data being produced by both humans and applications, combined with a trend towards ever-increasing pervasiveness of applications, entails processing demands that cannot be met by the traditional request/reply interaction model.

Publish/subscribe interactions offers better scalability, flexibility and timeliness than request/reply to the kind of pervasive, information-driven applications discussed in this thesis. However, publish/subscribe alone only allows for *stateless* subscriptions, operating on single events in isolation. While sufficient for a range of simpler tasks, more advanced applications require putting events in context. By introducing *statefulness* and *context* to the publish/subscribe model, CEP can be seen as a natural evolution of publish/subscribe middleware.

8.1 Summary

In this thesis, we have discussed the challenges involved in building information-driven applications from the ground up, starting with the sensor/actuator network of a smart home, and ending with the real-time processing of the events generated by hundreds of thousands of connected devices.

The heterogeneity of hardware and protocols in smart home systems is one of the main obstacles preventing the widespread adoption of these technologies. With the SENSEWRAP middleware, presented in Chapter 3, we addressed the

challenge of HETEROGENEITY, through the *virtualization* of physical resources.

Furthermore, in order for a middleware to effectively support both sensors and actuators, it needs to facilitate both *pull* and *push* interactions (INTERACTION STYLES). This challenge was also addressed by SENSEWRAP.

The final challenge in the sensor/actuator domain, SERVICE DISCOVERY, was to find a scalable and convenient way of handling service discovery in the sensor network of a typical smart home. With SENSEWRAP, we demonstrated how the ZeroConf suite of protocols provide elegant mechanisms for service discovery in sensor networks.

Moving on to the event processing domain; being able to process the aggregated events generated by the devices situated in the kind of pervasive environments that smart home systems represent in near real-time, opens up an array of possibilities for new functionality. However, such applications relies on a middleware capable of handling large volumes of potentially heterogeneous events in near real-time. The challenge of developing a general event processing platform that is able to handle these requirements was identified as EVENT PROCESSING ARCHITECTURE.

With the implementation of the EVENTCASTER platform, presented in Chapter 4, we have addressed this challenge: By combining message-oriented middleware and an event processing engine with our own extensions, the EVENTCASTER middleware can effectively handle and analyze the output of hundreds of thousands of connected devices in near real-time. The usefulness and performance of the EVENTCASTER platform is proven by a real-world deployment in the Altibox network.

In addition to this, a paradigm comparison between an imperative and declarative approach to event processing highlighted the advantages and drawbacks of the two paradigms in terms of complexity and performance, addressing the TRADEOFFS challenge. Our findings indicate that the specialized, declarative approach has the edge when it comes to simplicity, and, consequently maintainability, while the imperative approach is the most performant, but requires more effort with regards to optimization.

We have also highlighted some of the shortcomings of the current media

measurement regime, and demonstrated how these can be addressed through the use of event processing techniques: The viewer statistics application, presented in Chapter 6 is a significant improvement of the current model for viewership measurements. This, and the ADSCORER application bridges the gap between online and offline media measurement, giving content providers and advertisers an unprecedented level of detail of television viewership by capitalizing on the two-way communication capabilities of IP-enabled STBs.

While being Research Contributions in their own right, demonstrating novel areas of application for CEP, the viewer statistics application and ADSCORER system also serve as proof of the capabilities of the EVENTCASTER platform.

8.2 Future Work

Our experiences from the real-world deployment of the viewer statistics applications indicates that the relational database, where state and historical statistics are being persisted is the bottleneck of the current deployment. It is likely that a relational database is the wrong tool for the job in this case, and that the persistence and logging functions of such a high-throughput event processing system as this would be better served by a time-series database [112], optimized for rapid writes and large tables. This would also allow for logging at a higher granularity than the 1 minute resolution used for historical data in the current implementation.

However, the issue of persisting high volumes of data is a research area in its own right, and was defined as out of scope for this thesis, but should nonetheless be addressed at a future point in time.

It could also be worthwhile to evaluate MOM based on other specifications than JMS for the dissemination of events, such as DDS, which addresses some of JMS' shortcomings, such as the lack of type safety and QoS, while providing a fully distributed messaging service.

The possibilities for further statistical analysis of the ZAP events collected by the ZAPCOLLECTOR are many. We have probably just scratched the surface of viewer behaviour analysis enabled by modern-day STBs. The correlation

between RMS levels in advertisements and viewer behaviour, as discussed in Section 7.3, is one example of an area that could be subject to further analysis. The regional differences in viewer shares for individual advertisements, as discussed in Section 7.6.2, is another case that could be subject to further research.

Bibliography

[1] K. Aberer, M. Hauswirth, and A. Saheli. The global sensor networks middleware for efficient and flexible deployment and interconnection of sensor networks. Technical report, School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland, 2006.

[2] Apache activemq. Website, 2010. `http://activemq.apache.org/`.

[3] Agama web site. Web, 2011. `http://www.agama.se`.

[4] BB Agarwal, SP Tayal, and M. Gupta. *Software Engineering & Testing: an Introduction*. Jones & Bartlett Learning, 2010.

[5] R.E. Al Qutaish and A. Abran. An Analysis of the Design and Definitions of Halstead's Metrics. In *15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag*, pages 337–352, 2005.

[6] Altibox web site. Web, 2011. `http://www.altibox.no`.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[8] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and

tracking. *Computer Networks*, 46(5):605–634, 2004. Military Communications Systems and Technologies.

[9] J. Kalliosalo I. Karvinen B. Silverajan. Using ietf service discovery methods in ipv6 and middleware platforms and implementing slpv2 for ipv6. In *EUNICE 2003*. 2003.

[10] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to siena. *Comput. J.*, 50:444–459, July 2007.

[11] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *In Proceedings of 25th ICDCS*, pages 437–446, 2005.

[12] Jakob E. Bardram and Martin Mogensen. Dolclan - middleware support for peer-to-peer distributed shared objects. In Jadwiga Indulska and Kerry Raymond, editors, *Distributed Applications and Interoperable Systems, 7th IFIP WG 6.1 International Conference, DAIS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4531 of *Lecture Notes in Computer Science*. Springer, 2007.

[13] P. Baronti, P. Pillai, V.W.C. Chook, S. Chessa, A. Gotta, and Y.F. Hu. Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and zigbee standards. *Computer communications*, 30(7):1655–1695, 2007.

[14] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39:86–98, February 1996.

[15] Blerta Bishaj. Comparison of service discovery protocols. *Helsinki University of Technology*, 2007.

[16] Lars Brenna. *System Support for a Push-based Web*. PhD thesis, University of Tromsø, 2010.

[17] François Bry, Michael Eckert, and Paula lavinia Pătrânjan. Querying composite events for reactivity on the web. In *In Proc. Int. Workshop on XML Research and Applications*, pages 38–47. Springer, 2006.

[18] Bskyb preparing for linear targeting, scheduled for spring 2013. Website, 2011. `http://www.v-net.tv/bskyb-preparing-for-linear-targeting-in-spring-2013/` (accessed 30.05.2012).

[19] CasterStats. Web, 2011. `http://www.casterstats.com/` (accessed 29.11.2011).

[20] Meeyoung Cha, Pablo Rodriguez, Jon Crowcroft, Sue Moon, and Xavier Amatriain. Watching Television over an IP Network. In *IMC*, 2008.

[21] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM.

[22] K. Chandy and W. Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[23] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of ipv4 link-local addresses. Web, May 2005. `http://tools.ietf.org/html/rfc3927` (accessed 02.10.2012).

[24] S. Chesire. How does zeroconf compare with viiv/dlna/d-hwg/upnp? `http://www.zeroconf.org/ZeroconfAndUPnP.html` (accessed 31.08.2012).

[25] S. Chesire and D.H. Steinberg. *Zero Configuration Networking - The Definitive Guide*. O'Reilly, 2006.

[26] Coalition for innovative media measurement. Web, November 2011. http://www.cimm-us.org/about.htm.

[27] Cimm lexicon 1.0. Web, May 2010. http://www.cimm-us.org/CIMM_STB_Lexicon_1_May_2010.pdf (accessed 08.11.2011).

[28] Esper contributors & EsperTech Inc. Esper - complex event processing. http://esper.codehaus.org/index.html, 2007.

[29] David M. Cooperstein, Kim Le Quoc, and Jean-Yves Lugo. The future of media measurement. Web, January 2010. http://www.forrester.com/The+Future+Of+Media+Measurement/fulltext/-/E-RES54091?objectid=RES54091 (accessed 19.03.2013).

[30] A. Corsaro, L. Querzoni, S. Scipioni, Tucci S. Piergiovanni, and A. Virgillito. *Quality of Service in Publish/Subscribe Middleware*, volume 8. IOS Press, July 2006.

[31] Angelo Corsaro. The data distribution service for real-time systems: Part 1. Web, 2010. http://www.drdobbs.com/architecture-and-design/the-data-distribution-service-for-real-t/222900238#/ (accessed 13.08.2012).

[32] L. Coyle, S.Neely, G. Stevenson, M. Sullivan, S. Dobson, and P. Nixon. Sensor fusion-based middleware for smart homes. pages 53–60. Int'l Journal of Assistive Robotics and Mechatronics (IJARM), 2007.

[33] Dave Crane, Eric Pascarello, and Darren James. *Ajax in Action*. Manning Publications, October 2005.

[34] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, jun 2012.

[35] Edward Curry. *Middleware for Communications*, chapter 1, pages 1–28. Wiley, 2004.

[36] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Trans. Softw. Eng.*, 5:96–104, March 1979.

[37] V.K. Da Yu Li and O. Ormandjieva. Halstead's Software Science in Today's Object Oriented World. *Metrics News*, pages 33–41, 2004.

[38] Tv ad spending shoots up in 2011. Website, 2012. `http://www.thedailystar.net/newDesign/news-details.php?nid=222421` (accessed 30.05.2012).

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, January 2008.

[40] Ron Dearing. J2me clients with jini services. Web, 2003. `http://java.sys-con.com/node/37557` (accessed 19.03.2013).

[41] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. *Lecture Notes in Computer Science*, 3896:627–644, 2006.

[42] Alan Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M White, et al. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*, pages 412–422, 2007.

[43] P. Dobrev, D. Famolari, C. Kurzke, and B.A. Miller. Device and service discovery in home networks with osgi. In *IEEE Communications Magazine • August 2002*, pages 86–92. IEEE, 2002.

[44] S. Dobson, P. Nixon, L. Coyle, S. Neely, G. Stevenson, and G. Williamson. Construct: An open source pervasive systems platform. In *4th IEEE Consumer Communications and Networking Conference (CCNC 2007)*, pages 1203–1204. IEEE, 2007.

[45] S. Dorai-Raj, Y. Interian, I. Naverniouk, and D. Zigmond. Adapting online advertising techniques to television. *Online Multimedia Advertising: Techniques and Technologies*, page 148, 2010.

[46] Esper, Performance-Related Information. Web, 2011. `http://esper.codehaus.org/esper/performance/performance.html`.

[47] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29, January 2007.

[48] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[49] Pål Evensen and Hein Meling. Sensewrap: A service oriented middleware with sensor virtualization and self-configuration. In *ISSNIP 2009: Fifth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 261–266, Piscataway, N.J., December 2009. IEEE.

[50] Pål Evensen and Hein Meling. Sensor virtualization with self-configuration and flexible interactions. In *Casemans '09: Proceedings of the 3rd ACM International Workshop on Context-Awareness for Self-Managing Systems*, pages 31–38, New York, NY, USA, 2009. ACM.

[51] Pål Evensen and Hein Meling. A paradigm comparison for collecting tv channel statistics from high-volume channel zap events. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based Systems*, DEBS '11, pages 317–326, New York, NY, USA, 2011. ACM.

[52] Pål Evensen and Hein Meling. Adscorer: an event-based system for near real-time impact analysis of television advertisements (industry article). In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 85–94, New York, NY, USA, 2012. ACM.

[53] R. Fielding, H. Frystyk, T. Berners-Lee, J. Gettys, and Jeffrey C. Mogul. Hypertext transfer protocol - http/1.1. 1996.

[54] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[55] UPnP Forum. Upnp device architecture 1.0, 2008.

[56] Tony Fountain, Sameer Tilak, Peter Shin, Sally Holbrook, Russell J. Schmitt, Andrew Brooks, Libe Washburn, and David Salazar. Digital moorea cyberinfrastructure for coral reef monitoring. In *ISSNIP 2009: Fifth International Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pages 243–248, Piscataway, N.J., December 2009. IEEE.

[57] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[58] Tak-chung Fu, Fu-lai Chung, Robert Luk, and Chak-man Ng. Stock time series pattern matching: Template-based vs. rule-based approaches. *Eng. Appl. Artif. Intell.*, 20(3):347–364, apr 2007.

[59] Hva er TNS Gallup TV-panel? (What is TNS Gallup TV-panel?). Web, 2011. `http://www.tns-gallup.no/?aid=9072596`.

[60] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, New York, NY, USA, 2008. ACM.

[61] Daniel Giusto, Antonio Iera, Giacomo Morabito, Luigi Atzori, Markus Eisenhauer, Peter Rosengren, and Pablo Antolin. Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *The Internet of Things*, pages 367–373. Springer New York, 2010. `10.1007/978-1-4419-1674-7_36`.

[62] Marissa Gluck and Meritxell Roca Sales. The future of television? advertising, technology and the pursuit of audiences. Web, The Norman Lear Center, University of Southern California, September 2008. http://www.learcenter.org/pdf/FutureofTV.pdf.

[63] O. Gnawali, B. Greenstein, K. Jang, et al. The tenet architecture for tiered sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems(SenSys'06)*. ACM, November 2006.

[64] David Goetz. Mpg signs rentrak deal for set-top-box data to help with upfront planning. Web, April 2011.

[65] GreenBus Datasheet. Web, 2011. http://www.greenenergycorp.com/File/View/5052bd52-fb51-4add-a095-c9e300ffed24 (PDF, accessed 06.06.2012).

[66] William Grosso and Robert Eckstein. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2001.

[67] Gürgen, Roncancio, Labbé, Bottaro, and Olive. Sstreamware: a service oriented middleware for heterogeneous sensor data management. In *5th int'l conf. on Pervasive services*, Sorrento, Italy, 2008.

[68] J. Haartsen. Bluetooth-the universal radio interface for ad hoc, wireless connectivity. *Ericsson review*, 3(1):110–117, 1998.

[69] Kim Haase. *Java™ Message Service API Tutorial*. Sun Microsystems, Inc., 2002. http://cs.unc.edu/Courses/jbs/documentation/j2ee_messaging/jms_tutorial.pdf (accessed 22.11.2010).

[70] Paul Haase. Intelligrid: A smart network of power. *EPRI Journal*, (Fall):26–32, 2005.

[71] Peter G. Hamer and Gillian D. Frewin. M.H. Halstead's Software Science - a critical examination. In *ICSE*, 1982.

[72] Dongsu Han, Ashok Anand, Fahad Dogar, Boyan Li, Hyeontaek Lim, Michel Machado, Arvind Mukundan, Wenfei Wu, Aditya Akella,

David G. Andersen, John W. Byers, Srinivasan Seshan, and Peter Steenkiste. Xia: Efficient support for evolvavle internetworking. In *The 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, Berkeley, CA, USA, 2012. USENIX Association.

[73] Bjarne E. Helvik. *Dependable Computing Systems and Communication Networks - Design and Evaluation*. Tapir academic publisher, January 2009.

[74] Kathleen Hickey. Agencies finding more uses for sensor technology. `http://gcn.com/articles/2010/06/24/sensor-technology-use-expected-to-spread.aspx`, 2010.

[75] Hornetq user manual. Website, 2010. `http://hornetq.sourceforge.net/docs/hornetq-2.1.2.Final/user-manual/en/html_single/index.html`.

[76] Jill Huntington-Lee, Kornel Terplan, and Jeff Gibson. *HP Openview: A Manager's Guide*. McGraw-Hill, Inc., New York, NY, USA, 1997.

[77] IETF. Internet engineering taskforce. http://www.ietf.org/.

[78] Rajive Joshi. A comparison and mapping of data distribution service (dds) and java message service (jms). White paper, 2006. `http://portals.omg.org/dds/sites/default/files/Comparison_of_DDS_and_JMS.pdf`.

[79] Kamstrup. Website, 2012. `http://kamstrup.com` (accessed 01.07.2012).

[80] D. Kempe and K.C. Wilbur. What can television networks learn from search engines? how to select, price, and order ads to maximize advertiser welfare. Technical report, Working paper, Viterbi School of Engineering, University of Southern California. http://ssrn.com/abstract1/41423702, 2009.

[81] Landis+gyr. Website, 2012. http://www.landisgyr.com (accessed 01.07.2012).

[82] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[83] Latens web site. Web, 2011. http://www.latens.tv.

[84] Bruce Lawson and Remy Sharp. *Introducing HTML5*. New Riders Publishing, Thousand Oaks, CA, USA, 1st edition, 2010.

[85] Shuoqi Li, Ying Lin, Sang H. Son, John A. Stankovic, and Yuan Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems*, 26:351–368, 2004. 10.1023/B:TELS.0000029046.79337.8f.

[86] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful Bulk Processing for Incremental Analytics. In *SoCC*, 2010.

[87] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. Web, April 2011. http://tools.ietf.org/html/rfc6202 (accessed 21.09.2012).

[88] Peter Lubbers and Frank Greco. The go programming language. Web, 2012. http://www.websocket.org/quantum.html (accessed 21.09.2012).

[89] David Luckham. What's the difference between esp and cep? http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/, 2006.

[90] Karl Philip Lund. Gamle målemetoder! Web, October 2010. http://www.kampanje.com/kommentert/article5772345.ece (accessed 25.11.2011).

[91] Mariner Partners - IPTV Monitoring Software. Web, 2011. `http://www.marinerpartners.com`.

[92] Apache maven. Website, 2012. `http://maven.apache.org/` (accessed 22.05.2012).

[93] Scott McLean, Kim Williams, and James Naftel. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002.

[94] Paul McNamara. Ban on loud tv commercials takes effect today. Web, 2012. `http://www.networkworld.com/community/blog/ban-loud-tv-commercials-takes-effect-today` (accessed 21.01.2013).

[95] Magna: Tv ad spending on the upswing for foreseeable future. Website, 2010. `http://www.mediapost.com/publications/article/129755/` (accessed 30.05.2012).

[96] Hein Meling. *Adaptive middleware support and autonomous fault treatment : architectural design, prototyping and experimental evaluation*. PhD thesis, Norwegian University of Science and Technology, 2006.

[97] Daniel A. Menasce. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing*, 9(2):90–93, mar 2005.

[98] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced Event Processing and Notifications in Service Runtime Environments. In *DEBS*, 2008.

[99] Sun Microsystems. Sun small programmable object technology. `http://www.sunspotworld.com/`, 2012.

[100] Misc. Jboss community. Web, 2013. `http://www.jboss.org/(accessed11.01.2013)`, url = http://www.jboss.org/,.

[101] Miscalleneous. Streaming text-oriented protocol. Web, 2012. `http://stomp.codehaus.org` (accessed 01.11.2012).

[102] Miscellaneous. Faq: Technical - mono. Web. http://www.mono-project.com/FAQ:_Technical (accessed 19.03.2013).

[103] Miscellaneous. Cacti. Web, 2012. http://www.cacti.net/.

[104] Miscellaneous. Comet (programming). Web, 2012. http://en.wikipedia.org/wiki/Comet_(programming) (accessed 21.09.2012).

[105] Miscellaneous. Det norske markedet for elektroniske kommunikasjonstjenester. Web, 2012. http://www.npt.no/marked/ekomtjenester/statistikk/det-norske-ekommarkedet-rapporter/_attachment/3910?_ts=13a78405f3b (accessed 08.01.2013).

[106] Miscellaneous. The go programming language. Web, 2012. http://golang.org/.

[107] Miscellaneous. The norwegian post and telecommunications authority. Web, 2012. http://eng.npt.no/portal/page/portal/PG_NPT_NO_EN/PAG_NPT_EN_HOME (accessed 08.01.2013).

[108] Miscellaneous. Rentrak expands tv contract with local tv, llc. Web, 2012. http://www.prnewswire.com/news-releases/rentrak-expands-tv-contract-with-local-tv-llc-151096845.html (accessed 24.11.2012).

[109] Miscellaneous. Sinclair drops nielsen for rentrak in 4 cities. Web, 2012. http://www.tvnewscheck.com/article/60414/sinclair-drops-nielsen-for-rentrak-in-4-cities?utm_source=Listrak&utm_medium=Email&utm_term=Sinclair+Drops+Nielsen+For+Rentrak+In+4+Cities&utm_campaign=Sinclair+Drops+Nielsen+For+Rentrak+In+4+Cities (accessed 24.11.2012).

[110] Miscellaneous. Splunk. Web, 2012. http://www.splunk.com/.

[111] Miscellaneous. Squidbee. Web, 2012. `http://www.libelium.com/squidbee/`.

[112] Miscellaneous. What's opentsdb. Web, 2012. `http://opentsdb.net/` (accessed 11.10.2012).

[113] Brian C. J. Moore, Glasberg Brian R., and Michael A. Stone. Why are commercials so loud? perception and modeling of the loudness of amplitude-compressed speech. *Journal of the Audio Engineering Society*, 51(12):1123–1132, 2003.

[114] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[115] Ted Neward. The busy java developer's guide to scala: Dive deeper into scala concurrency. `http://www.ibm.com/developerworks/java/library/j-scala04109.html`, April 2009.

[116] Nielsen Ratings. Web, 2011. `http://en.wikipedia.org/wiki/Nielsen_ratings`.

[117] E. Nordstrom, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M.J. Freedman. Serval: An end-host stack for service-centric networking. *Proc. 9th USENIX NSDI*, 2012.

[118] S. Oaks, B. Traversat, and L. Gong. *JXTA in a Nutshell*. O'Reilly Media, Incorporated, 2002.

[119] Opher Etzion and Peter Niblett. *Event Processing In Action*. Manning, August 2010.

[120] M. Parashar and S. Hariri. *Autonomic computing: concepts, infrastructure, and applications*. CRC Press/Taylor & Francis, 2007.

[121] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, 2004.

[122] M. Pipattanasomporn. Multi-agent systems in a distributed smart grid: Design and implementation. In *Power Systems Conference and Exposition, 2009. PSCE '09. IEEE/PES*, pages 1–8, 2009.

[123] Edmund Prater, Gregory V. Frazier, and Pedro M. Reyes. Future impacts of rfid on e-supply chains in grocery retailing. *Supply Chain Management: An International Journal*, 10(2):134–142, 2005.

[124] IBM Research. The gryphon project. `http://www.research.ibm.com/distributedmessaging/gryphon.html` (accessed 28.11.2010).

[125] D. Retkowitz and S. Kulle. Dependency management in smart homes. In Twittie Senivongse and Rui Oliveira, editors, *Distributed Applications and Interoperable Systems, 9th IFIP WG 6.1 International Conference (DAIS 2009)*, volume 5523 of *LNCS*, pages 143–156. Springer Verlag, 2009.

[126] Mark Richards. Understanding the differences between amqp & jms. Web, 2011. `http://www.wmrichards.com/amqp.pdf` (accessed 01.11.2012).

[127] Java rmi over iiop. http://java.sun.com/products/rmi-iiop/. Last visited December 2010.

[128] C. Rong, H. Meling, and D. Wåge. Towards integrated services for health monitoring. In *First Int'l Workshop on Smart Homes for Tele-Health*, Niagara Falls, Canada, May 2007.

[129] P. Saint-Andre, K. Smith, and R. Tronçon. *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, Incorporated, 2009.

[130] M Scheffler and E Hirt. Wearable devices for telemedicine applications. *Journal of Telemedicine and Telecare*, 11(1):11–14, 2005.

[131] Jean Schmitt. NetComplete Home Performance Management (PM). White paper, November 2009. `http://www.jdsu.com/ProductLiterature/netcompletehomepm_WP_sas_TM_AE.pdf`.

[132] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[133] V.Y. Shen, S.D. Conte, and H.E. Dunsmore. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Trans. Softw. Eng.*, 9:155–165, 1983.

[134] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical report, Harvard University, 2004.

[135] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web services with SOAP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[136] Statistisk sentralbyrå (statistics norway). Website, 2012. `http://www.ssb.no/familie/` (accessed 26.05.2012).

[137] Brian Stansberry, Galder Zamarreno, and Paul Ferraro. High availability enterprise services with jboss application server clusters. Web, 2012. `www.jboss.org/jbossclustering/docs/cluster_guide/5.1/pdf/Clustering_Guide.pdf` (accessed 08.10.2012).

[138] Jon Stearley, Sophia Corwell, and Ken Lord. Bridging the gaps: joining information sources with splunk. In *Proceedings of the 2010 workshop on Managing systems via log analysis and machine learning techniques*, SLAML'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[139] Inc. SYS-CON Media. J2me clients with jini services. `http://www2.sys-con.com/itsg/virtualcd/Java/archives/0806/patil/index.html`, 2004.

[140] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[141] R. Thurlow. Rpc: Remote procedure call protocol specification version 2. Web, May 2009. `http://tools.ietf.org/html/rfc5531` (accessed 18.11.2012).

[142] TNS Global Market Research. Web, 2011. `http://www.tnsglobal.com/`.

[143] TRA Global. Web, 2011. `http://www.traglobal.com/` (accessed 27.11.2011).

[144] Robert Tripp. Report 1.2: Core banking processes and recent strategies. Web, April 2003. `http://www.howbankswork.com/1-2.html#7` (accessed 02.10.2012).

[145] S. Vinoski. Rpc under fire. *Internet Computing, IEEE*, 9(5):93–95, 2005.

[146] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14:46–55, 1997.

[147] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2006. ACM.

[148] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1997.

Appendix A

Comparing Two M-Shape Implementations

This appendix contributes to the paradigm comparison presented in Chapter 6. Performing an additional paradigm comparison between Java and another specialized event processing language than EPL, provides us with further observations regarding the tradeoffs between different programming language paradigms for performing event processing.

In particular, we are interested to see how the differences in complexity between implementations using Java and a specialized event processing processing languages compares to our previous findings when applied to another application and another event processing language.

For this reason, we compare the complexity of two functional identical implementations of an algorithm for detecting an "M-shape" pattern (also referred to as "double top" in the financial industry [58]) in a stream of values. One is implemented in Java, while the other is implemented using the specialized event processing language Cayuga Event Language (CEL). CEL is part of the Cayuga event monitoring system [42], and like EPL, it is a declarative programming language with an SQL-like syntax. The CEL implementation is borrowed from Lars Brenna's PhD thesis [16].

Listing A.1 lists the CEL implementation. Listing A.2 lists the Java implementation. We refer to Brenna's thesis and other Cayuga publications for

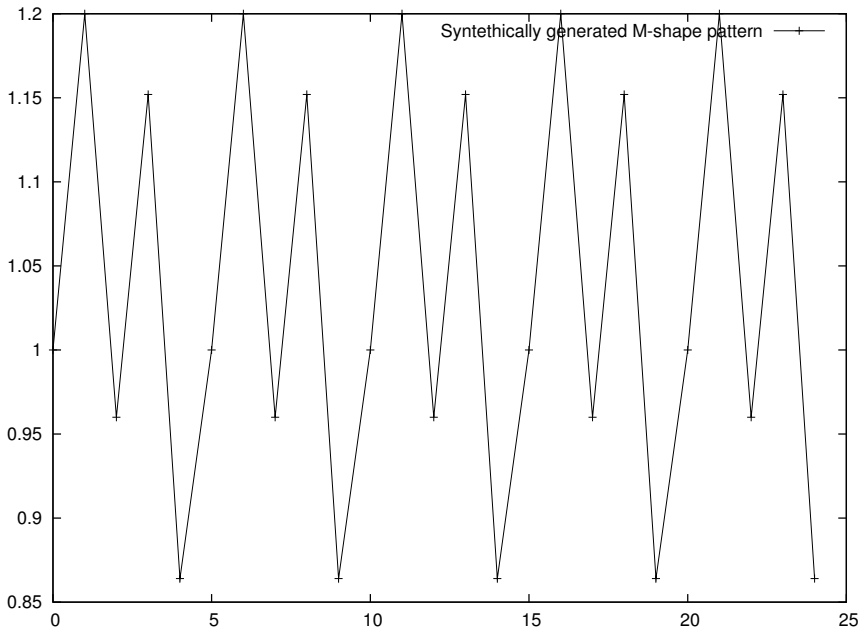

Figure A.1: Repeating M-shape pattern

details concerning CEL. Tests were performed on the Java implementation to verify that it exhibited the desired behaviour. Clients were connected through HornetQ.

To test the performance of the Java implementation of the algorithm, a file consisting of 100 000 repeating M-shapes was generated. The file also included some ”noise“ in the start, to verify that the pattern detection was working correctly. Figure A.1 shows the start of the file, and includes four valid M-shapes, where the middle of the ”M“ is not lower than the beginning or end of the pattern.

Ideally, we would also have compared the performance of the two implementations. However, a turn of events led us to focus our efforts on developing the ADSCORER application instead.

Using Halstead's metrics in the same way as described in Section 6.5.3, we were able to calculate complexity scores for each implementation, as shown in Table A.1. As illustrated in Figure A.2, the Java implementation has higher complexity scores in all categories. This is hardly surprising when considering

that CEL is a specialized tool for these kinds of tasks, and adds further weight to our conclusions in Chapter 6, where Java was compared to another dedicated event processing language. Similar to the results of this exercise, the complexity metrics of the implementation presented in Chapter 6 written in the EPL language were lower than the metrics of its Java equivalent.

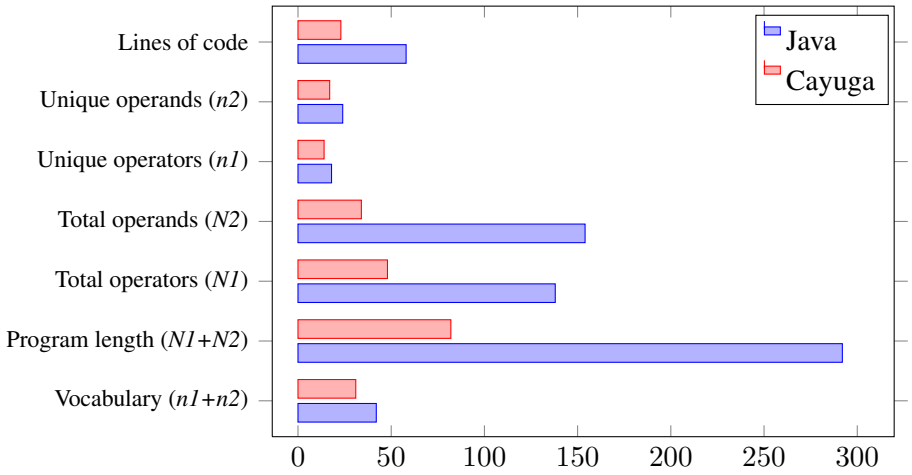

Figure A.2: Complexity metrics breakdown for CEL and Java

Metric	**Java**	**CEL**
Source lines of code	58	23
Program length (*N1*+*N2*)	292	82
Unique operators (*n1*)	18	14
Unique operands (*n2*)	24	17
Total operators (*N1*)	138	48
Total operands (*N2*)	154	34
Vocabulary (*n1*+*n2*)	42	31

Table A.1: The underlying complexity metrics for Figure A.2

Listing A.1 Detection of M-shape stock pattern, written in CEL

```
1  SELECT Name, pA, pB, pC, pD, Price AS pE, pF
2  FROM FILTER {pF  Price AND pF  pA} (
3  FILTER {Price  1.1pB} (
4  SELECT Name, pA, pB, pC, p 1 AS pD, Price
5  FROM
6  FILTER {Price  0.9pB} (
7  SELECT Name, pA, pB, p 1 AS pC, Price
8  FROM
9  FILTER {Price  0.9pA AND Price = 1.1pA} (
10 SELECT Name, pA, p 1 AS pB, Price
11 FROM
12 FILTER {Price  1.2pA} (
13 SELECT Name, p 1 AS pA, Price
14 FROM
15 (FILTER {Price  p 1}(
16 (SELECT Name, Price FROM Stock)
17 NEXT {$1.Name = $2.Name} Stock)
18 FOLD {$1.Name = $2.Name, $2.Price  $.Price,} Stock))
19 FOLD {$1.Name = $2.Name, $2.Price  $.Price,} Stock)
20 FOLD {$1.Name = $2.Name, $2.Price  $.Price,} Stock)
21 FOLD {$1.Name = $2.Name, $2.Price  $.Price,} Stock)
22 NEXT {$1.Name = $2.Name2}
23 (SELECT Name AS Name2, Price AS pF FROM Stock))
```

Listing A.2 Detection of M-shape stock pattern, written in Java (partial)

```
1  public void eval(Double newValue) {
2    switch (state) {
3      case Reset:
4        Double rVal = 0.0;
5        if(baseValues.get(Reset) != 0)
6          rVal = baseValues.get(Reset);
7        baseValues.clear();
8        if(rVal != 0.0)
9          baseValues.put(Reset, rVal);
10       else
11         baseValues.put(Reset, prevValue);
12       if (newValue >= (baseValues.get(Reset) * 1.2)) {
13         state = Up;
14         baseValues.put(Up, newValue);
15       }
16       else if (newValue < prevValue)
17         baseValues.put(Reset, newValue);
18       break;
19     case Up:
20       if (newValue < baseValues.get(Reset))
21         state = Reset;
22       else if (newValue <= (baseValues.get(Up) * 0.9)) {
23         state = UpDown;
24         baseValues.put(UpDown, newValue);
25       }
26       else if (newValue > baseValues.get(Up))
27         baseValues.put(Up, newValue);
28       break;
29     case UpDown:
30       if (newValue < baseValues.get(Reset))
31         state = Reset;
32       else if (newValue >= (baseValues.get(UpDown) * 1.1)) {
33         state = UpDownUp;
34         baseValues.put(UpDownUp, newValue);
35       }
36       else if (newValue < baseValues.get(UpDown))
37         baseValues.put(UpDown, newValue);
38       break;
39     ...
40     // Listing is only partial to fit code on one page
41   }
42   prevValue = newValue;
43 }
```