MANAGEMENT AND FEDERATION OF STREAM PROCESSING APPLICATIONS

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

MICHAEL ALEXANDER DULLER

Master of Science ETH in Computer Science, ETH Zurich

born July 29, 1981

citizen of Austria

accepted on the recommendation of

Prof. Gustavo Alonso
Prof. Friedemann Mattern
Prof. M. Tamer Özsu
Prof. Nesime Tatbul

2011
In loving memory of my grandmother.
Abstract

A decade ago, stream processing has enabled a new class of applications by employing a fundamentally different processing model than conventional data base management systems. These applications process large volumes of continuous streams of input data with high throughput. Examples include network intrusion detection, financial transaction processing, and variable road tolling. Data stream management systems have evolved into industrial strength solutions for this class of applications and use long-running queries to process high volumes of continuous input data with low latency. However, they still lack flexibility in terms of large-scale deployment, integration, extensibility, and interoperability.

In the last years, a substantial ecosystem of new applications has emerged that can potentially benefit from stream processing. They range from the federation of existing but heterogeneous streaming applications to automated deployments of streaming applications in large clusters or cloud environments to processing personal information like photos, e-mail, or text (SMS) messages as data streams. These applications introduce different requirements on how stream processing solutions can be deployed, integrated, extended, and federated.

This thesis explores stream processing with the help of traditional stream processing applications as well as applications that process personal information as data streams. Despite or even because of the very different application use cases, the thesis identifies
the fundamental properties that are common to all stream processing systems. The result is a generic model for stream processing and an architecture for a dynamic platform that supports the model. The model separates processing (operators) and data management (buffers) into distinct entities. Stream processing systems that have been implemented conforming to the model, including existing systems that have been ported, benefit from deployment and runtime support by the platform. They also support basic interfaces for data exchange. Intermediate and final results are made available in the model’s explicit entities for data management.

These properties enable the automated deployment of applications, facilitate the federation of applications running on heterogeneous stream processing systems, and leverage stream processing in new application domains. This thesis validates the generality of the model, the feasibility in terms of overhead, and the claims made in terms of deployment and integration. Experiments on PlanetLab, on a cluster, and on individual nodes confirm that the model and platform proposed in the thesis enable the interoperability between heterogeneous stream processing engines, facilitate the distributed deployment, and add functionality to the engines (ability to replace operators at runtime or to run a distribution-agnostic engine in a distributed setup) with negligible overhead.
Zusammenfassung


Contents

Abstract v

Zusammenfassung vii

1 Introduction 1
   1.1 Beyond Core Stream Processing . . . . . . . . . . . . 2
   1.1.1 Virtualizing Stream Processing . . . . . . . . . . . 3
   1.1.2 Personal Information as Data Streams . . . . . . . 5
   1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . 8

2 Design 11
   2.1 Architectural Philosophy . . . . . . . . . . . . . . . . 12
   2.2 System Model . . . . . . . . . . . . . . . . . . . . . . 14
   2.3 Use Case: Photo Exchange . . . . . . . . . . . . . . . 16
      2.3.1 End User . . . . . . . . . . . . . . . . . . . . . 16
      2.3.2 Communities of Users . . . . . . . . . . . . . . 18
      2.3.3 Developers . . . . . . . . . . . . . . . . . . . . 19
      2.3.4 Service Providers . . . . . . . . . . . . . . . . 20
   2.4 Data Processing Model . . . . . . . . . . . . . . . . . 20
      2.4.1 Requirements . . . . . . . . . . . . . . . . . . . 20
      2.4.2 Entities . . . . . . . . . . . . . . . . . . . . . . 23
      2.4.3 Processing Mesh . . . . . . . . . . . . . . . . . 26
      2.4.4 Data Items . . . . . . . . . . . . . . . . . . . . 27
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.5</td>
<td>Stream and Store Data Paths</td>
<td>27</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Channels as Views</td>
<td>28</td>
</tr>
<tr>
<td>2.4.7</td>
<td>Merging and Replication</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Implementation Model</td>
<td>31</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Requirements</td>
<td>31</td>
</tr>
<tr>
<td>2.5.2</td>
<td>A Service-Oriented Architecture</td>
<td>33</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Entities</td>
<td>34</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Processing Mesh</td>
<td>37</td>
</tr>
<tr>
<td>2.5.5</td>
<td>Data Access Facets and Mechanisms</td>
<td>38</td>
</tr>
<tr>
<td>2.5.6</td>
<td>Distribution</td>
<td>39</td>
</tr>
<tr>
<td>2.5.7</td>
<td>Basic Interfaces and Extensibility</td>
<td>42</td>
</tr>
<tr>
<td>2.5.8</td>
<td>Component Processing Model</td>
<td>43</td>
</tr>
<tr>
<td>2.5.9</td>
<td>Component Life Cycle Management</td>
<td>47</td>
</tr>
<tr>
<td>2.5.10</td>
<td>Application Setup and Management</td>
<td>48</td>
</tr>
<tr>
<td>2.6</td>
<td>Interface</td>
<td>49</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Requirements</td>
<td>49</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Application Builders</td>
<td>49</td>
</tr>
<tr>
<td>2.7</td>
<td>Federation and Semantics</td>
<td>50</td>
</tr>
<tr>
<td>2.8</td>
<td>Summary</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>Implementation</td>
<td>55</td>
</tr>
<tr>
<td>3.1</td>
<td>XTream Platform</td>
<td>55</td>
</tr>
<tr>
<td>3.1.1</td>
<td>OSGi</td>
<td>56</td>
</tr>
<tr>
<td>3.1.2</td>
<td>R-OSGi</td>
<td>57</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Platform Implementation</td>
<td>58</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Component Implementation</td>
<td>64</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Component Binding and Interaction</td>
<td>68</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Data Management</td>
<td>73</td>
</tr>
<tr>
<td>3.1.7</td>
<td>Component Life Cycle Management</td>
<td>74</td>
</tr>
<tr>
<td>3.1.8</td>
<td>Application Setup and Management</td>
<td>75</td>
</tr>
<tr>
<td>3.1.9</td>
<td>Distributed Operation</td>
<td>77</td>
</tr>
<tr>
<td>3.2</td>
<td>Declarative Buffer Implementation</td>
<td>80</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Windows on the Stream Data Path</td>
<td>80</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Access Variations and Interface</td>
<td>81</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Window Configuration Language</td>
<td>82</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Data stream processing is a well-established technique to process continuous streams of input data in a data management system. In the last decade, the discipline evolved from a disruptive research idea into an important business area of major database vendors. Today, data stream processing (or stream processing for short) is the foundation for a number of enterprise applications that use long-running queries to process large amounts of continuous input data with low latency. Examples of these applications include financial transaction processing, network intrusion detection, traffic monitoring, and sensor data aggregation.

Stream processing enables this new class of applications by turning the processing model of traditional databases upside down. Instead of storing data and processing queries, which arrive repeatedly, stream processing systems store queries and process data, which arrives continuously, through the stored queries. Figure 1.1 illustrates the processing model of traditional databases on the left and the stream processing model on the right. The components of a data stream management system (DSMS), such as query compiler, scheduler, and storage manager, are designed and optimized towards this form of processing.
This thesis explores opportunities and challenges of stream processing beyond its traditional application domains and core data management issues (e.g., query optimization or storage management) to further leverage the technique of stream processing and extend its applicability to new domains.

1.1 Beyond Core Stream Processing

The research presented in this thesis intends to grasp stream processing at a more general level, detached from specifics of individual models, semantics, or implementations. It answers the question *What are the fundamental properties common to all stream processing systems?* by studying traditional stream processing applications as well as new areas of stream processing. The answer to this question is a generic model for stream processing and an architecture of a stream processing platform that supports this model and thus a wide range of applications of stream processing.

This model and its supporting platform enable the virtualization of conventional stream processing applications to facilitate their extensibility, interoperability, and deployment [16]. They also facilitate the application of stream processing techniques to application domains beyond conventional stream processing applications. As an
example for such a new application domain that is very different from traditional applications on data streams, this thesis explores the processing of personal information as streams of data [15].

1.1.1 **Virtualizing Stream Processing**

Stream processing has proven to be useful for many of the traditional streaming applications. However, its applicability is still limited for a number of real-world use cases (targeting in particular stream processing in clusters and cloud-based systems) due to challenges that arise from interoperability, extensibility, integration, and deployment.

Interoperability refers to the integration of heterogeneous stream processing engines (SPEs) [46]. A common scenario involves different engines run by different and autonomous entities that must work together but cannot resort to a homogeneous solution. We have encountered such a scenario in automatic financial compliance checking, where government authorities validate streams of transactions created by financial institutions. A similar scenario arises in the supply chain management, where different RFID and bar code technologies, pallet and container tracking systems, and bookkeeping and stock control software need to be coordinated even across large geographic distances. From these scenarios we derive the requirement to be able to host in a single platform different systems that communicate through well-defined interfaces, while maintaining the authoritative boundaries imposed by each organization.

Extensibility refers to the desirable property of being able to easily extend an existing system. In industry, this includes adding new features or making required adaptations to make the system suitable for new or changed application requirements. In research, extensibility facilitates trying out new concepts, algorithms, and implementations.

In terms of integration, stream processing systems lack a common interface of how to interact with the system and how to package and integrate it with other software. Related issues for relational
database systems have been dealt with in standards like JDBC and ODBC as well as library databases like H2 or SQLite. JDBC and ODBC standardize how to connect to and interact with a relational database. In combination with packaging database engines as library databases, relational database technology is used not only in large enterprise setups but also in smaller applications like desktop applications. Stream processing technologies could also be beneficial to many smaller applications that currently resort to custom implementations. Once the integration of stream processing systems is similarly simple as it is for relational databases, stream processing technologies will also be leveraged in many smaller applications.

In terms of deployment, there is an increasing need to deploy SPEs flexibly to run them as virtual entities across a cluster and even across a cloud computing facility. The scenarios and motivation for this requirement are identical to those for standard applications and relational databases: elasticity, cost reduction, and fast provisioning. What is needed here is an automatic way to change the deployment of (parts of) a streaming application to be able to place it in a single server or across multiple servers depending on need. Being able to dynamically replace components in a running system is important in such scenarios for both maintenance purposes, but also to facilitate fault-tolerant solutions.

To address these challenges we explore the possibility of virtualizing any component of a stream processing system (operators and buffers as well as entire engines) so that they can be automatically deployed, managed, and composed in a flexible and dynamic manner. Like other virtualization approaches (e.g., machine virtualization like Xen or VMware and managed language runtimes for bytecode like Java VMs or the .NET CLR), the main objective of our approach is to gain flexibility (e.g., location transparency), ease of management (e.g., push-button deployment of virtual machines), and potential for optimizations (e.g., replacement of performance-critical code with an optimized version at runtime).

We propose a middleware platform that hosts virtualized components of stream processing systems. Middleware has already proven
to be useful to provide additional features for data processing systems like, e.g., TP monitors or message queuing systems for traditional databases. In fact, several stream processing engines have been extended with middleware platforms: IBM’s System S [28] or Yahoo’s S4 [33]. These systems are built as extensions to one particular implementation of an engine. Our approach is a pure middleware platform that is independent of a particular engine. We leverage existing engines with their idiosyncrasies and the applications already existing for these engines. In that way we do not impose engine-specific semantics or a processing model but cater to dynamic and distributed operation, deployment, and life cycle management and provide interoperability between heterogeneous SPEs with potentially different semantics in order to further leverage stream processing and promote its ubiquitousness.

1.1.2 Personal Information as Data Streams

We propose to use stream processing to process and disseminate personal information. Personal information includes information that is directly relevant to people like, e.g., photos, e-mail, chat, and text (SMS) messages. One the one hand, the application of stream processing in this area validates the generality of the model and platform presented in this thesis. On the other hand, it demonstrates the usability of stream processing in other domains in tackling the problems and challenges of processing personal information as outlined below. It therefore also confirms that it is worthwhile to leverage stream processing by facilitating its integration and deployment, as done with the platform presented in this thesis.

Pervasive access to the Internet as well as the proliferation of mobile devices for both producing (e.g., cameras, smartphones) and consuming (e.g., smartphones, e-readers) digital media have led to a vast amount of personal information being shared and distributed across the globe. In addition to pictures and video clips, there is also a wide variety of other personal information being exchanged such as chat, text (SMS), and e-mail messages.
Online services and communities like Flickr, YouTube, Twitter, or Facebook provide a partial solution to the challenge of handling personal information. In these platforms, users upload their information to a centralized service which in turn stores it and provides access to the owner as well as to designated groups of users or the public. We identify the following limitations and challenges of these services that need to be addressed to leverage the full potential of personal information:

** Processing:** The primary goal of these services is to store data. They do not support sophisticated data processing, since it is not possible to provide significant amounts of CPU cycles to the large number of users of the service.

** Extensibility:** The functionality provided by these services is typically limited. Some services provide extension mechanisms for custom applications (e.g., Flickr App Garden [21], Facebook API [20]). However, as the primary focus is on storing data, these extension mechanisms do not allow for complex processing.

** Integration:** Integration of these services is limited to the possibilities implemented by the providers. The possibilities for integration of services on the client side are also limited, as the web browser is the primary interface and client to these services; text can be copied through the clipboard and files exchanged by downloading and uploading them, but that is about all.

** Dissemination:** These services do not provide the means for efficient dissemination of the information that has been uploaded. Instead, users can subscribe to notifications and then access the new information by visiting the service's website. The approach is clearly related to the advertisement-driven business model of the services.
1.1. BEYOND CORE STREAM PROCESSING

**Size:** The ever increasing amount of information being uploaded to these services requires the steady expansion of the data centers backing the services. This directly translates into operational costs that must be recovered by the services’ business models. Given the rising energy prices and the increasing awareness of ecological concerns, we can assume that this problem will become more relevant in the future.

**Privacy:** Not everybody feels comfortable with giving away information to a service operated by a big corporation just to be able to easily exchange it with friends. An increasing number of users are becoming aware of and concerned with the loss of control over their personal data, terms of use granting the service provider irrevocable rights to, e.g., private pictures, and the target advertisement they are subjected to.

An ideal solution to dealing with personal information should address the challenges identified above. It should provide not only a storage facility but also processing capabilities that allow to arbitrarily process the data. It should also be easy to integrate with other kinds of data and information.

In addition to storing data, it should also be possible to disseminate data efficiently and proactively to users’ devices. Like push e-mail, this will allow users to access information directly and potentially in a situation without connectivity, because the information has been pushed to the device at an earlier point in time.

Finally, limitations of centralized services in terms of storage space and processing capacity can be mitigated by leveraging the users’ devices to store and process information. There are millions of devices in operation and mostly idling in peoples’ homes like Internet gateways (e.g., wireless routers), network attached storage (NAS) devices, or even dedicated home servers. Taking advantage of such devices will lead to users benefiting from low latency, direct access to their own data, virtually unrestricted storage space and CPU cycles, and greater control over their data. Leveraging users’ devices will
push content and processing back to the edge of the network—where it is and has always been produced and consumed.

However, we cannot assume that everybody has a suitable device or is willing to keep the device running around the clock. Therefore, it is important that the advantages of the approach (flexible processing, efficient data dissemination) can also be leveraged in a hosted environment, and that hosted and private setups can interact seamlessly with each other.

1.2 Contributions

This thesis presents a generic system model for stream processing, an architecture of a stream processing platform that support this model, an implementation of the architecture, application implementations, and an evaluation of the architecture.

Chapter 2 introduces the architectural philosophy behind our approach and contributes a complete, layered system model for processing and disseminating streaming data. The model does not impose any restrictions on the implementation of stream processing elements and carefully separates concerns so that data processing, data management, and communication are treated separately. The requirements for each layer of the system model are derived from analyzing existing stream processing systems, an example use case of processing personal information as streams, and the desired properties in terms of interoperability, integration, and distributed deployment. The conclusions result in detailed models for each layer of the system model. They provide well defined, extensible interfaces for encapsulating stream processing entities (operators, buffers) and building applications on top of these. In addition to the model, this chapter also contributes the consequential architecture of an open, flexible, and extensible platform for distributed stream processing at global scale that supports dynamic composition and component life cycle management.
1.2. CONTRIBUTIONS

Chapter 3 contributes our implementation of a corresponding distributed stream processing platform and thus illustrates the concepts outlined in the models in Chapter 2. XTream, our implementation, is based on OSGi, a module and service framework for Java, and provides a clean API for the development of stream processing components and the composition of the same into applications. This chapter also explains the development of applications using the platform and specifies how existing stream processing engines can be ported to the platform. An implementation of a declarative buffer framework exemplifies the extension of components of the system and completes the implementation chapter.

Chapter 4 contributes the implementation of two applications on personal data streams and illustrates the origins of processing personal information as data streams. They motivate the requirements for a stream processing platform running this kind of applications and illustrate how the model and architecture proposed in Chapter 2 meet these requirements.

Chapter 5 contributes the evaluation of the model presented in Chapter 2 using the implementation described in Chapter 3. This chapter quantifies the low overhead of the approach and validates its applicability for traditional and new application domains of stream processing using a standard stream processing benchmark and a deployment on PlanetLab. It demonstrates the functionality gained: the ability to replace parts of the processing mesh at runtime, the federation of heterogeneous stream processing engines, and the distributed deployment and operation of a centralized stream processing engine that improves benchmark performance by an order of magnitude. Finally, this chapter quantifies the development effort required to use the platform.

Chapter 6 provides the background for stream processing and presents related approaches to runtime platforms for stream pro-
cessing as well as orthogonal concepts that can be combined with the work presented in this thesis. It also discusses related techniques for processing and disseminating data.

Chapter 7 summarizes the contributions of this thesis. It concludes with an overview over ongoing and future work that is related to and was inspired by work and experiments conducted in the context of this thesis.
Chapter 2

Design

As stream processing has matured over the last decade, a variety of stream processing systems became available. While they share the same processing model—which makes them stream processing systems in the first place—they differ in a variety of aspects. Examples include high-level aspects like the query interface, query language, and data model as well as implementation aspects like the processing model, the implementation language, or buffer implementations.

In this chapter we analyze the requirements and properties of existing stream processing systems as well as an example use case of processing personal information as streams. In conjunction with the desired properties of interoperability, extensibility, integration, and distributed deployment, we derive a generic model for stream processing that satisfies these requirements. The model examines stream processing at different layers of abstraction: interface, data processing model, implementation model, and actual instantiation. Requirements and properties are aligned with the layers and are discussed in the respective sections.

The model guides the design of the architecture of a stream processing platform that supports a variety of different stream processing systems. Properties of the architecture and how they satisfy the
requirements are discussed in the sections discussing the respective layers of the system model.

In addition to the layered system model, its components, and the architecture of an open, flexible, and extensible platform for distributed stream processing, this chapter also presents the architectural philosophy behind our approach and discusses implementation and semantic aspects of federating streaming applications.

2.1 Architectural Philosophy

To support a variety of stream processing systems with their specific policies and implementation aspects, we propose an *exoengine* architecture that makes as few assumptions and implements as few policies as possible. In contrast to implementing yet another complete stream processing system that implements all currently known and required functionality, our approach provides more flexibility and is thereby future-proof and widely applicable.

The exoengine architecture we propose is inspired by the concept of the exokernel for operating systems. An exokernel ideally implements no abstractions or policies and instead only securely multiplexes the hardware between multiple domains (processes) [18, 31]. Operating system functionality like virtual memory, paging, file systems, or the network stack is implemented inside the domains and can be reused as *libraries* in multiple domains. Leaving operating system abstractions out of the kernel allows applications to tailor them and choose the most appropriate implementation of each abstraction or even omit some or all of them altogether. Examples include managed language runtime systems that implement their own memory management, databases that access raw disk blocks and provide their own buffering, or firewall systems that inspect raw network packets. It is possible to build, e.g., a full UNIX operating system as a library on top of an exokernel (*library OS*) and run standard UNIX applications on it. Once implemented, a library operating system can be reused to run any application that runs on
the operating system. The application itself does not need to be adapted to the exokernel.

Today, the concept of the exokernel is widely used in the form of type 1 hypervisors for virtualization, e.g., VMWare ESXi [50] or Xen [7]. The hypervisor securely multiplexes the hardware between virtual machines and the VM abstraction enables their flexible management, deployment, and migration. Conventional operating systems run on top of the hypervisor and allow to run any application that runs on the respective operating system.

Similar to exokernels, we postulate that a stream processing platform should implement as few policies (e.g., scheduling) and make as few assumptions about applications and their implementation (e.g., storage) as possible. The platform does not provide a stream processing engine (SPE), individual operators, or buffer implementations. This functionality is implemented as component libraries on top of the platform. Such components can be single operators or full streaming engines. Applications that have been custom tailored to an SPE and its particular query language and optimizations can be run unmodified using the corresponding library SPE implementation on the platform. The library SPE only needs to be provided once by adapting the existing, standalone implementation of the SPE.

Figure 2.1 illustrates the exoengine architecture:

1. A specific stream processing engine (e.g., SPE $X$) is running as library SPE on top of the exoengine platform. Applications defined for this specific stream processing engine (e.g., App $X_1$, App $X_2$) run on top of the library SPE.

2. Multiple, heterogeneous engines running on an exoengine platform can exchange data either through the basic interfaces defined by the platform or through agreed-upon, richer interfaces extending the basic interfaces.

3. Through the virtualization of components, applications or individual components thereof can be moved between instances.
of the platform, thereby giving the designer more freedom in the deployment and configuration.

4 Single applications can be deployed and run across multiple machines, enabling the use of centralized engines in a distributed setting with the platform taking care of communication and data forwarding.

2.2 System Model

The nature of the requirements of stream processing systems and the desired properties is varied. Some are high-level characteristics like, e.g., the interface to the system. Others are low-level implementation aspects like, e.g., whether an engine’s processing model is push or pull driven. Therefore, we investigate stream processing systems and applications at different layers of abstractions. These layers comprise the system model as it is illustrated in Figure 2.2.

On top, a high-level abstraction, e.g., a streaming query language, presents the interface to the system. In the example shown,
2.2. SYSTEM MODEL

SELECT AVG(priority)
FROM requests [RANGE 1 DAY]

Figure 2.2: System model

the interface is CQL [32], a language for continuous queries. Section 2.6 elaborates on interfaces. Being able to expose arbitrary, high-level interfaces on top of the system enables reuse of existing applications developed against these interfaces.

The data processing model is the data management view onto the architecture and captures how data flows and is processed. It is a graph of entities that process data ("operators" as a first approximation), which we call slets, and entities that buffer and forward data, which we call channels. Section 2.4 elaborates on details of the data processing model. It is generic enough to fit different flavors of stream processing and thus enables interoperability.

The implementation model is the systems view onto the architecture. It specifies implementation details of slets and channels (e.g., interfaces) and adds a connector entity, which captures distribution in the model. Section 2.5 elaborates on details of the implementation model. It grasps elements as individually managed components, wires them using loose coupling, enables remote operation through connectors, and thus enables flexible deployment.

Ultimately, an instantiation of these entities is concretely implemented in some programming language. The generic parts of these entities as well as the platform itself are implemented by the platform
provider and the specific parts of these entities (e.g., operator logic, custom buffer implementation) are implemented manually as part of the application implementation or generated by the application (e.g., by a query compiler). Chapter 3 presents an implementation of the platform.

In summary, the system model formalizes various processing and implementation aspects of stream processing systems and makes them explicit (e.g., buffers) at its different levels of abstraction.

2.3 Use Case: Photo Exchange

This section motivates an example use case of personal information processing, namely how users process and exchange photos with a distributed stream processing platform. It illustrates the desired properties and complements existing stream processing systems used to derive the requirements of a generic stream processing model.

Figure 2.3 illustrates a simple application built on our platform that processes and exchanges photos. Throughout this thesis, we will explain this example step by step and use it to illustrate key aspects of the model and the platform. The desired system for processing personal information as streams—building on the platform—can be seen from four different perspectives: the end user, communities of users, developers, and service providers.

2.3.1 End User

For each single user, the system is intended as a platform for managing information. It turns arbitrary sources of data into data streams and pushes them through a mesh of processing stages to deliver them to the user at, potentially, several end devices. It is intended to support multiple sources and multiple sinks for the information flow, with the sinks being in most cases different end devices (e.g., a computer, a screen, a PDA, a mobile phone). The sources of information we are considering include photos and videos from digital
2.3. **USE CASE: PHOTO EXCHANGE**

Cameras, voice and text messages, reports on calls/text messages/e-mails from several addresses, RSS feeds, etc. The system allows the user to define the sources, define the sinks, and specify from a library of processing steps how the data will be combined/filtered/processed as it moves from the sources to the sinks. The system supports rules to decide which path the information should follow (e.g., depending on time of day, content, or user settings). Finally, it also supports the storage and caching of the data, offering functionality for querying the data in different ways.

In the photo application in Figure 2.3, the processing mesh on the upper left shows how a user can collect photos from different devices (photo camera, mobile phone) in a common place (channel *MyPhotos*), from where they are accessed by a photo widget running on the user’s desktop. A selection of photos is made publicly accessible through channel *Public*, which is fed by an slet *Filter*, which can implement any kind of filter from looking at photos’ metadata (e.g., EXIF) to face recognition to manual selection by the user. In addition, a low resolution version of all photos is made available in
channel \textit{LowRes}, which is fed by a scaling slet \textit{Scale}. The low resolution variants are copied to the internal memory of a digital photo frame (DPF) when it is connected, thus saving space and fitting more photos. The main part of the processing mesh is running on a network attached storage (NAS) device, which is always powered on. Camera and photo frame are connected via USB to the corresponding source and sink adapters running on the NAS device. Adapters for the phone’s camera and the desktop widget are running on the phone and desktop machine, thus already turning the user’s own setup into a distributed application.

\subsection{2.3.2 Communities of Users}

For communities of users, the system offers the possibility of linking the \textit{personal data processing and dissemination mesh} of each user with those of other users, thereby building an even larger mesh. The platforms communicate directly with each other, in a peer-to-peer manner. Connections are either established directly (connecting to the address of a known, remote platform) or by discovering channels of interest on devices in the network vicinity (see Section 3.1.9).

Through standard interfaces in the processing stages and the communication channels between processing stages, the system gives users the option of publishing any of the final or intermediate results of their personal processing mesh while other users can connect to that information and feed it into their own meshes. The scenarios that we are targeting include forwarding of notifications to other persons depending on conditions (e.g., user sets a \textit{do-not-disturb} flag, routing to other users or devices depending on the time of day), raising and propagating alarms, content based routing, etc. In this way, the system can be used not only to disseminate data among users but also to build data dissemination and processing meshes that are shared by a group of users and that directly feed on the personal meshes of each user. Taking it to the extreme, processing meshes could all be interconnected, in the same way that following a small number of links allows to reach almost any web page in the
2.3. USE CASE: PHOTO EXCHANGE

Internet. The system and the supporting platform are designed to function and operate at such Internet scale.

In the photo application in Figure 2.3, another user (Friend 1, on the lower left) accesses the public photos and merges them with her own photos (MyPhotos) into channel All. This channel contains all photos that are less than a week old and is connected to a sink on Friend 1’s mobile phone. The photos in channel All are synchronized with the phone and also available on the phone when it cannot connect to her laptop, which is hosting her personal processing mesh.

2.3.3 Developers

For developers, the system provides a very clean and rigorous system design. Processing happens in so called slets, which use standard interfaces for input and output and are language and OS independent. That way, the libraries of specialized (e.g., e-mail or SMS filters) or general (e.g., round-robin splitter) processing steps can be built independently of how they are combined by users. Communication between the slets happens through channels. Slets put data into channels or read data from them. Channels store/buffer/forward the data from the slets providing the input to the slets interested in their contents. There is no processing and there are no side effects in the channels. Channels also support callbacks to send request back along the reverse path of the processing and dissemination mesh. This allows slets to request specific data from given sources. Slets and channels can be added or removed at runtime, with the system dealing with the corresponding dynamic changes. The system has been designed to allow separate development of slets, of channels, and of the processing and dissemination meshes—which we envision may in fact be done by completely different people.

In the photo application in Figure 2.3, examples for slets that can be developed separately and then deployed and integrated into a processing mesh by different people are the adapters for the mobile phone, desktop photo widget, and photo frame. They are custom with respect to the sink they adapt (phone, widget, frame) but once
implemented they can be reused by any user who wants to use the same kind of sink and connect it to a channel containing photos.

2.3.4 Service Providers

Users of the system who do not have a device at hand that is connected to the Internet all the time can use the services of respective platform providers. These companies offer access to a platform instance for individual users, similar to mailboxes or accounts for existing photo, video, or social network sites. In contrast to the limited, application-specific possibilities that these sites offer, users can deploy and run their personal processing mesh with all the flexibility and different possible applications on their provider’s infrastructure. In contrast to the extreme of infrastructure as a service (IaaS), the platform as a service (PaaS) approach saves resources on the provider side (only one OS instance with the platform running on top is required). It also allows for more fine-granular optimizations in terms of, e.g., machine utilization or data locality compared to the coarse granularity of a virtual machine.

In the photo application in Figure 2.3, Friend 2 and Friend 3 host their personal meshes at a provider. The meshes of the two friends frequently exchange data and are thus located on the same physical machine.

2.4 Data Processing Model

The data processing model is the data management view on stream processing. At this level of abstraction it captures how data flows and is processed in terms of operators, buffers, and the units of data.

2.4.1 Requirements

Stream processing systems exhibit the following properties and requirements with respect to the data processing model. The use case
for personal information processing does not add its own set of requirements to the data processing model. Instead, it contributes details and specific flavors to some of the requirements gathered from stream processing system like, e.g., the importance of intermediate buffers as well-defined “connection points”. This is a strong indication that the concept of stream processing can indeed be applied to process personal information. The requirements for the data processing model are the following.

**Data Unit**

Stream processing systems process data as *discrete items*. Relational systems process streams of relational tuples and the unit of data is one tuple.

Some XML-based systems process streams of XML fragments, others streams of XML tokens. An XML fragment consists of a tree structure of one or more XML elements. While an XML fragment structurally is like a part of an XML document, XML streams are typically referred to as infinite sequence of XML fragments (as opposed to a finite XML document). XML tokens are the atomic elements of an XML document or stream, i.e., tags, attributes, and values. Processing a stream of XML tokens is similar to processing events of a SAX parser.

In the context of personal information, one unit of personal information corresponds to one data unit, e.g., one e-mail message, one photo, one video clip, or one text message.

**Data Format**

Streaming data formats are *varied*. Corresponding to the discussion of data unit above, the format of items of data streams can be relational tuples, XML fragments or tokens, or arbitrary text and binary data.

Besides payload items in the data stream, some stream processing systems implement techniques that utilize items or tuples with
special semantics like, e.g., negative tuples [23] or punctuations [49]. A generic stream processing platform that does not define or interpret semantics can treat them as elements in the stream like standard payload items.

**Operator Graph**

Tuples are processed through a *mesh of operators*. This mesh is typically referred to as the query graph or the processing mesh. Operators transform, consume, or emit items and potentially update internal state as a result of processing an input tuple. They can have *multiple input streams* and *multiple output streams*. Operators can also be *parametrized*.

**Buffers**

Stream processing systems use buffers to *temporarily or persistently materialize* a data stream. Operators that can only operate on finite sets of data (e.g., aggregation) use buffers to create windows over their input streams. These window buffers are mostly *implicit* and thus not part of an explicit processing mesh. *Explicit* buffers are used to materialize important intermediate results or serve as connection point to them. While also being important for many traditional stream processing applications, explicit buffers, which materialize data and serve as connection points, are particularly important for exchanging data in the scenario of processing personal information as streams.

**Data at Rest**

Most stream processing systems allow access to *data at rest* in addition to streaming data. This allows to join live streaming data with archived data and to archive results of stream processing.
2.4. DATA PROCESSING MODEL

Sources and Sinks

Stream processing systems provide adapters to interface with external streaming data sources and sinks. They form the boundaries in the processing mesh within which processing in the system happens.

2.4.2 Entities

The two fundamental building blocks of stream processing are operators, which process data, and buffers, which forward and buffer data. The data processing model of the exoengine architecture thus considers operators, which we call slets, (pronounced s-let, short for streamlet), and buffers, which we call channels, as distinct entities. In contrast to models that only consider a mesh of operators with implicit buffers, this model enables to reason about buffers explicitly, which is an important property when federating multiple applications. Section 2.4.6 provides insight into this aspect. Figure 2.4 illustrates the model as a mesh of channels (rectangles) and slets (ovals). Data flows from sources on the left through the mesh of slets and channels to sinks on the right, as illustrated by the direction of the arrows. Slets interact with channels by using input and output ports and every port connects to exactly one channel. In the figure, ports are implicitly illustrated as the places where arrows enter or leave slets. The data processing model considers the following entities.
Source

A data source is any external device, application, or software component that emits data. External refers to the property that it is not part of the processing mesh of slets and channels but needs to be adapted. There are no specific requirements for data sources except that they provide at least one means of accessing the data they provide. This can range from a shared memory region to a local or network pipe to high-level communication mechanisms like, e.g., SOAP RPC.

Sink

A sink is any external device, application, or software component that consumes data. Like sources, there are no specific requirements for sinks except that they provide at least one means of consuming data.

α-Slet

α-slets provide a clean, well-defined interface for exchanging data with external data sources. In Figure 2.4 they are located at the left, source-side edge of the processing mesh. An α-slet adapts and conceptually wraps an external source through one of the interfaces that the source provides. α-slets have no input ports, they get data only from the source they adapt. If the source can be divided into well-defined subparts like, e.g., folders of an e-mail account, each subpart corresponds to one output port of the α-slet. Similarly, every individual source is adapted by one particular instance of an α-slet, resulting in a one-to-one relationship between α-slets and external sources as well as a one-to-one relationship between ports of an α-slets and subparts of an external source.
2.4. DATA PROCESSING MODEL

\( \omega \)-Slet

\( \omega \)-slets provide a clean, well-defined interface for exchanging data with external data sinks. In Figure 2.4 they are located at the right, sink-side edge of the processing mesh. An \( \omega \)-slet adapts and conceptually wraps an external sink through one of the interfaces that the sink provides. \( \omega \)-slets have no output ports, they output data only to the sink they adapt. If the sink can be divided into well-defined subparts like, e.g., a logging facility that records multiple streams at the same time, each subpart corresponds to one input port of the \( \omega \)-slet. Similarly, every individual sink is adapted by one particular instance of an \( \omega \)-slet, resulting in a one-to-one relationship between \( \omega \)-slets and external sinks as well as a one-to-one relationship between ports of an \( \omega \)-slet and subparts of an external sink.

\( \pi \)-Slet

\( \pi \)-slets resemble conventional operators. They process data and can have any number of input and output ports. \( \pi \)-slets can execute arbitrary operations on data accessed through upstream channels. They can drop data (filter), aggregate it into internal state, join it with data from another input port, transform it, or create and emit new data. \( \pi \)-slets potentially emit data through their output ports as a result of processing input data. They can be parametrized with configuration information that influences their behavior. Examples include the value(s) for a filter attribute or, in the photo application in Figure 2.3, the Scale slet, which can be parametrized with the desired result size and compression settings. Parametrization facilitates the reuse of slets in other applications, e.g., where different sizes for photos are needed.

Channel

Channels forward and optionally buffer data between producing and consuming slets. Any number of input and output ports can connect to a channel. Exposing the functionality of forwarding and
buffering as an explicit, first-class entity between every operator is a distinct feature of the model. Subsequent sections elaborate on the properties achieved by this approach.

2.4.3 Processing Mesh

Slets and channels are connected in a processing mesh, similar to a query plan or operator graph. Every output port of an slet connects to the input of one channel. Multiple output ports can connect to the input of one channel. Likewise, every input port of an slet connects to the output of one channel and multiple input ports can connect to the output of one channel. Channels cannot connect directly to other channels and slets cannot connect directly to other slets. In situations where the channel (e.g., simple push-through) or slet (e.g., generic merger) can be omitted, the representation of an application in the data processing model must still keep them. In the application in Figure 2.3, the All channel of Friend 1 merges her own photos (MyPhotos) with the publicly accessible photos of the user’s channel Public through a generic merger slet, as it is required by the model.

The underlying implementation, however, can omit channels or slets provided all properties of the model are preserved. For example, if a pass-through channel between two slets is removed in the implementation it must still be possible to attach other slets to this channel, e.g., by dynamically inserting the channel in the implementation again when the other slet has to be attached.

The processing mesh can have cycles and the entities setting up and manipulating the mesh must ensure that this has no adverse effects. Supporting cycles is necessary, e.g., to utilize a channel as storage for data at rest like, e.g., materialized aggregate values. Triggered by data arriving at a different input port, an slet connected to input and output of the same channel could retrieve an item from the channel and save an updated value of the item back into the channel.
2.4. DATA PROCESSING MODEL

2.4.4 Data Items

Data is processed as discrete items (or tuples) that flow from the sources on the left through the processing mesh to the sinks on the right. The data processing model does not impose any specific data format per se. \( \alpha \)-slets packetize data from external data sources into discrete parts—items.

Items typically represent a unit of information that corresponds to the stream’s content like, e.g., a relational tuple, an XML fragment or token, a single photo, video clip, or e-mail message. Every item is associated with metadata that contains a type description of the item and a timestamp indicating when the item container was created by the \( \alpha \)-slet.\(^1\) Items are immutable and can either be short-lived, for immediate processing and subsequent discarding, or long-lived, for temporary or persistent buffering of data.

As the model does not impose a specific data format, \( \pi \)-slet implementations depend on the application and thus a platform implementation cannot provide predefined \( \pi \)-slets operating on any specific kind of data. However, a platform implementation can and should provide a set of generic \( \pi \)-slets that do not operate on the item data but on items as a whole. Examples of these include generic merging of channels or round-robin distribution of a channel to multiple channels.

2.4.5 Stream and Store Data Paths

The data processing model distinguishes between two different data access facets: the stream and the store data path. Channels implement both facets and allow for access to the stream and store data

\(^1\)While typically not required for traditional stream processing systems, which exchange items of a defined type, the type description metadata allows to specify complex type descriptions that can be useful in the context of personal information processing. For example, in our implementation, which is presented in Chapter 3 and which is based on Java, the type description contains one or more fully qualified names of a Java type.
paths. Slets can access data from channels in any combinations thereof.

Data arriving on the stream data path resembles the actual streaming data and is pushed to the channel by upstream slets. Every downstream slet that connects to a channel specifies its buffer requirements—a window over the streaming data. The channel then manages a corresponding buffer for this slet and buffers items that arrive on the stream data path accordingly. In the photo application in Figure 2.3, the All channel of Friend 1 keeps photos that are at most one week old for the ω-slet connecting to the mobile phone. Section 2.5.5 discusses different access patterns to data on the stream data path and Section 3.2 describes an example implementation of a corresponding declarative buffering framework.

In contrast to data arriving on the stream data path, which arrives by virtue of the upstream processing mesh, data delivered on the store data path arrives as response to explicit requests from the downstream processing mesh. A pull request on the store data path returns all items in the channel. The result of a pull request on the store data path is returned in a buffer that is (logically) separate and independent from the stream data path.

### 2.4.6 Channels as Views

Channels can be seen as views over the upstream processing mesh. Similar to views in traditional databases, they contain the results obtained by processing source data (data sources in this model, data tables in databases) with the view definition. In databases, the view definition is a query and in this model it is the part of the mesh that is connected to the channel’s input.\(^2\)

On the store data path, a channel holds all data produced by the upstream slets and channels it is connected to. Analogously, on the

\(^2\)The upstream processing mesh itself could also be specified at a higher level, similar to a declarative query. However, since the processing mesh can also be specified other than through such a declarative query, this is not a requirement and the actual processing mesh is the view definition in our model.
stream data path, a channel will receive all future items that pass the processing of the upstream slets and channels it is connected to.

Channel implementations can persist their contents, resembling materialized views in databases, or request data on demand when being polled by a downstream slet. In the photo application in Figure 2.3, the LowRes channel contains low resolution copies of all photos in the MyPhotos channel. If LowRes is operating in materialized mode, it will physically have copies of the scaled photos. Else it will request the slet it is connected to (Scale) to return all items when it itself receives a request from the downstream DPF sink slet. This will trigger that the Scale slet fetches all photos from MyPhotos, scales them, and returns the scaled photos to the LowRes channel. Depending on the materialization mode of MyPhotos, requesting all photos from it can itself result in a request to its upstream mesh and potentially in a cascade through the whole upstream mesh until eventually α-slets and the ultimate sources they adapt are reached. Materialization allows to trade storage requirements off against result latency.

Figure 2.5 illustrates the concepts of channels as views of their upstream mesh and their relation to the stream and store data paths. It shows how to model an example application that processes e-mail messages with this model. E-mail has both a stream facet (new e-mail messages arriving) and a store facet (e-mail messages stored in the mailbox). In the example, an IMAP server with three folders acts as data source and three mail clients act as data sinks. The server is adapted by an IMAP α-slet providing one output port per IMAP folder. The first sink, Mail S1, accesses the Inbox channel containing all mails stored in the inbox folder. Mail S2 accesses the All channel, containing all e-mails stored on the mail server, which is the result of a generic merger slet M merging Inbox, Sent, and Archive channels. Mail S3 accesses the Matter X channel which is the result of a filter slet X that filters e-mail messages concerning a certain matter X (e.g., an ongoing project). When a sink queries for e-mail messages, e.g., containing a certain string in the subject line, the queries are applied to the messages contained in the channel
that the sink is connected to, resulting in a conjunction with the operations implemented by the upstream mesh—the view definition. Likewise, when new messages are pushed into the mesh on the stream data path it will only be pushed on to matching channels; e.g., filter slet \( X \) drops any message pushed to it that does not concern matter \( X \).

The example presented in Figure 2.5 also illustrates how stream and store data paths are integrated in the model and how they can be combined within an application. Modeling access to streaming data and data at rest in a single channel entity instead of two separate entities is particularly advantageous for hybrid data sources, like, e.g., the IMAP server, which provide data as streaming data and as data at rest. The processing mesh and thus the views like, e.g., \( \text{Matter } X \) are automatically the same and consistent. However, it is still possible to access stream and store data paths of hybrid sources independently through two discrete parts of the processing mesh that only share the \( \alpha \)-slet adapting the source.

Section 2.5.8 describes the processing model of slets and channels and provides more details on the integration of the stream and store data paths.

### 2.4.7 Merging and Replication

If multiple ports are connected to the input of a channel, the channel merges incoming items on the stream data path as they arrive. On the store data path, the request will be forwarded to all connected ports and the results merged.
Similarly, when multiple ports are connected to the output of one channel, the channel replicates data to all connected ports. Though variations of channels could be implemented that support, e.g., round-robin distribution to multiple downstream slets, this is not allowed by the model. Selective data routing decisions taken in the channel would void their status as views over the upstream processing mesh as it would violate the principle that a channel holds all the data created by the upstream mesh it is connected to. Thus, selective replication techniques have to be realized by adding a respective slet to the mesh.

In the photo application in Figure 2.3, the user’s channel Public replicates its contents to all components connected downstream; the slets of the meshes of Friend 1, Friend 2, and Friend 3 connected to channel Public.

2.5 Implementation Model

The implementation model is the systems view on stream processing. At this level of abstraction it captures how components of a stream processing system are implemented and interact.

2.5.1 Requirements

Stream processing systems exhibit the following properties and requirements with respect to the implementation model. The use case for personal information processing as well as the desired improvements in terms of interoperability, extensibility, integration, and deployment contribute significantly to the requirements for the implementation model. Therefore, and in contrast to the data processing model, which generally closely resembles the ordinary data processing model of stream processing, the implementation model significantly extends the state of the art of implementing a stream processing system. The requirements for the implementation model are the following.
Distribution

_Distributed operation_ is a requirement for a distributed stream processing platform and should be an inherent property of the implementation model.

Dynamism

Changes to the processing mesh can be _dynamic_ and happen at runtime. Examples include attaching a new sub-mesh to intermediate results of an existing mesh, shutting down a computer that was running a personal processing mesh connected to other friends’ meshes, replacing parts of a processing mesh with a different implementation, and changing the deployment of a distributed application.

Management of Individual Components

Components of the processing mesh have to be _managed individually_, including the management of instance-specific configuration (e.g., the value for a filter attribute) and state information (e.g., the current value of an aggregation). In conjunction with support for a dynamic processing mesh, which manages the binding between components, this enables the complete management of applications.

Push and Pull Processing

The execution model of existing stream processing engines can be _push or pull_ driven. Supporting both is required for the integration of applications defined for different SPEs.

Different Implementation Languages

Different SPEs are implemented in _different programming languages_. Similar to push and pull processing, supporting engines implemented in different programming languages is required for the integration of applications defined for different SPEs.
Implicit and Explicit Buffers

Data is buffered in both *implicit and explicit buffers*. As introduced in Section 2.4.1 of the data processing model, buffers used in stream processing systems can be implicit like, e.g., windows kept for an individual operator or can be explicit like, e.g., explicit materialization points for important intermediate results.

Custom and Rich Interaction

Existing stream processing systems utilize *custom and rich interaction* between their components. This includes the data plane, where operators and buffers interact through, e.g., an index-based mechanism, as well as the control plane, where, e.g., a query optimizer instructs an operator to flush its state before it is being replaced. To leverage existing SPEs with their specific performance characteristics it is imperative to also keep their intrinsic implementation details.

2.5.2 A Service-Oriented Architecture

The implementation model proposes a component- and service-oriented architecture (SOA) for the implementation of stream processing systems. In contrast to other architectures like, e.g., a plain object oriented architecture, a component- and service-oriented architecture exhibits advantageous key properties like individual component management, dynamic service binding, clear service interface abstractions, and the possibility to augment service instances with arbitrary, declarative attributes and filter services with respect to these attributes. The implementation model leverages these SOA properties to meet the requirements identified above. The subsequent sections elaborate on the details of the SOA-based implementation model.
2.5.3 Entities

\(\pi\)-, \(\alpha\)-, and \(\omega\)-slets and channels are also entities in the implementation model and herein reflect their roles in the data processing model as discussed in the preceding section. In addition, the implementation model adds connectors as a third entity to slets and channels. Every component exposes one or more specific service interfaces for interaction and each instance of a component is subject to life cycle management by the platform.

Connectors add a level of indirection between slets and channels. Figure 2.6 illustrates the implementation model. Slets’ output ports are connected to the input of a channel through connectors. The output of the channel is connected to slets’ input ports through connectors. The implementation model separates concerns of processing (slets), storage (channels), and communication (connectors) into separate entities. This separation facilitates capturing resource requirements and implementing respective optimizations. Although slets’ ports have been made explicit they are treated as an integral part of slets and not as individual, independent components.
Slet

$\pi$-slets have any number of input and output ports. They can create and remove ports during initialization or at runtime. Individual configuration properties can be assigned to every instance of an slet to customize the behavior of the particular instance.

$\pi$-slets expose an Slet service interface used for managing their configuration and state. Additionally, every port of an slet offers an InputPort or OutputPort service providing methods for bringing data to the port or requesting data from it, respectively.

The same rules and properties apply to $\alpha$- and $\omega$-slets with the exception that $\alpha$-slets have no input ports and $\omega$-slets have no output ports. Instead, they connect to the source or sink they adapt through the means that source or sink provide. $\alpha$-slets packetize data from external data sources into items. They optionally have to convert, deserialize, or embed data into a general data type of the implementation language of the platform, e.g., a Java object, if the source is implemented in a different language. Likewise, $\omega$-slets unpack data from the item container of the platform and deliver it to the sinks in the sinks’ native formats.

In the photo application in Figure 2.3, $\alpha$-slets adapt the user’s photo camera and mobile phone. In the example of the camera, the $\alpha$-slet is running on the home gateway together with the remaining mesh of the protagonist user and interfaces with the camera via USB. Similarly, the digital photo frame interfaces via USB when it is connected to the home gateway. In contrast, the source gathering the photos taken on the mobile phone as well as the sink displaying photos in a desktop widget run on the phone or the desktop machine, respectively. There they form a small processing mesh themselves that interacts with the mesh on the NAS through a remote connection, as discussed in Section 2.5.6.
CHAPTER 2. DESIGN

Channel

Channels forward data between slets and are the well-defined connection point to access (intermediate) results. As such, they host explicit buffers if, e.g., intermediate results are materialized.

Channels provide a Channel service interface for management. Every channel also exposes a ChannelInput service interface used by input connectors and a ChannelOutput service interface used by output connectors. These interfaces provide methods for data exchange between connectors and channels.

Connector

Connectors provide a level of indirection in the interaction between channels and slets and host implicit buffers as well as distribution in the model. Slets’ ports connect to connectors instead of directly to channels. Multiple output ports can be connected to one input connector. Multiple input connectors can be connected to the input of one channel. Likewise, multiple output connectors can be connected to the output of one channel and multiple input ports can be connected to one output connector.

Connectors provide a Connector service interface for management. For data exchange, input connectors expose CICI and CICO service interfaces (ChannelInputConnectorIn and -Out) and output connectors COCI and COCO service interfaces (ChannelOutputConnectorIn and -Out).

Output connectors host implicit buffers (e.g., windows for the downstream slet), which enables the sharing of one physical buffer for multiple logical windows kept for each connected slet. Section 3.2 presents an example implementation of a buffering output connector. In a local setting, input connectors typically only forward requests between channels and ports. Thus, they can be omitted as optimization and channels and ports interact directly with each other. Similarly, if no implicit buffers are hosted in an output connector, also the output connector can be omitted in the local case.
In a distributed setting, connectors provide means for accessing a channel in a remote instance of the platform. They act as proxies for the remote channel in the local platform and—typically in the case of output connectors—host implicit buffers for the downstream slets accessing the remote channel. In addition to locally buffering data of the remote channel for windows requested by connected slets, they can also implement further optimizations like, e.g., caching the whole contents of the channel for fast access on the store data path as well. Section 2.5.6 discusses and illustrates the details of distributed operation using remote connectors.

2.5.4 Processing Mesh

The processing mesh in the implementation model resembles the processing mesh in the data processing model, with connectors added between slets and channels. In Figure 2.6, arrows between components are bidirectional as they represent the component interaction in terms of service method invocations rather than in terms of data flow. Ellipses between slets or connectors indicate that any number of instances thereof can exist and interact with one instance of a connector or channel, respectively. In Figure 2.6, each component itself represents its service interface for management (Slet, Channel, Connector). Service interfaces used in data exchange are depicted using thick, red bars.

To manage a component, the platform invokes methods on the management services. To exchange data, components invoke methods on the InputPort, In, Out, and OutputPort services of their connected peers. It is a common misconception that service-oriented design is not suitable for the data plane of highly performance critical applications, as the dynamic binding, lookup, and invocation of services would cause significant overhead. We will show in the implementation (Section 3.1.5) and in the evaluation (Chapter 5) that it is possible to achieve the properties and behavior of a dynamically composed, service-oriented software system with negligible overhead on the data plane.
2.5.5 Data Access Facets and Mechanisms

All components of the model implement both the stream and store data path facets. Access to data at rest through the store data path is naturally implemented using a pull mechanism, invoked from the downstream mesh. The most basic operation is to request all items from the upstream component. Though the pull request itself flows from right to left (from sinks to sources), data—the reply to the request—always flows from left to right (from sources to sinks).

Streaming data is normally pushed by the upstream mesh, which corresponds to the nature of streaming data. The most basic operation is to push an item to the downstream component. Thus, the puristic, basic interfaces for data exchange in the implementation model define and consist of two methods:

**push(item):** push an item to the input of an slet, channel, or connector

**pull():** request items from the output of an slet, channel, or connector; returns a set of 0...n items

The basic functionality of push for streaming data and pull for data at rest is sufficient to model stream processing applications. However, existing operators and engines for streaming data can also be pull-driven. Furthermore, more sophisticated access variations to streaming data like, e.g., through a window, are possible for both push- and pull-driven operators. Every input port of a downstream slet that connects to a channel specifies its buffer requirements and its access method. The connector between channel and port then manages a corresponding buffer for this port. It buffers items that arrive through the stream data path and, if requested, notifies the slet’s port of the arrival of new elements (in case of pull access) or directly pushes one or more items to the port (in case of push access). Section 3.2 shows an example of a buffering output connector that supports different windows, notification variants, and push and pull access to the stream data path.
By using connectors that provide buffers for individual ports, it is possible to combine push- and pull-driven slets at the boundary of channels. Furthermore, the logical separation of stream and store data path in the components allows to implement and combine both facets in the model as well, in particular supporting the concept of channels as views of the upstream processing mesh.

### 2.5.6 Distribution

Channels contain intermediate and final results and resemble a view of their upstream processing mesh (see Section 2.4.6). Hence, they are typically the entity on a platform that a remote application, running on another platform instance, wants to connect to. Likewise, in the distributed setup of a single applications, channels contain intermediate results to be exchanged between upstream and downstream slets and thus represent the places in the processing mesh where the boundaries between nodes can be drawn.

A channel hosted on one instance of the platform is accessed by another, remote instance of the platform using *remote connectors*, which are illustrated in Figure 2.7. Remote connectors encapsulate communication between the connected channel and slets and the instances of the platform hosting them. One half of a remote connector is installed in the platform where the channel resides and implements either the CICO or COCI service interface. The other half of the connector is installed in the remote platform where the channel needs to be accessed and implements either the CICI or the COCO service interface. This latter half represents (proxies) the channel in the remote platform. For every remote platform instance that accesses the channel, one remote connector is used to serve all slets on that instance.

In the illustration in Figure 2.7, the channels hosted on the platform on the left hand side are accessed by the platforms on the right hand side. The upper channel is accessed by both platforms on the right and thus two input-halves of a remote channel output connector are connected to it. On the upper right platform, the channel is
accessed by two slets. Both connect to the same output-half of the remote connector installed on their platform. The lower channel is accessed by the platform on the lower right only and thus only one remote connector is connected to it.

The two parts of a remote connector communicate with each other and act as one logical connector. Typically, the half installed on the platform that wants to access the channel (the two platform instances on the right hand side in Figure 2.7) implements a local buffer that satisfies the buffer requirements of all the connected slets.

In addition to plain buffering on the stream data path, *smart connectors* can cache the content of the channel they are representing, serve requests from their own buffer, and thus reduce latency and save bandwidth. Similarly, when the connection between the smart connector and its counterpart on the remote platform is not available, the connector can autonomously work in offline mode. The wiring of slets to the connector can be left unchanged, because transitions between online and offline mode happen inside the connector, behind the service interfaces. In contrast, standard connectors are simply removed (disconnected) from the mesh if the network commu-
2.5. IMPLEMENTATION MODEL

communication fails. This is supported by the model as any other dynamic change to the processing mesh.

Using connectors as local proxies for channels serves the following principal concepts:

**Local buffers:** Logical buffers (windows) over the streaming data received through a remote channel can be directly accessed as local memory because they are proxied physically in the local connector part. Likewise, smart connectors which, e.g., buffer the contents of a materialized channel for the store data path can also provide access to their buffer instantaneously as local memory.

**Performance:** By moving the buffer for an slet to the node on which the slet runs, access times are reduced and bandwidth can be saved. This can include proactive caching of new items arriving on the stream data path, caching of results on the store data path, and keeping a full and up-to-date copy of a channel’s buffer.

**Transparency:** Remote connectors expose the same interfaces for data exchange as local connectors. Thus, they can be accessed like a local connector and seamlessly fit into the implementation model and its processing mesh.

**Offline operation:** When the connection between connector and remote channel is not available, the connector can autonomously work in offline mode. The composition of slets with the connector can be left unchanged, as the transition to offline mode and back happens inside the connector.

In the photo application in Figure 2.3, the connector connecting *Friend 1’s* mobile phone to channel *All* is a smart connector that supports offline operation. The half of the connector that is installed on the mobile phone caches the channel’s content (photos less than seven days old). Thus, these photos are also accessible on the phone when there is no connection to the laptop.
2.5.7 Basic Interfaces and Extensibility

Section 2.5.5 defined the basic interfaces for data exchange of the implementation model. They provide a `push(item)` method on the stream data path (In and `InputPort` services, see Figure 2.6) and a `pull()` method on the store data path (Out and `OutputPort` services, see Figure 2.6). It also motivated the need for an extension of these interfaces at least between input ports and connectors’ Out services to support pull access mechanism or windowed access patterns to streaming data.

Additional advanced functionality and optimizations in a stream processing system require that the interfaces for data exchange between all components is extensible. Examples include existing rich interaction between operators and storage instances (e.g., index-based access to data at rest or streaming data materialized in windows, batch access to all items in a window), or the possibility to augment a pull request on the stream data path with a query expression to push selectivity towards data sources—particularly when channels are not operating in materialized mode.

Slet and channel implementations can extend the basic In, Out, `InputPort`, and `OutputPort` service interfaces for data exchange (illustrated as thick bars in Figure 2.6). Thus, they can interact through additional methods as needed, without losing the property that the platform manages the dynamic binding between components’ services.

Connectors also need to support the methods of the extended service interfaces between slets and channels. If they are part of the extension like, e.g., they contain a buffer to translate between push and pull access to the stream data path or implement windows, they require custom implementation. Else, they can be omitted in the local case. In the distributed case, standard remote connectors only need to pass method calls through. Thus, they can be created automatically by the platform by inspecting the extended interfaces of the channel and ports they connect by using, e.g., reflection. This enables centralized implementations to operate in a
distributed way. Distribution-aware implementations of SPEs can provide implementations of smart connectors for enhanced remote operation, as described in Section 2.5.6.

In addition to extensions of interfaces on the data path (data plane), implementations of components can also extend their management interfaces (control plane) to, e.g., allow an optimizer to replace parts of the processing mesh in a controlled manner (e.g., instruct buffers to pause and operators to persist internal state) or implement a rich configuration mechanism.

2.5.8 Component Processing Model

The processing model of slets, channels, and connectors is primarily dictated by $\alpha$-, $\pi$-, and $\omega$-slets, which feed, process, and consume data and thus drive the processing mesh.

Slet

A $\pi$-slet can become active under three circumstances. The first case is the arrival of new data on the stream data path of a channel connected to an input port. The second case is the arrival of a request on the store data path from a channel connected to an output port. The third case is an access to the slet’s management interface (e.g., updating the slet’s configuration). Depending on what event caused the slet to become active, it can execute an arbitrary combination of operations valid for that cause. Table 2.1 lists which operations are valid for activity caused by new data arriving on the upstream stream data path, a request on the downstream store data path, or an access to the management interface. The operations are:

**Access stream data:** If the slet’s ports are connected to channels through connectors that provide buffers/windowing on the stream data path (because the slet stated its requirements accordingly), it can access these buffers.
Access store data: The slet can pull on the store data path of any channel connected to its input ports.

Update state: The slet can update its internal, volatile state.

Update configuration: The slet can update its own configuration, which will be persisted by the platform.

Emit items: The slet can emit (push) any number of items to any of its output ports.

Return results: The slet must return results to the pull request it received through a downstream channel. The result set can be empty. This operation is compulsory, must only be executed once, and forms the end of processing a request on the store data path.

Modify ports: The slet can create and destroy ports or change buffer requirements of existing ports.

Looking at the first two rows of the table, we can see that slets can pull data from their upstream stream and store data paths regardless of being activated on an upstream stream data path or downstream store data path. This capability bridges both access facets to data and thus is the key to the integration of stream, store, and hybrid data sources.

Check marks for accessing stream and store data paths initiated through access to the management interface (first two rows, last column in Table 2.1) are in parentheses to indicate that a scheduler could control the execution of an slet. However, a scheduler or any other management component of an SPE can only schedule the execution time but not create the actual event that causes the specific slet to execute. Such an event will always be either a new item arriving on the stream data path or a request arriving on the store data path. If a scheduler-driven slet does not process the event immediately, processing of the event can be considered suspended and
2.5. IMPLEMENTATION MODEL

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stream</th>
<th>Store</th>
<th>Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Stream Data</td>
<td>✓</td>
<td>✓</td>
<td>(✓)</td>
</tr>
<tr>
<td>Access Store Data</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Update State</td>
<td>✓</td>
<td>✓</td>
<td>(✓)</td>
</tr>
<tr>
<td>Update Configuration</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Emit Items</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return Result</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Modify Ports</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.1: Slet processing model

resumed at a later point, when the scheduler triggers it. Therefore, the set of valid operations relating to the original event still applies.

The processing model for α- and ω-slets is very similar to that of π-slets, with a few differences. Push or pull requests can arrive through the external sources or sinks they adapt and the same rules apply as for π-slets. In terms of scheduling, α- and ω-slets typically drive the execution of slets and channels in the mesh, either using a thread or by external request. α-slets can access stream data or store data if the data sources they are connected to support the respective kind of interaction. The same holds for ω-slets and sinks in terms of emitting items and returning results.

Channel

Channels forward and buffer data between slets. Thus, their processing model is dictated by the processing model of slets. When a new item arrives on the stream data path, they forward the item to all downstream connectors. If a channel operates in materialized mode, it also adds the item to its buffer. When a pull request arrives on the store data path, channels forward the pull request to all connected upstream connectors and return the combined results to the requesting downstream connector. If a channel operates in ma-
CHAPTER 2. DESIGN

terialized mode, it directly returns its contents instead of forwarding the request upstream.

Connector

Connectors act as proxies for channels. As such, they also primarily forward data and operate like channels. Connectors that maintain buffers (windows) on the stream data path for connected slets might not push items arriving on the stream data path to the connected slet but instead only add them to the buffer until it is requested by the slet. Likewise, fully caching remote connectors, as introduced in Section 2.5.6, might add a new item arriving on the stream data path to their internal buffer like materialized channels, independently of any connected slets’ buffering requirements.

Advanced Implementation and Custom Interaction

Extensions made to the basic interfaces to support the rich and custom interaction between operators and buffers of nontrivial stream processing systems require additional processing steps like, in the example of a materializing channel, adding an item to an internal buffer in addition to pushing it to downstream connectors. Such processing is allowed as long as the following properties hold:

**View consistency:** The data processing model’s property of channels being views of the upstream processing mesh (see Section 2.4.6) must hold. In particular, this implies that no data processing (e.g., filtering) or selective routing (see Section 2.4.7) happens in the channel.

**Basic interface support:** The implementation model’s basic interfaces (push(item) for the stream data path and pull() for the store data path) must be supported to allow for connecting other processing meshes (e.g., for federating applications) at any time.
Likewise, implementations can generally be written in any implementation language, as long as they implement the basic interfaces. A loosely coupled, service-oriented architecture does not preclude the use of arbitrary implementation languages.

2.5.9 Component Life Cycle Management

Every component in the model provides a management service. At runtime, the platform interacts with the component through this service to perform common, generic management tasks like monitoring, suspending, and restarting it, or exchanging individual configuration data. The platform takes care of managing and persisting component configuration data (e.g., the particular query an engine in an slet is executing), internal component state (e.g., the counters of an aggregation operator wrapped as an individual slet), and applications (e.g., which instances of slets are connected to which instances of channels and thus form an application).

In addition, the life cycle of every slet, connector, and channel is individually managed. The primary runtime states are running and suspended, whereas the former is the normal state of operation and the latter is used to suspend the component. Components can be suspended for, e.g., temporarily disabling an application as a whole (or parts of it), shutdown of the whole platform, or moving a component to a remote platform. The component itself, however, cannot tell the difference and only has to implement the transition from running to suspended and back. Typically, $\alpha$- and $\omega$-slets need to shut down and reestablish their link with the external entities they adapt. Channels and $\pi$-slets, on the other hand, serialize and deserialize their internal state and buffers. The platform manages the serialized state similar to the individual configuration information of each component.
2.5.10 Application Setup and Management

An application is implemented as a specific mesh of slets and channels. Specific refers to which types of slets and channels are used, their individual configuration, and how they are wired. The low-level mechanisms for configuring components (management interface) and wiring them (dynamic service binding) have been described in the previous sections. Entities that actually create, configure, wire, and modify instances of components in a processing mesh are named application builders.

Application builders are also the entities in a system that expose the interface to the system, accept an application definition through this interface, and implement the application by creating, configuring, and wiring instances of slets, channels, and connectors. Examples include the query interface of traditional SPEs or a graphical user interface where users visually compose their personal processing mesh. The next section discusses the interface of stream processing systems and application builders. However, the term application builder subsumes all entities of a system that modify the processing mesh, thus also including, e.g., optimizers.

To facilitate the implementation of application builders, we propose a management module to be part of the platform. This management module provides high-level abstractions of the tasks of creating, configuring, and wiring instances of slets, channels and connectors and hides their details. Application builders interact with the management module and do not need to bother with the low-level details of the SOA-based implementation model. Every platform instance provides a management module, through which application builders can also interact with management modules of remote instances. The management module also acts as synchronization point for all operations modifying the state of the platform. Thereby, it ensures consistency and provides the possibility to approve or reject operations.
2.6 Interface

The interface layer of the system model considers the high-level abstraction provided by the system and how clients interact with it. Applications are specified against these abstractions.

2.6.1 Requirements

Every stream processing system has its own specific interface. Examples include command line interfaces that accept a query string in a streaming SQL dialect, an object-oriented Java API to instantiate the engine and provide a streaming XQuery expression to it, and a graphical user interface that allows to compose a processing mesh using drag and drop. Therefore, the only hard and essential requirement with respect to the interface is flexibility.

The interface of a stream processing system is what defines how it is used. Therefore, to be able to leverage existing stream processing systems and keep their peculiarities, which make them apt for particular applications, we must also be able to expose the same interface and thus run the applications that already exist for this system without modification.

The use case for personal information processing further confirms that supporting arbitrary interfaces is an important requirement. Applications over personal information are not going to be defined as a set of declarative queries expressed in a SQL or XQuery dialect. Rather, these applications require interfaces that may be more restricted but intuitive to use for the every-day user. For example, the interface to compose the application in Figure 2.3 could very much resemble the illustration of the processing mesh in the figure itself.

2.6.2 Application Builders

Application builders are the entities that expose the interface to the system. Section 2.5.10 introduced application builders and their role in setting up and managing applications. In terms of providing
the interface, they adapt the control plane of the outside world (the clients using the system) like the $\alpha$- and $\omega$-slets adapt the data plane of the outside world (sources and sinks). There are no specific requirements for exposing the interface except that clients must be able to access it like in the unmodified system. In terms of mechanism, this can range from a shared memory region to a local or network pipe to high-level communication mechanisms like, e.g., SOAP RPC to a graphical user interface. In terms of semantics, this can range from programmatic instructions to declarative queries to visual composition.

A query compiler that has been turned into an application builder still fulfills its original task: it translates a high-level description of the application into a processing mesh. The difference, however, is that instead of instantiating buffers and operators manually and wiring them themselves, application builders instruct the platform to create instances of slets and channels and how to wire them. The low-level details of these steps as well as maintaining individual operator state, configuration, and wiring information are handled by the platform.

### 2.7 Federation and Semantics

Federation of applications using the exoengine architecture happens by attaching an application (processing mesh) to one or more channels of another application. It is technically possible to host both applications on the same platform. However, the motivating applications both in traditional stream processing as well as in personal information processing suggest that the main property of federating applications is to connect applications that run in different and independent authoritative domains (companies, government authorities, individual users) in a controlled and well-defined manner. Therefore, most federations will happen between platform instances, through a remote connection to the channel of interest provided by one application and accessed by the other. The architecture and model
proposed fit this kind of federations well, supporting their inherent requirements:

**Well-defined connection point:** Channels represent views over the upstream processing mesh. They contain intermediate or final results. Thus, they make ideal well-defined connections points for applications defined in this model.

**Well-defined interface:** The basic operations defined for channels provide well-defined interfaces to exchange data. For many federations the basic operations are sufficient, as no tight integration between applications is desired, but primarily independent operation with a easy way of exchanging data. More sophisticated, agreed-upon interfaces can still be implemented and used between applications exactly like extended interfaces can be used within the components of a single application.

**Independent operation:** Applications that are federated run in different and independent authoritative domains. Operation is not orchestrated across applications and thus the operation of one application must not be negatively influenced by the operation of the other application. The model supports dynamic changes to the processing mesh to deal with, e.g., stuck components, faulty network links, or the plain shutdown of the other application.

**Independent development:** Like independent operation, applications and the systems they are built on are also developed independently. The implementation model supports different implementation languages to support independently developed systems. Behind the well-defined interfaces, applications and systems can also evolve safely without requiring changes to the federated application each time a change is made to a local application.

The properties of the model and architecture proposed support the syntactic, technical interoperability of different stream process-
ing systems. The model does not dictate semantics or provide inherent properties for semantic compatibility of SPEs. Semantic compatibility is an orthogonal aspect to the work presented in this thesis. In our research group, we explore semantic aspects of integrating heterogeneous SPEs in the MaxStream project [9].

2.8 Summary

This chapter presented the system model and architectural philosophy of our approach to virtualizing stream processing. The exo-engine architecture proposed is similar to the exokernel architecture for operating systems in that it dictates as few abstractions and policies as possible to support a wide range of different SPEs and application domains.

The system model looks at stream processing at different levels of abstraction. At the interface level, application builders expose high-level interfaces to define applications on data streams. They translate these definitions into a processing mesh of slets, which resemble operators, and channels, which resemble buffers. Channels are made explicit in the data processing model. They are views of the upstream processing mesh and support access to streaming data as well as access to data at rest. At the level of the implementation model, the processing mesh of slets and channels is implemented as a service- and component-based architecture. The architecture manages slets and channels individually and supports different implementation languages as well as different processing models. Connectors add a level of indirection between slets and channels. They support implicit buffers (e.g., windows) and capture distributed operation in the model.

The properties of the model and the architecture support and facilitate interoperability and integration (different implementation languages, different processing models, well-defined interfaces), extensibility (extensible interfaces, dynamic processing mesh, possibility to provide custom connectors), and deployment (individu-
ally managed components, management of configuration, state, and wiring, high-level management module, common and distributed runtime platform).
Chapter 3

Implementation

We have implemented the concepts presented in the preceding chapter in our XTream platform. This chapter elaborates on XTream, the technology used, and specific implementation details. It will also concretize certain aspects that the model leaves open (e.g., how a component configuration mechanism can be implemented) by exemplarily showing how we implemented it in XTream. It also elaborates on the implementation of a buffering output connector and thereby exemplifies how to provide additional functionality as reusable components that extend that basic interfaces. Furthermore, this chapter discusses how existing stream processing engines can be ported to the platform.

3.1 XTream Platform

XTream is an implementation of a distributed stream processing platform conforming to the exoengine architecture and system model specified in the preceding chapter. The platform implementation is modular and dynamic itself. It is based on OSGi and utilizes R-OSGi for distributed operation.
3.1.1 OSGi

OSGi [35] is a widely used framework for module management and service composition for Java. OSGi is a complete framework to build modular, service-oriented applications. However, OSGi is often used as the solid foundation for comprehensive software frameworks and platforms. Examples include the well-known Eclipse IDE, the Virgo application server (formerly SpringSource dm Server), the server architecture of Oracle CEP [34], or automotive infotainment platforms like, e.g., in BMW cars. We also use OSGi as a foundation for our platform.

Modules in OSGi are called bundles and explicitly state code dependencies on other bundles. Bundles can be installed, uninstalled, updated, started, and stopped at runtime. The OSGi framework handles the dependencies that arise in the process. Physically, a bundle is a Java archive (JAR) file that contains class files and other resources. The manifest of a bundle JAR contains specific entries (e.g., code dependencies) that make the JAR file an OSGi bundle. If the manifest contains a Bundle-Activator entry, its value contains the fully qualified name of a class that implements org.osgi.framework.BundleActivator. This class will be instantiated and start(..) and stop(..) methods invoked on this object when the bundle is started or stopped.

Services are implemented as ordinary Java classes. Instances of classes (objects) are registered with the OSGi framework’s service registry under one or more interfaces. A service registration can further be augmented by a set of key/value properties. Service clients can look services up in the registry, including filters on properties. When fetching a service they receive a direct Java reference to the object registered as the service.

In addition to the framework, OSGi also specifies a number of OSGi services. These services provide common functionality in a well-defined way and implementations of most of these service specifications are freely available.
3.1.2 R-OSGi

OSGi provides loose coupling of components and dynamic service composition within a Java VM. The open source project R-OSGi [39] extends OSGi to support dynamic service composition across multiple Java VMs. R-OSGi is implemented as a standard OSGi bundle and can thus be added to any OSGi framework. It employs dynamically generated proxies and supports different communication protocols.

Services can be shared using R-OSGi by adding a specific property to the service’s service properties. Alternatively, they can be registered explicitly with the local R-OSGi bundle, which allows for R-OSGi-agnostic services to be shared by a third party bundle. When a shared service is accessed by a remote OSGi framework, the local R-OSGi bundle sends the interface description to the R-OSGi bundle on the remote framework. There, a proxy bundle is generated on the fly and installed in the local framework. This proxy bundle registers a proxy service that has the same interface that was used when registering the original service. Client bundles on the remote OSGi framework can access the proxy service like any other service registered with their OSGi framework. Calls to the methods of the proxy service are converted to remote procedure calls to the original service and processed by the R-OSGi bundles on the remote and local OSGi frameworks.

We have extended R-OSGi to support very resource-constrained devices like TinyOS [26] sensor nodes to tap these devices as OSGi services [40]. These devices are not able to run a full Java VM and OSGi framework. However, they “speak” the R-OSGi protocol and, utilizing the R-OSGi bundle, can thus be used in OSGi applications as a standard OSGi service. This work is complemented by enabling R-OSGi to use transport protocols supported by these devices like, e.g., our implementation of a reliable transport based on TinyOS’ active messages communication layer. In the context of this work we also added transparent support for Java streams to R-OSGi. Many resource-constrained devices deliver data in a stream-
ing manner (e.g., sensor readings). Using Java streams directly is a better fit for these applications than remote method invocations. While outside of the strict scope of this thesis, this work enables the straightforward use of small and resource-constrained devices as data sources and sinks in our OSGi-based platform implementation.

3.1.3 Platform Implementation

XTream is implemented as multiple OSGi bundles and uses R-OSGi as the communication fabric to interact with remote platforms. Bundles are grouped into sets of bundles in our implementation. These sets, however, are only of organizational nature and have no technical meaning with respect to functionality or deployment. All bundles that comprise XTream are standard OSGi bundles. Like the actual streaming applications built on the platform, XTream is modular and dynamic itself and not all of its sets of bundles are permanently required for normal operation. They can be loaded and unloaded at runtime. Figure 3.1 illustrates the logical layering of the implementation bundles of XTream on OSGi. Bundles shaded gray are not mandatory during operation and can be loaded and unloaded at runtime.
3.1. XTREAM PLATFORM

XT-base

The set XT-base contains those four bundles that are permanently required during normal operation. They form the base of the platform implementation.

xtream-core contains Java interfaces defining the basic service interfaces for data exchange and management of slets, channels, and connectors to avoid circular dependencies between the other three bundles in the base set. It also contains constants defining property names and values, which are heavily used in the implementation. These include both generic and component-specific property names and values. For example, the name of the key in the service properties of a connector that specifies to which channel it is connected to is defined as string xtream.connector.channel by constant ch.ethz.inf.xtream.connector.ConnectorConstants.PROP_CONNECTOR_CHANNEL. Likewise, the value that represents the special null component id (no component) is defined as string null_cid by ch.ethz.inf.xtream.core.XTreamConstants.VALUE_NULL_CID.

xtream-core also contains the abstract class ch.ethz.inf.xtream.core.AbstractComponent, which contains methods and fields common to slets, channels, and connectors; the item container ch.ethz.inf.xtream.core.XTreamItem; exceptions; and a utility class that contains helper methods for commonly used tasks. Every XTream bundle has a (OSGi bundle) dependency to xtream-core.

xtream-slet contains the implementation of the generic parts of every slet and its ports. Primarily, this comprises the interaction with the XTream platform implemented using OSGi mechanisms. Additionally, this bundle also contains the application programming interface (API) for slets, which is discussed in Section 3.1.4. Specific slet logic is implemented against this API and without touching OSGi at all.
xtream-channel contains the implementation of the generic parts of every channel, similar to xtream-slet containing the implementation of the generic parts of every slet and its ports. In addition, this bundle contains the implementation of an empty pass-through channel that forwards requests between its input and output.

xtream-connector contains the implementation of the generic parts of every connector, similar to xtream-slet containing the implementation of the generic parts of every slet and its ports. In addition, this bundle contains the implementation of empty pass-through input and output connectors that forward requests between their inputs and outputs.

XT-administrative

The set XT-administrative contains two bundles that form the management module of the platform. They are not permanently required during normal operation and can be loaded and unloaded at runtime. The XT-administrative bundles rely on XT-base bundles to be present.

xtream-management contains the service interfaces provided by the management module and its implementation. These interfaces, which are in package ch.ethz.inf.xtream.management, provide the following functionality:

- **ChannelAdmin**: create, destroy, and list channels.
- **ComponentManager**: get and set the human readable name of components.
- **ConnectorAdmin**: create and destroy connectors.
- **SletManager**: install, uninstall, and list slet classes (types of slets, see Section 3.1.4 for details); create, destroy, and list instances of a specific slet class; start and stop slet instances; get and set custom configuration information of slet instances.
• **WiringController**: connect and disconnect ports to/from channels’ inputs and outputs, optionally specifying a custom implementation for the connector.

SletManager contains the full implementation of management functionality for slets. In particular, it allows to install slet classes, which are plain JAR files with only one required entry in the manifest (see Section 3.1.4 for details). ChannelAdmin and ConnectorAdmin require that channel and connector implementations are already registered in the system, i.e., they are implemented as active OSGi bundles themselves. In a production system, ChannelAdmin and ConnectorAdmin would be replaced by ChannelManager and ConnectorManager implementations that resemble the existing SletManager implementation. For this research implementation it was sufficient to demonstrate the desired properties in terms of development comfort and ease of use for one kind of component (slets).

**xtream-monitoring** provides a comprehensive monitoring interface to the system and employs the whiteboard pattern [30]. Instead of the traditional listener/observer pattern, where clients register themselves explicitly with the notification distributor or the observed object, clients implement certain interfaces and register them as services with the OSGi framework. When a specific event occurs, the monitoring component will notify all services registered as listener for that event. Since the OSGi framework will automatically unregister all services registered by a bundle when the bundle is uninstalled, the whiteboard pattern not only relieves the monitoring component from managing all listeners but also avoids stale listener registrations. The monitoring component also receives a notification through the OSGi framework when a new listener service has been registered and can thus immediately update it with the current state of the system. With respect to its implementation, the monitoring bundle consists of a number of customized **ServiceTracker** objects that track services implementing the following interfaces, which are in package `ch.ethz.inf.xtream.monitoring`:
• **ChannelListener**: for clients interested in channel events (e.g., a channel has been added).

• **ConnectorListener**: for clients interested in connector events (e.g., an input connector has been added).

• **PortListener**: for clients interested in port events (e.g., an slet has created an input port).

• **SletClassListener**: for clients interested in slet class events (e.g., a new slet class has been installed).

• **SletListener**: for clients interested in slet life cycle events (e.g., an slet has changed state).

• **WiringListener**: for clients interested in wiring events (e.g., a port has been connected to a connector).

• **XTreamComponentListener**: for clients interested in generic life cycle events of components (e.g., a component has been removed).

**XT-remote**

**xtream-remote** contains the generic parts of remote connectors, a concrete implementation of a pass-through remote output connector, and the management functionality. The remote connector extends the local connector provided by the XT-base set. The remote management functionality complements the management functionality provided by xtream-management and consists of two services in package `ch.ethz.inf.xtream.remote`:

• **RemoteXTream**: connect to a remote XTream platform.

• **RemoteXTreamPeer**: get the management services (ChannelAdmin, ComponentManager, ConnectorAdmin, etc.) of the remote peer. For every connected peer, one instance of this service is registered and the property `xtream.remote.peer`. 
3.1. XTREAM PLATFORM

name (defined by ch.ethz.inf.xtream.remote.RemoteXTreamPeer.
PROP_REMOTE_PEERNAME) identifies the peer a particular
service instance corresponds to.

Wiring across platform instances happens transparently. To con-
nect to a channel on a connected remote platform it is sufficient to
instruct the local wiring controller to wire the local slet port and the
remote channel (identified by its unique ID), like wiring two local
components. The remote channel ID can be found by, e.g., getting
the peer’s ChannelAdmin service and then listing all channels.

Additional Bundles Used

In addition to the bundles described above, XTream uses a set of
additional bundles and systems, as depicted in Figure 3.1. At the
bottom a Java VM runs the OSGi framework. The base bundles
utilize Configuration Admin and Log services, which are part of the
OSGi specification. The Configuration Admin service allows to set
the configuration of bundles. The service persists the configuration
and delivers it to the same bundle on a framework restart. XTream
uses the Configuration Admin service to persist individual compo-
nent configuration as well as wiring and other platform state. The
Log service provides a general purpose message logger. XTream uses
the Log service for logging.

The R-OSGi bundle used by XTream’s remoting bundle as well
as the remoting bundle itself are optional. However, once distributed
operation has been established, they cannot be unloaded. In con-
trast, the administrative bundles can always be loaded and unloaded.
Unloading the monitoring bundle results in no events delivered to
potentially still registered listeners and unloading the management
bundle results in the inability to modify the processing mesh cur-
rently executing. Such modifications are normally issued to the man-
agement bundle by applications builders (see Section 2.5.10).
CHAPTER 3. IMPLEMENTATION

3.1.4 Component Implementation

Components for the XTream platform are implemented in a reusable manner, which allows the use of multiple instances of each component without any side effects (e.g., no global state, no singletons). Every instance of a component has a unique identifier in the platform, which is provided automatically by the platform. The unique identifier is preserved across restarts of the platform, when the processing mesh and its components are recreated to their previous state. Every instance of a channel or connector provides one distinct service for its input and one for its output. Every port of an slet also provides a distinct service for data exchange. These services implement the basic interfaces defined by the model and potential extensions thereof (see Section 2.5.7). In addition, every component implements the respective management interface (slet, channel, or connector) and potential extensions to it.

As part of the implementation of an exoengine platform, XTream provides the generic parts of slets, channels, and connectors as library code in the respective bundles outlined in the preceding section. These generic parts contain the necessary and recurring glue code that deals with registering the component’s services with the underlying OSGi framework, creating and destroying slet ports, exchanging configuration and state with the platform, and retrieving the connected components’ input (port) and output (port) service objects to invoke methods on them. The developer of a component concentrates on the component’s actual functionality and writes the component against the API of the glue code library—thus generally not having contact with the details of the underlying, OSGi-based implementation of the platform.

The following sections elaborate on details of implementing, deploying, and using an slet in the XTream platform. APIs and procedures are analogous for connectors and channels.
3.1. XTREAM PLATFORM

Slet API

The following classes and interfaces in package ch.ethz.inf.xtream.slet.api comprise the API for slet implementations:

- **SletException**: the type of the checked exception that can be thrown in some of the methods of the API.

- **SletInputPort**: the interface of an input port. An object of this type will be returned if the slet requests to create an input port.

- **SletInputReader**: the interface for the object that the slet implementation must supply when creating an input port. Methods of this interface will be called when an item is pushed to the port.

- **SletMain**: the interface for the class that will be instantiated for every instance of a specific slet. It defines methods that are called at initialization or state transitions.

- **SletOutputFeeder**: the interface for the object that the slet implementation must supply when creating an output port. Methods of this interface will be called when an downstream request on the store data path arrives at the port.

- **SletOutputPort**: the interface of an output port. An object of this type will be returned if the slet requests to create an output port.

- **SletUtil**: the interface of a utility object that is provided to the slet implementation immediately after instantiation. The slet interacts with the platform through this utility object to create and destroy ports, update its configuration, or log messages.
Implementation and Packaging

Slets can be implemented as any number of Java classes. One class of an slet implementation must implement the interface `SletMain`, which defines methods that are called at initialization or state transitions. This class must have a nullary constructor (no-argument constructor). In Figure 2.6, this class is represented as solid black disk. The colored shapes represent instances of other classes used by a particular slet implementation.

The class that implements `SletMain` and all additional classes needed by the slet implementation are packaged into a Java archive (JAR) file. The JAR’s manifest contains one or more entries specific to XTream. These are processed by the management bundle when the specific type of slet (referred to as slet class) is installed and the JAR is transformed to an actual OSGi bundle. The XTream-specific manifest entries are:

- **SletMain-Class**: required; specifies the fully qualified name to the class implementing `SletMain`.
- **Slet-Name**: optional; provides a human readable name for this type of slet.
- **Slet-Description**: optional; provides a short description for this type of slet.

In addition, the installation mechanism honors Java’s standard **Class-Path** attribute, which is translated into the **BundleClass-Path** attribute processed by the OSGi framework.

Installation and Instantiation

Multiple instances of the same slet class (type of slet) can exist. When an JAR representing an slet class is installed through the

\[\text{\textsuperscript{1}}\text{Though a file is the common scenario used, the management bundle’s slet manager accepts the JAR through any input stream, which, e.g., also allows to stream it through a network connection or through a local pipe.}\]
management bundle’s slet manager, it is transformed to an OSGi bundle on the fly. Manifest attributes as described in the preceding section are processed to identify the Java class that implements SletMain and to supply additional information like a human readable name and a short description. The slet class bundle is then installed and started. Thereby, the generic SletActivator (implemented in bundle xtream-slet), which has been designated as bundle activator in the converted manifest, is instantiated and called. It will register a managed service factory (a concept of OSGi’s Configuration Admin service) for slets of this specific type with the OSGi service registry.

When an instance of the slet is created through the management bundle, a configuration object for that instance is created and the configuration persisted with OSGi’s Configuration Admin service. This results in a callback to the particular instance of the managed service factory for slets. The factory then creates an instance of a generic SletImpl (part of xtream-slet’s implementation in package ch.ethz.inf.xtream.slet.impl) and an instance of the specific SletMain slet implementation (as specified in the slet class’ manifest entry SletMain-Class) using a nullary constructor. As a first step, the instance of SletImpl is supplied as the utility object of type SletUtil to the new instance of the slet’s main class. Then further initialization and start methods are called on the main class, which allows it to do custom, implementation-specific initialization and setup. Eventually, the instance of SletImpl is registered unter the Slet service interface with the OSGi service registry. The registration contains a set of service properties, which include the unique identifier for this slet instance.

The above steps are implemented in XTream’s management bundle and the slet factory and are they hidden behind one straightforward method of the management bundle’s SletManager service interface. Likewise, interaction of a component with the platform is implemented and hidden behind APIs as well. For example, when an slet calls SletUtil’s API method to create an input port, a port object is instantiated, a unique identifier assigned, the object added to the slet’s list of ports, and eventually registered with the OSGi
service registry under the \texttt{InputPort} service interface and with its unique identifier and the slet’s identifier as service properties.

**Configuration Information**

XTream implements custom configuration information for individual component instances as key/value pairs. The key must be a string and the value can be any serializable Java object. The custom configuration information is embedded into the service properties of the service corresponding to the specific component instance. Thereby, our implementation also processes and persists it through the Configuration Admin service. If custom configuration information has changed, \texttt{SletImpl} will extract the configuration from the service properties and pass it to the slet’s main class.

If the supplied mechanism of key/value pairs for custom configuration is not sufficient, components can provide a custom configuration mechanism as an extension to their management interface (see Section 2.5.7).

### 3.1.5 Component Binding and Interaction

Interaction between slets’ ports, connectors, and channels happens through services and unique identifiers.

**Component Identifier**

Every component in XTream can be uniquely identified within its instance of the XTream platform, including every port which can be uniquely identified in the scope of the slet it belongs to. In conjunction with a unique identification of a platform instance, a unique identifier can be created that can serve as a URI for the component. The name of the key of the service property that contains a components unique identifier is \texttt{xtream.cid} and is defined by \texttt{ch.ethz.inf.xtream.core.XTreamConstants.PROP_CID}. The identifier is a conjunction of the platform instance’s unique identifier and
the component identifier. The platform’s unique identifier is its fully qualified domain name followed by a colon and the TCP port used by R-OSGi. The component’s identifier in the scope of the platform instance is created by the component’s respective managed service factory and is typically the unique class name of the components main class followed by a dot character and a sequential number. Platform and component identifier are joined by a slash character. An example of a unique component identifier is:

```
sgn-dullerm-01.ethz.ch:10001/ch.ethz.inf.xtream.
slets.mxquerybox.MXQueryBoxSlet.7
```

Ports depend on the slet they belong to. Their unique identifier is the unique identifier of the slet and the port’s name, joined with either a less-than (input port) or greater-than (output port) character. Ports’ names must be unique for an slet and SletImpl ensures this property in its methods to create ports. The unique identifier of the output port with the name IN-PORT-ACCSNOT1 of the above slet is:

```
sgn-dullerm-01.ethz.ch:10001/ch.ethz.inf.xtream.
slets.mxquerybox.MXQueryBoxSlet.7<IN-PORT-ACCSNOT1
```

**Recording Component Bindings**

Figure 3.2 illustrates the binding of components. It shows one slet with two output ports, two connectors, and one channel; their services; and their respective service properties. Components are bound to each other—i.e., a link is created in the mesh—by assigning the unique identifier of the service of the component with cardinality one to a specific “connected to” property of the service of the component with higher cardinality.

For example, when a port is connected to a connector, the unique identifier of the connector is communicated to the port. The port saves the connector’s unique identifier in its properties, which in turn also get persisted. Using this identifier, the port can directly
fetch the right instance of the CICI (ChannelInputConnectorIn—the In of an input connector) or COCO (ChannelOutputConnectorOut—the Out of an output connector) service when interacting with the connector. The connector can access all ports that are connected to it by fetching all instances of the InputPort or OutputPort service that have the connector’s own unique identifier saved in their properties.

Likewise, connectors are connected to channels by saving the unique identifier of a channel in the properties of a connector. The minimal overhead of communicating the unique identifier to connect two components as well as the indirection of fetching a service by a unique identifier establish loose coupling between components and basically allow components to come and go at any time. It also facilitates the exchange of implementations, as only the unique identifier of the exchanged entity itself has to be transferred to the new implementation. Furthermore, the fact that there is only one distinct location—the ports’ or connectors’ service properties—that specifies a distinct link between two entities benefits consistency.

This approach also provides better performance in contrast to the obvious alternative—also communicating the port’s or connector’s identifier to the connector or channel. First, only the port or
connector service has to update its properties. The property holding the “foreign” identifier is not different from the property holding its own identifier. Thus, OSGi’s service tracker can be used straightforwardly. Second, the filter expression used by the tracker in the connector or channel is always of fixed length, regardless of the number of ports or connectors connected to it—it only consists of one key (connector or channel identifier property \texttt{xt.conn} or \texttt{xt.chan} in Figure 3.2) and one value (the connector’s or channel’s actual identifier). Otherwise, the filter expression used by the tracker would consist of one key (ports or connectors identifier property) and a disjunctive list of actual port or connector identifiers, which linearly grows in length with the number of ports connected to the connector or connectors connected to the channel.

Figure 3.2 gives two examples of expressions for the service trackers as blue speech bubbles; one is for a port tracking a connector and the other one is for a channel tracking any number of connectors connected to it. It is important to note that all components can construct the tracker expression purely from their own service properties—either their own unique identifier (value of \texttt{xt.cid}) or the unique identifier of the component they are connected to (value of \texttt{xt.conn} or \texttt{xt.chan}). This is illustrated by the circled values connected by dashed lines with arrows at both ends. Furthermore, the expressions for channels tracking their connectors and connectors tracking their ports will not change throughout their lifetime as their own unique identifier will never change.

Finally, the fact that the wiring between components is saved in the components’ properties allows to keep the management module free of maintaining wiring and to unload it at runtime.

**Accessing the Target Service**

For interacting with the peer component(s), i.e., invoking a method of its/their service(s), we do not look up and fetch the service(s) for every single interaction. Instead, we use OSGi’s service tracker, which tracks the respective services we want to interact with. It
will lookup and fetch the services once, cache their service objects, and subscribe to OSGi system events concerning these services. Additional processing happens only in the case of service withdrawal or disconnection from and reconnection to a different connector or channel, in which cases the trackers are directly notified by the OSGi framework. Most of the time, however, during which a channel, connector, or port service is present and in use, the wiring does not change and the service objects are cached by the tracker and do not change either. Thus, the service oriented design only adds the cost of one additional dereferencing of one native Java object reference.

Figure 3.3 shows the result of a micro benchmark\(^2\) that illustrates the advantages of using caching and proactively updating the list of matching service objects when implementing a dynamic binding. We consider the time needed to transmit an item (average of 10 runs with 100 000 items transmitted in each run) from one slet through a connector, a channel, and a connector to another slet. Figure 2.6 illustrates this chain of slet – connector – channel – connector – slet. We compare our service tracker based implementation with one in which we removed only one of the four trackers (the one tracking the channel’s In service) and instead used dynamic service lookup in the OSGi registry. When considering only one channel present in the system (x axis), the one lookup involved already causes the time to quadruple. When increasing the number of channels present in the platform (they do not even have to be part of the processing mesh that is being measured), lookup time and thus overhead increases linearly with the number of channels. Achieving constant time regardless of the number of channels, connectors, and ports present in the system is key for scalability and general performance on the system’s data plane.

\(^2\)Times captured using `System.nanoTime()` on dual dual-core Opteron 275 machines with 4 GB RAM using Sun’s 64bit JDK 1.6 on 64bit Linux.
3.1.6 Data Management

XTream is implemented in Java, which primarily influences how data is managed.

Data Type

Every data item processed is represented as an instance of a Java reference type—an object. The object can have references to other objects, as long as every participating class can be serialized, which is required for distributed operation.

Item Container

The class ch.ethz.inf.xtream.core.XTreamItem contains the data item object and the metadata (see Section 2.4.4). The type description in the metadata used in conjunction with inheritance enables, e.g., that an item of a specific subtype can be processed by slets that only know one of its supertypes. Contrariwise, this can also be inhibited by not including the respective supertype.
Local Memory Access to Channel and Connector Buffers

An slet retrieves an array (or another agreed-upon data structure) of references to the XStreamItems contained in the connector’s buffer by invoking the respective method on the stream data path. Every call from a port to a connector and from a connector to a channel contains the identifier of the calling component to allow the called component to return the appropriate buffer over the stream data path. Likewise, access to the store data path results in the channel or smart connector returning the buffered items as an array of references.

Item Sharing Amongst Local Components

To reduce unnecessary copy operations inside XStream, item containers and their content data items are not copied when being exchanged between slets and channels. This happens naturally within one Java VM, as by default only references are passed in Java and the model requires item content to be immutable. When an item is contained in a channel that is also accessed by remote platform, a deep copy of the item will be sent to the remote platform.

In the setting of a platform-as-a-service provider, this potentially expands to the optimization of saving space by sharing items of meshes of different users if they exchange data. In such a case the provider may place both meshes onto the same platform instance, as is illustrated by the meshes of Friend 2 and Friend 3 in the photo application example in Figure 2.3.

3.1.7 Component Life Cycle Management

In our implementation, every type of slet is a separate bundle in OSGi, referred to as slet class bundle (after being automatically converted from a slet class JAR, see Section 3.1.4). Any number of instances of this kind of slet can be created once the respective slet class bundle has been installed. Every instance is managed individually in terms of state and custom configuration. Likewise, channels
and connectors are provided by one or more respective bundles per platform instance that may provide different implementation variants.

OSGi manages the life cycle of bundles, which includes that bundles that were active when the OSGi platform has been shutdown will be restarted when the platform is started again. XTream complements this feature with the managed service factories for components. This enables it to fully and automatically restore the state of an XTream platform as it was at the time when the platform was shutdown. This includes bindings of ports to connectors and connectors to channels (which are recorded as service properties) as well as all individual instances of components and their configuration and state. For example, when the slet bundle for a specific type of slet is restarted by the OSGi framework, the Configuration Admin service will deliver the persisted configuration for the managed service factory of that bundle to the factory. This configuration contains information about all instances of this type of slet and their individual configuration. The factory will process this information and will recreate and configure instances of this type of slet accordingly.

It is thus possible to restore a full XTream platform and its processing mesh without having to install the management bundle and rerun the application builder—an important feature especially for small devices that were configured once by an application builder running on a remote machine and are suspended regularly to conserve energy.

### 3.1.8 Application Setup and Management

Application setup and management on XTream closely follows the implementation model’s properties and procedures for application setup and management (see Section 2.5.10). Application builders register slet and channel implementations with the platform, instruct the platform to create instances thereof and how to wire them, interact with these instances through the management interface, and interact with remote platform instances. The controlling parts of a
stream processing system (query compiler, optimizer, control API) become application builders.

They range from manual to fully automatic solutions. The most simple solution is a graphical interface that allows a user to manually instantiate slets from a library of predefined slets and wire them to channels and thus create an application in the scale of applications implemented, e.g., using Yahoo Pipes [51]. Automatic means are required to implement larger, more sophisticated applications. Such an automatic application builder takes a description of the application (e.g., in a declarative or functional language) and translates it into a processing mesh.

Similar to the implementation of components, developers of application builders do not need to know the details of the underlying service-based implementation of the platform. Instead, the management service provided by XTream’s management bundle is in charge of composition and management of all components in one instance of the platform. It provides methods to, e.g., create a new instance of a component, wire two components, or change the configuration of a component, as outlined in Section 3.1.3. As a response to these methods, the implementation of the management service and other parts of the system create instances of slets, channels, and connectors; assigns unique component identifiers and configuration (e.g., the query that a particular slet is executing or whether a channel should persist data); and register them under the corresponding service interfaces.

It would be possible to allow application builders to directly execute these tasks on the SOA environment. However, the implementation of a central, authoritative management service provides a number of advantages. It provides a high-level interface that abstracts from implementation details and also allows access to a remote platform. The management module acts as synchronization point for all operations modifying the state of the platform, thus ensuring consistency and also providing the possibility to approve or reject operations. Finally, by decoupling the management, clients of the management module (and also the module itself) can be loaded
3.1. XTREAM PLATFORM

and unloaded at runtime, e.g., to save space or be replaced with a different version (e.g., an upgrade) without interrupting the running applications.

Similarly, the monitoring bundle establishes a loose coupling between monitored entities (slets, connectors, and channels) and its clients, thereby allowing not only to add and remove clients at runtime, but also to add and remove the monitoring bundle itself at any time.

3.1.9 Distributed Operation

Application builders access remote platforms through a remoting service provided by the platform. The remoting services utilizes R-OSGi to interact with remote instances of the XTream platform. Similar to the management service, the remoting service abstracts from implementation details and provides high-level methods to connect to known remote platforms, discover channels of interest in the network vicinity (e.g., using multicast discovery), access the management service of remote platforms, migrate components between platform instances, and connect local slets to remote channels and vice versa. Stateful components must implement methods for (de)serializing their state in order to enable their stateful migration (memento pattern). The platform calls these methods and handles the serialized state (or a serialized encoding thereof) between suspending and resuming a component.

Interaction between instances of the platform happens in a peer-to-peer manner and on the level of services. Every instance of the platform maintains one connection to every remote platform instance. Through this connection, all communication takes place, which is illustrated in Figure 2.7. The instances of the R-OSGi bundle running on each platform maintain a network link with those peers they are also logically connected to. For example, there is no link between the upper right and the lower right instances of the platform in Figure 2.7 because there are no remote connectors connecting them directly. XTream’s remote operations bundles (xtream-remote,
see Section 3.1.3) communicate with their peers through R-OSGi. All remote connectors shared between any two platform instances are multiplexed through the single network connection between the two platforms.

Channels contain intermediate and final results and are like a view of their upstream processing mesh (see Section 2.4.6). Hence, they are typically the entity on a remote platform that a local application wants to connect to. It is possible to directly connect to a remote channel (if its unique identifier is known) or list channels from a known, remote platform. However, some applications are not interested in a particular instance of a channel or do not know its identifier or even the address of the remote platform it resides on. Instead, they want to find channels in the network vicinity that match certain criteria. These criteria can include anything that can be saved in the Channel’s service’s properties, in particular its name and all the custom configuration it received.

To achieve this, XTream’s remote operation bundle employs R-OSGi’s discovery listeners. A discovery listener is registered for services of type Channel and with properties matching certain criteria. R-OSGi then discovers these services using discovery techniques of the network transport(s) used. For TCP/IP, it uses the Service Location Protocol (SLP) [25], for Bluetooth, it uses Bluetooth’s service discovery mechanisms.

Distribution inherently introduces new possibilities for failures not present in a local system and inevitably imposes longer invocation times. One can argue that transparent distribution can be realized in a SOA environment despite these immanent differences between local and remote operation. Exploiting the property that services can come and go at any time and implementations of services can be exchanged, it is possible to hide network failures behind service withdrawals and justify longer execution times as different implementation of the service. We employ this model, which was proposed by R-OSGi.

While the withdrawal of remote services can be a solution on the implementation level of XTream, we must also deal with it in
our model and expose it to management components like application builders, so that they can react and, e.g., reconfigure a distributed mesh. When the link breaks and the services used internally in the implementation of a remote connector are withdrawn, by default the remote connector shuts down. This results in a disconnection of the connector from the slet at runtime—a dynamic change to the processing mesh that is supported by the model and the architecture. Smart connectors (see Section 2.5.6) can implement different behavior. The monitoring bundle facilitates and abstracts the monitoring of channels, connectors, slets, and ports in the same way as the management service abstracts from implementation details of managing a platform. The monitoring service notifies distribution-agnostic clients uniformly about changes of both local and remote components, while distribution-aware clients can retrieve additional details from the notifications concerning remote components.

We measured the overhead of remote operation on XTream. Figure 3.4 plots transmission times on a buffered data stream for data items with differently sized payloads over a path that includes a gigabit ethernet link. We show the average transmission time of an item of 10 runs with 100 000 items transmitted in each run. We compare XTream to a direct TCP connection with no ports, connectors,
or channels involved. We also show the time spent solely for serializing data items but not sending them over the network (in fact, we write them to /dev/null). Though at this time scale scheduling and garbage collection effects materialize in errors and fluctuations, we can see that using XTream only adds about 3 microseconds overhead. One third thereof stems from the service calls in the chain, as discussed in Section 3.1.5 and Figure 3.3.

3.2 Declarative Buffer Implementation

We implemented a declarative buffering framework that is integrated into XTream as an output connector. It shows by example how to implement an extension to a component of the system. We also illustrate example use cases that utilize the commonly used buffering functionality on the stream data path (simple but flexible windows with different access methods) provided by the connector.

3.2.1 Windows on the Stream Data Path

This implementation provides a generic type of sliding windows with size and slide attributes, which allow to express different kinds of windows (e.g., a tumbling window has \( size = slide \)). In addition to sliding windows, the implementation also provides semantic windows, whose purging is done explicitly by the slet using the window.

A connector maintains one window (logical buffer) for each slet connected to it. An slet uses its service properties to communicate its window requirements to the connector. Section 3.2.4 elaborates on this mechanism and the integration into XTream. Each connector has one buffer (physical storage) that serves all windows (logical buffers). We will use the terms buffer for physical buffers/storage and window for logical buffers throughout the subsequent sections.
3.2. DECLARATIVE BUFFER IMPLEMENTATION

3.2.2 Access Variations and Interface

Section 2.5.5 introduced the possibility of accessing data on the stream data path by other means than the push(item) method of the basic interface for data exchange. Access to windows, which contain multiple items, as well as pull-driven slet implementations require such extensions. Therefore, the corresponding interfaces are extended. The following operation extends the InputPort interface:

\textbf{pushWindow(items[]):} Push the current window to the port.

The following operations extend the Out interface:

\textbf{pullWindow():} Request the contents of a window; returns a set of 0... items.

\textbf{pullAndClearWindow():} Like pullWindow() but the window is purged after its contents has been returned. This method is used primarily to implement semantic windows, which evict items once a certain condition is satisfied. This condition depends on the actual items in the stream and is thus evaluated by the connected slet, which is the entity that processes items.

We can further analyze the interaction between a window and its client in terms of the type of notification about new data and the type of access to the data. \textit{Push} (corresponding to method push(item)) and \textit{PushWindow} (pushWindow(items[])) are types of notification, because these are the operations that may be invoked once a new item has arrived at the connector through its the stream data path. In terms of actual access to the data, \textit{Push} (push(item)), \textit{PushWindow} (pushWindow(items[])), and \textit{PullWindow} (pullWindow() and pullAndClearWindow()) are types of access, because these operations bring the data to the slet.

It is sufficient for an slet to specify the type of notification as part of its window requirements. Specifying the type of access is not necessary. The notification types \textit{Push} and \textit{PushWindow} imply the respective access types \textit{Push} and \textit{PushWindow}. Pull access to the
window through methods `pullWindow()` and `pullAndClearWindow()` is always possible and independent of the selected notification type.

### 3.2.3 Window Configuration Language

This section defines a simple *window configuration language* (WCL) for expressing generic windows over data streams. It is well suited for many applications on XTream with straightforward buffering requirements. Though being designed for these applications and the XTream platform, it is independent of XTream. Also, though being restricted to non-complex window configuration, it can be extended if necessary.

**Design Considerations**

A language that allows an slet to specify its window requirements for the stream data path should be:

**Declarative:** The language design must allow to express a window configuration in a declarative way, i.e., without making any statements about how a particular window configuration is implemented.

**Complete:** The language must be complete enough to allow the definition of sliding windows with the attributes size and slide, semantic windows, and “null” windows (no window at all). The language must allow the distinction between count-based and time-based sliding windows. It must also allow the definition of the *notification type*, which was introduced in the previous section.

**Implementable:** The language must be designed in such a way that it is easy to represent it in a data format of the implementation language and write a corresponding parser for it. Having a language parser is particularly interesting if the complexity of the language in terms of number of language
elements and instantiations is high. For small and simple languages this property is less important.

**Extensible:** The language should be designed in such a way that it is easy to add new language elements (e.g., new properties of a window) or new instantiations of already existing properties (e.g., a new window type).

**Formal Definition**

Any window defined over the physical buffer is implemented as a general sliding window of a `<windowType>` with the attributes size and slide. They are represented by the language elements `<windowType>`, `<windowSize>`, and `<windowSlide>`. The `<notificationType>` element is required to support slets with different notification and access variants. The definition of the WCL is given in Backus-Naur-Form (BNF) below. Appendix A gives the definition and example instantiations in Backus-Naur-Form (BNF), XML Schema, and JSON Schema.

```
<windowConfig> ::= <notificationType> <windowType> <windowSize> <windowSlide>
<notificationType> ::= 'None' | 'Push' | 'PushWindow'
>windowType> ::= 'None' | 'Count' | 'Time' | 'Semantic'
>windowSize> ::= <Unsigned Integer>
>windowSlide> ::= <Unsigned Integer>
```

where `<Unsigned Integer>` represents any integer $> 0$.

**Language Elements**

This section details the language elements and explains the meaning of the various possible values.
notificationType specifies if and how the slet is notified when a new item arrives at the connector. The connector can:

- **Push**: Push the received item to the slet. The slet can also ignore the actual item being pushed and only utilize it as notification that a new item has arrived.
- **PushWindow**: If a new window was created based on the slide attribute, push all elements in the new window to the slet.
- **None**: The Connector will not inform the slet, which is typically the case for purely pull-based implementations.

windowType specifies the criterion based upon which a window completed and a new window is created:

- **None**: No window is kept. The connector implements the same behavior of the default push-through connector and thus conforms with the requirements for extending the data processing interfaces defined in Section 2.5.8.
- **Count**: The window’s size and slide attributes are defined in terms of number of items. I.e., a count-based window with size $n$ and slide $m$ maintains up to $n$ items at a time and advances every $m$ item arrivals.
- **Time**: The window’s size and slide attributes are defined in terms of time passed in milliseconds. I.e., a time-based window with size $n$ and slide $m$ contains all items that arrived within the last $n$ milliseconds and advances every $m$ milliseconds.
- **Semantic**: The window’s size depends entirely on the slet. An slet may decide about the point in time when a new window should be started (and the old one evicted), typically based on the content of the items. The semantic window will add items as long as the slet does not tell it to do otherwise.
3.2. DECLARATIVE BUFFER IMPLEMENTATION

**windowSize** determines the size of a count- or time-based window. If the window is count-based, it states the maximal number of items that the window will contain at any given time. If the window is time-based, it is interpreted as a time period in milliseconds. The window includes all items that arrived at most \textit{windowSize} milliseconds before the current point in time.

**windowSlide** determines when the window is advanced. If the window is count-based, it is the number of items that must arrive before the window is advanced. If the buffer is time-based, it is a time period in milliseconds that must elapse before the window is advanced. In this implementation, the window slide must not exceed the window size, i.e.,

\[
\text{windowSlide} \leq \text{windowSize}
\]

must hold. The special case \textit{windowSlide} = \textit{windowSize} specifies a tumbling window.

3.2.4 Integration into XTream

The buffering output connector implemented according to the preceding sections’ specifications is integrated into the platform as a connector library. Slets utilize the buffering connector by specifying their window requirements in the properties of their ports. If an slet requires more than one window over the stream data path provided by a specific channel, it can open multiple ports with different window requirements and have them connected to the same channel. The ports will be connected through the same connector and thus share the same physical buffer in the connector implementation like multiple slets connecting through the same connector do.

The requirements are set in the port’s service properties as properties \texttt{xstream.port.buffering.notificationtype}, \texttt{.windowtype}, \texttt{.windowsize}, and \texttt{.windowslide}. These properties correspond to the language elements of the WCL and their values are interpreted...
as outlined in the preceding section. Through the service properties the port declares its windowing requirements.

Like every implementation of a connector in XTream, the buffering connector’s main class extends `ch.ethz.inf.xtream.connector.impl.AbstractConnector`. This abstract class contains functionality common to all connector implementations. It also provides the following three callback methods that inform the connector about changes in the connected ports. The implementation of the buffering connector overwrites these methods to receive notifications about connected ports:

**portConnected(cid, reference):** A new port has been connected to the connector.

**portChanged(cid, reference):** The properties of the connected port have changed.

**portDisconnected(cid, reference):** The port has been disconnected from the connector.

Through these callbacks and the service references provided as argument, which allow to examine the port’s service properties, the buffering connector maintains its logical buffers for connected ports, according to their requirements, and also manages its physical buffer backing all logical buffers. The `cid` contains the unique component identifier of the port. The same identifier is added by XTream to calls from the ports to the connector’s methods for data exchange. They enable the port to correctly correlate requests and windows.

Slets that are unaware of the capabilities of the buffering connector can still connect to a channel through a buffering connector. They do not specify any window requirements for their ports. The implementation of the buffering connector then chooses the default Push notification type. The behavior then conforms to the behavior specified in the basic interfaces (items arriving at the connector are immediately pushed to the port using `push(item)`), see Sections 2.5.5
and 2.5.6) and thus complies with the component processing model for advanced implementations (see Section 2.5.8).

The buffering connector does not interfere with the store data path, i.e., pull() requests are passed on to the connected channel. The implementation of the buffer framework can also be used in a remote connector for distributed operation, to provide windows on the platform instance hosting the slets that access the remote channel. In addition, full caching behavior on the store data path can be added straightforwardly as all the mechanisms and implementation for buffering data and maintaining logical windows are already in place.

### 3.2.5 Application Area

The windowing functionality provided by the buffering connector implementation is common and generic. It can be readily used by slets with minimal implementation effort and the slet developer can focus on the actual functionality of the slet. Whenever no complex, specialized buffers are required, the buffering connector, available as a library for XTream, can be used.

Personal information processing is one area where small, well defined slets are composed into applications on personal data. Their window requirements are easily met by the buffering connector. For example, in the photo application in Figure 2.3, the All channel of Friend 1 keeps photos that are at most one week old for the $\omega$-slet connecting to the mobile phone.

Another example use case is the processing and dissemination of results of long-running scientific experiments, which is illustrated in Figure 3.5. Lab equipment continuously logs the results and also publishes a stream of the results. It is connected to channel Exp. readings, which contains these results. A buffering connector is connected to the channel. Applications with the following functionality can be implemented easily as slets that utilize the connectors windowing facilities:
CHAPTER 3. IMPLEMENTATION

Recent readings. A portable and stateless display shows the ten most recent readings. When a new reading arrives, the display device is notified through the standard `push(item)` method. The device then pulls the last ten readings from the buffer. The display device can also access the current content of the buffer at any time (e.g., for redrawing), independently of the arrival of new readings.

SMS alert. All new readings are pushed into an alarm application that filters these readings and matches them against predefined criteria. If there is a match, text messages (SMS) are sent to the supervisors of the experiment.

Daily digest. A summary application produces a digest of all readings from the last 24 hours and sends an e-mail with the information. The requested time-based window automatically closes every 24 hours and the buffering connector pushes the contents of the window to the application.

Eco-hardcopy. Readings are regularly printed. The printing application triggers the printing of a page when there are enough readings to complete that page. Whenever a new reading arrives (notified by `push(item)`), the printing application fetches the contents of the window and renders it on a page. If a page can be filled, it is printed and the window cleared.

Data browser. A data browser application keeps a local copy of the index of all readings including the most important header information, e.g., the time the reading was recorded. New readings are added to this index upon arrival on the stream data path. When a user selects a reading to be displayed in detail, the details are fetched using the store data path.

These applications have a number of different window requirements and data access methods. However, they can all be satisfied using the simple yet flexible buffering framework implemented in a channel output connector.
3.3. **PORTING STREAM PROCESSING ENGINES**

In addition to implementing applications and their components from scratch, applications for existing SPEs can be reused by porting these SPEs to the platform as library SPEs. We distinguish three levels of granularity for porting existing SPEs: individual operators (“assimilation”), bare stream processing engines (“partial assimilation”), and complete applications (“encapsulation”). Generally, when porting an SPE, operators are wrapped in slets and buffers in channels. The controlling parts of an SPE (query compiler, optimizer, control API) become application builders. However, porting existing SPEs can happen at different degrees of granularity to trade off porting effort against flexibility.

3.3.1 **Assimilation**

If a stream processing engine has been implemented in a sufficiently structured way, our platform allows to wrap operators and buffers so that they become explicitly visible to the platform as slets and channels. This allows reusing operators and buffers without changing the semantics of the underlying engine while opening up all four possibilities illustrated in Figure 2.1. Richer interaction between
operators and buffers (e.g., index-based access to data) is possible by extending the basic interfaces for data exchange. Operations on a set of operators and buffers, e.g., optimizations of the operator graph, can still be conducted by modifying the processing mesh of slets and channels and—if necessary—extending the management interfaces of slets and channels to achieve precise control over the components. Assimilation is illustrated in Figure 3.6a, where individual operators and buffers are wrapped as individual slets and channels.
3.3. **PARTING STREAM PROCESSING ENGINES**

### 3.3.2 Partial Assimilation

It is possible to wrap monolithic engines as an slet and use multiple instances thereof to compose an application, e.g., consisting of multiple queries. Each query is executed by one instance of the engine slet. This is the approach we use in the evaluation (see Section 5.2) to wrap existing XQuery [11] and STREAM [32] engine implementations. Partial assimilation reduces the porting effort, while still providing access to intermediate results and allowing distributed deployment. It opens up all four possibilities illustrated in Figure 2.1 but at coarser granularity (e.g., at the level of individual queries) compared to full assimilation. In Section 5.2.5 we show the effectiveness of turning a centralized application composed of multiple instances of a partially assimilated XQuery engine into a distributed application and with it enabling it to handle higher load. Partial assimilation is illustrated in Figure 3.6b, where the two big, gray slets contain multiple operators and also buffers/internal state. These are not explicitly visible to our platform. However, intermediate results are still visible and accessible through the channels between the two slets.

### 3.3.3 Encapsulation

In encapsulation, we wrap an entire application with all the engines and queries into a single slet. This might be the only option if the application and/or its SPE cannot be ported at a finer granularity, e.g., due to licensing reasons. Encapsulation allows to take already existing applications and make them available as a service, in the form of an slet. While limited in flexibility, this approach still facilitates the runtime management of the application and the combination of its ultimate inputs and outputs with other applications. Encapsulation is illustrated by the large slet in Figure 3.6c, which encapsulates everything.
3.4 Summary

This chapter presented implementation aspects of the model and architecture proposed in Chapter 2. It elaborated primarily on XTream, our implementation of an exoengine platform. XTream implements the model and architecture described in Chapter 2 and thereby also concretizes some aspects that the model does not explicitly specify.

XTream is written in Java and uses OSGi, a component and service framework for Java, and R-OSGi, an extension to OSGi that enables remote operation. XTream’s component model for slets, channels, and connectors complements OSGi’s life cycle management facilities. It uses OSGi’s configuration admin service to manage and persist configuration and platform state. The APIs provided by our platform hide the details of the service-based design and allow developers to concentrate on the actual logic of the components. Interactions with the OSGi platform, like registering services or persisting configuration, are implemented in XTream and happen automatically behind the APIs.

The platform implementation is modular and dynamic itself. It consists of different sets of bundles, out of which some are mandatory for normal operation while others can be loaded and unloaded at runtime. Components are implemented against the APIs provided by the platform and their wiring is recorded in their service properties using unique component identifiers. Interaction between components is fast due to proactively caching matching components using service trackers.

The implementation of a buffering output connector exemplified how a flexible library component is developed that can be reused in different applications. The connector implementation extends the basic interfaces, complies with the component execution model, and provides additional functionality. Slets can use its windowing functionality straightforwardly and with minimal implementation effort. Conceptual work behind the buffering implementation and example use cases complemented this piece of work.
Finally, this chapter also showed the different levels of granularity at which existing stream processing engines can be ported to the platform and discussed implications and trade-offs.
Chapter 4

Personal Data Streams

We have implemented two example applications on personal data streams that went along with the development and refinement of the architecture and model. Using these applications we motivated processing personal information using the stream processing model and identified and discussed the properties that a stream processing platform should provide for this purpose [17]. As use cases, they complement the clean slate use case presented in Section 2.3. This chapter presents the motivation for processing personal information as data streams, describes the two applications, and highlights where various aspects of the model manifest themselves in the applications.

4.1 Origin

To explore processing the functionality that a stream processing platform should provide, we focus on personal data streams, i.e., quasi-real-time data streams relevant to people, and two applications built on those streams. The data streams we consider include, among others, e-mails, information on phone calls, text messages (SMS), and information from the digital home (e.g., status of home
appliances, events in the home or office). We use these streams and the experience gathered from building the applications to show that although one can cast the problem in terms of continuous queries and triggers, in practice querying ends up not being the biggest problem. Instead, we show that acquisition, distributed processing, customization, and the integration with a wide variety of devices are the real challenges. Furthermore, flexible deployment and automatic software configuration, typically outside the realm of database-centric data streaming systems, are mandatory properties.

### 4.2 Integration of VoIP and E-Mail

Personal data streams must be user-centric. The user knows best what information to integrate and what is relevant. Accordingly, our goal is to provide users with the tools to do so by approaching the problem as a user-driven, dynamic data stream integration problem. To illustrate the types of interaction that such an integration implies, we have built a data stream application that combines the VoIP client Skype [43] with IMAP e-mail servers to provide context to a phone call.

Figure 4.1 illustrates the application setup. On the left, data sources like the Skype client and an IMAP mail server are adapted by \( \alpha \)-slets and their output is connected to Call Events and E-Mail channels. There is also a Call Simulator slet, whose output is merged into the Call Events channel, and a Contacts slet that holds mappings between e-mail addresses and phone numbers or VoIP usernames. In the middle, the Phone Call Mail slet contains the applications-specific logic and outputs mail messages matching the currently active calls to the Call Mails channel. This channel is consumed by a Mail Sink \( \omega \)-slet that adapts an e-mail client.

Upon receiving a Skype call, the name of the caller is extracted from the meta-data associated to the call. Then the name is looked up in the contacts database and translated into an e-mail address. This e-mail address is used to query an IMAP server to retrieve the
last ten e-mails from and to that caller. The e-mails are displayed and directly available during the call. Furthermore, new e-mail from and to the caller will be displayed immediately while the call is active. The application is easily extensible and its behavior can be dynamically changed (e.g., display e-mail from other sources, different criteria can be used to select the e-mails, list past Skype calls with all the persons addressed in the e-mails displayed, etc.).

This apparently simple application involves a rather complex set of functionalities: event processing triggered from a push stream (Skype producing an asynchronous data stream announcing calls), data filtering to extract the user name, ability to query stored data to retrieve the e-mail address, support for fetching stored data from the IMAP server (initial retrieval of e-mails when call is established) and streaming data (keeping the list of e-mails up to date while the call is active) along the same path of the processing mesh. In addition, the ability to define workflows with all these operations (similar to a business process) and provide the right abstractions for the interactions is crucial to implement such an application.

It is important to note that even though the Phone Call Mail slet accounts for the logic of the application, the full semantics of this
application does not arise solely from that slet. Taking the setup presented in Figure 4.1, it is feasible to say that the “result” of this application lies in the Call Mails channel, being a view over its upstream processing mesh. When looking at this channel, its exact semantics is not

e-mail messages that are associated with active calls

which would be the mere—and per se useless—semantics of the Phone Call Mail slet. Rather, the meaning of the channel is

e-mail messages in the “INBOX” and “Sent” folders for user “x” on IMAP server “y” that are associated with active calls on the Skype client or on the simulator and that could be resolved to e-mail addresses by the Contact DB slet.

The semantics arises from the combination of both the logic of slets and the wiring of the same through channels into an application. The beauty of this concept is that components can be reused as parts of different logical applications implemented within one XTream system. Not only single components but also existing composed semantics can be reused easily. For example, the application shown in the subsequent section reuses this application’s Call Events channel.

In addition to the composition and reuse of components in a processing mesh, this application demonstrates further properties of the architecture. Access to data on the stream data path, which happens on the Call Events channel, access to data on the store data, which happens on the Contacts channel, and the combination of both, as it happens on the e-mail channel, are integrated into the same processing mesh. When requesting e-mails with the appropriate address(es), the Phone Call Mail pushes selectivity to the IMAP \( \alpha \)-slet through an extended interface used for the store data path (see Section 2.5.7). The \( \alpha \)-slet even pushes selectivity further to the IMAP server if it supports IMAP SEARCH extensions.
4.3 Streams in the Smart Home

The second scenario involves a digital home setting. The scenario is part of the Smart Home project, a cooperation between ETH Zurich and Siemens AG, which has been completed by now. The goal of the Smart Home project is to integrate technologies for security, lifestyle, and communication and make them available in the home. We use the smart home scenario to illustrate the importance of rapid development, integration, and deployment of personalized data stream services across a wide variety of platforms and devices.

Figure 4.2 illustrates the setup of the smart home application. Both the events generated by the smart home and the interactions with the appliances are realized with data streams. For example, users can subscribe to certain events and receive a notification on their mobile phone. Furthermore, control of the smart home can be automated depending on the users’ preferences. We will illustrate such an automation by integrating the call information of the Call Events channel as it is used in the Skype/e-mail application described in the preceding section. The goal is for the user to set up different controls in the house that will react differently depending on who is calling (e.g., the so called mood controls determining light levels, opening or closing of blinds, etc.).

The Smart Home scenarios explore in more detail the problem of event management and the programming and deployment flexibility needed to interact with hardware appliances. The application must be able to not only consume streams, but also to react to incoming data streams by generating additional streams, some of them with commands for devices. The proliferation of devices to control and be controlled opens up interesting opportunities for optimizing the deployment of the different parts of the processing chain. While these aspects are considered to be orthogonal or even irrelevant to pure data stream processing, it is crucial that the underlying stream processing platform supports them in order to be able to build the outlined applications on data streams.
CHAPTER 4. PERSONAL DATA STREAMS

4.4 Summary

Applications on personal data can be implemented in an elegant way using stream processing technology—if the right conditions in terms of deployment and integration support are fulfilled. The applications presented are not particularly challenging in terms of data processing. Instead, they highlight the importance of aspects that go beyond core stream processing issues. The deployment of an application as a composition of individual components (single Phone Call Mail slet vs. its use as part of a processing mesh), the ability to share and reuse parts thereof (Call Events channel), the hybrid support for streaming data and data at rest (IMAP server), and the possibility to access external sources and sinks using a diversity of mechanisms (TCP for IMAP, Microsoft Windows’ Window Messages for Skype, XML RPC for the smart home system) are crucial for the implementation of the applications presented above.
Chapter 5

Evaluation

In this chapter, we use XTream to test the proposed model and architecture for the fitness for the desired purposes. On the one hand, we test XTream’s ability to process and disseminate data over the Internet with an experiment on PlanetLab [37]. This setup resembles users exchanging personal information around the globe, in a mesh of unreliable nodes and network links.

On the other hand, we also test XTream’s ability to host existing stream processing systems with little overhead and verify the expected enhancements in terms of interoperability, extensibility, integration, and deployment. The micro benchmarks presented in Section 3.1.5 indicated that the careful design of XTream imposes low overhead. We evaluate the total overhead imposed by the platform using the Linear Road benchmark [5], a well-established benchmark for stream processing systems, and also demonstrate the functionality gained using this benchmark. Finally, we also elaborate on the implementation effort for using the platform.
5.1 Global Scale Mesh on PlanetLab

PlanetLab [37] is a network testbed consisting of 1137 nodes in 518 locations worldwide, primarily hosted by academic or research institutions. Users can acquire a *slice* of this testbed to run their experiments. Other users’ experiments contending for resources on the same node, unpredictable node behavior, and the “real” Internet providing connectivity between nodes make it a good testbed for widely distributed applications.

These conditions make the PlanetLab testbed a good environment to test the feasibility of our platform for the widely distributed personal data stream processing applications we have in mind, which exchange data in an autonomous, peer-to-peer way. Furthermore, these applications also have to run on machines that run other applications at the same time and that can go down at any time.

Figure 5.1: Detail of the synthetic PlanetLab application
5.1. GLOBAL SCALE MESH ON PLANETLAB

5.1.1 Experiment Setup

We selected 200 nodes from 200 distinct PlanetLab sites and deployed Sun’s JRE 1.6, our XTream platform, and a synthetic application on each node. The application simulates users that consume data streams, process them, and publish some of them to be consumed by other users—an abstract description of the collaborative, personal data processing applications we have in mind. The result is a global data stream processing system implementing a complex and dynamic processing mesh of slets and channels. The experiment exposes our platform to dynamic and unpredictable situations, namely the setup phase of the processing mesh (see below) and inevitable node failures in PlanetLab.

Figure 5.1 illustrates a part of the synthetic application running on each PlanetLab node. When starting up, every node creates an identical local processing mesh, which is fully depicted for nodes 1 and 2. It consists of five chains with a source \( T_x \), a channel \( ch_x \), and a sink \( L_x \). Every source creates an item every five seconds. An item consists of a sequence number, the local hostname, and the local timestamp. It is then pushed through the channel and received by the sink, which logs every item to a log file.
After the local mesh has been set up, every node randomly chooses zero to four other nodes to connect to. If it cannot connect to the node immediately, the connection is retried ten times with a timeout of one minute between retries. For each connected node, it connects the local channel with number $x$ to the output of the remote channel with the same number $x$ with probability 0.8 for channel 0, 0.5 for channel 1, 0.3 for channel 2, 0.2 for channel 3, and 0.1 for channel 4. On the connecting node, remote and local channels are connected by an slet $A_x$ that appends the local hostname and timestamp to items. These slets also drop items already seen to break loops.

We ran the experiment for four hours and then collected the logged data.

### 5.1.2 Average Item Latency

Figure 5.2 shows the average transmission time of an item between two hops over the duration of the experiment, grouped into bins of ten seconds. The number of items transmitted on all nodes in the specific time bin is plotted as well. We observe that the average transmission time stays below 500 ms throughout the whole experi-
5.1. GLOBAL SCALE MESH ON PLANETLAB

Figure 5.4: Average transmission times per chain length

5.1.3 Impact of Chain Length

Figures 5.4 and 5.5 analyze the experiment with respect to the length of a processing chain. Figure 5.4 shows that the average transmission times between hops are not influenced by the chain length. As the experiment setup naturally exhibits a smaller total number of chains with increasing length and the probability for a “bad” node to take part in the chain also increases with length, outliers have a higher impact in longer chains causing the values for longer chains to fluctuate slightly more than those of shorter chains. Figure 5.5 gives the number of items received per channel for individual chain lengths in a ten hour run of the same experiment. Subchains logged
Figure 5.5: Numbers of items received per channel and chain length by upstream nodes (which are not the last node in the chain) are excluded from this figure. The different chain lengths per channel follow from the different probabilities for connecting to a channel.

In the figure we see the aggregated number of items received grouped by the number of hops that the item traversed until it was logged, including sender and receiver. Since every item is logged on every node and multiple nodes can connect to the same channel, items in the channel are replicated to the local sink as well as to all downstream nodes. The high probability of ch0 being connected leads not only to longer chains but also to more items being received in total. On ch0 1211928 items have been created and received by their local sinks and an additional 19122401 copies have been received by sinks on nodes other than the creating node (chain length $> 1$).

5.1.4 Conclusion

Deploying and running XTream on PlanetLab, together with the performance numbers shown, demonstrates that XTream can operate in a highly distributed manner and can support large scale processing and dissemination meshes where nodes connect sponta-
neously to stream sources and publish streams in a continuous manner. The experiment demonstrates that the depth of the processing pipeline is not an issue and can be sustained by XTream. Also, channels can deal with varying amounts of data arriving concurrently and we did not perceive any significant impact on average transmission times between nodes. Finally, XTream proves to be resilient to failures of individual nodes, an advantage of how component interaction is implemented in XTream. This is exactly the behavior needed to implement large scale, collaborative exchange of data streams.

5.2 Linear Road Benchmark

The Linear Road benchmark [5] is a well established benchmark for stream processing systems. It simulates variable tolling based on traffic conditions on a fictitious linear city, consisting of a number of straight, 100 mile long, parallel highways. The input to the system increases in rate during a full, three hour run of the benchmark and consists of car position reports and requests for toll information and balance reports. The output of the system consists of accident alerts, tolls, and balance reports. A system running the Linear Road benchmark must emit an output tuple (e.g., balance report) within at most five seconds of when the last input tuple that causes the output to be generated (e.g., request for balance report) entered the system. The number of concurrent highways (in units of .5 for separate directions of highways) that a system can cope with constitutes its load factor $L$. Due to the coarse granularity of this load factor $L$, we will fix $L$ across comparable experiments and examine average tuple latencies as measure for performance impact.

5.2.1 Experiment Setup

Unless noted differently, the experiments were run on a machine with a single Core i5-750 CPU (quad-core, 2.66 GHz) and 8 GB RAM, running the 64bit version of FreeBSD 8.2 configured to use
the CPU’s TSC register as timecounter. We use OpenJDK 6b22 as Java runtime with maximum heap size set to 5 GB.

The numbers presented refer to the toll alerts output of the Linear Road benchmark. They average 4 repetitions of a full, three hour run with an input load $L = 5.0$. Figure 5.6 illustrates the input tuple rate for $L = 5.0$, which increases over the three hours of the benchmark run.

5.2.2 Porting MXQuery and Linear Road

Applications for the MXQuery system typically consist of multiple instances of the MXQuery engine and MXQuery’s storage implementations. Glue code creates and links them to each other to form the final application. Every instance of the engine executes one specific XQuery. A query is compiled into a query graph consisting of multiple operators that potentially have small, internal, implicit state and/or buffers between each other. The storage instances provide windowing or persistent storage and serve as explicit buffers for (intermediate) results between instances of the engine.
5.2. **LINEAR ROAD BENCHMARK**

The MXQuery engine has been wrapped as a $\pi$-slet and the storage implementations as channels. The rich interface between engine and storage, which, for example, allows for index-based access to data in the storage, remains in use as an extension to the basic interfaces of ports and channels. The code that loads the input file of the Linear Road benchmark and feeds it to the benchmark as well as the code that writes the result files have been wrapped as $\alpha$- and $\omega$-slet, respectively. The code that sets up the Linear Road benchmark by creating instances of all involved components (data loader and writer, storage, MXQuery engine), assigning queries to the instances of the engine, and linking these components to each other has been turned into an application builder. The application builder registers the slets (MXQuery, data loader, data writer) and the channel implementations with XTream. It then instructs the platform to create respective instances thereof and connect them. The queries that each instance of the engine has to execute as well as input and output file names are passed to the slets through XTream’s configuration mechanism. Once the application builder has completed, the implementation of the Linear Road benchmark as illustrated in Figure 5.7 has been set up in XTream and processing starts. With the exception of the daily expenditures query\(^1\), the implementation is the same as the one presented in [11] and consists of 9 instances of the MXQuery engine, which process 9 different queries.

\(^1\)We have removed the daily expenditures query from both the original and the XTream implementation due to its negligible impact and the effort required to deploy the historical data.
5.2.3 Effective Platform Overhead

Every additional layer potentially adds overhead to a system and thus we first measure the overhead incurred by the modular and dynamic design employed by the platform. Figure 5.8 compares the original implementation of the Linear Road benchmark (Without) with the XTream version (With). Both implementations can handle the load well and the overhead added to the average processing time of tuples (147.90 ms vs. 148.92 ms) is statistically not significant, as can be seen from the error bars. Table 5.1 provides additional details about the experiment runs. It counts output tuples grouped by processing time in bins of 1 second for the first 5 seconds. One additional bin counts all tuples with processing time greater than 5 seconds.

We also measured raw throughput of the benchmark when being overloaded. The setup for the overload situation is an input load of $L = 0.5$ and the simulation clock ticking 1000 times faster. This results in input tuples being readily available for processing instead of being artificially delayed for a realistic simulation run. In this situation, the average throughput of the original implementation is 5023.15 tuples/s ($\sigma = 140.07$ tuples/s). The implementation on XTream processes on average 4765.68 tuples/s ($\sigma = 59.5$ tuples/s), which is 5% less than the original implementation.
5.2. LINEAR ROAD BENCHMARK

Table 5.1: Tuple latencies for original and XTream implementation

<table>
<thead>
<tr>
<th>Latency range</th>
<th>Without XTream</th>
<th>With XTream</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 1) s</td>
<td>11 329 044.00</td>
<td>11 333 460.50</td>
</tr>
<tr>
<td>(1, 2) s</td>
<td>52 042.75</td>
<td>48 511.00</td>
</tr>
<tr>
<td>(2, 3) s</td>
<td>14 830.25</td>
<td>14 367.75</td>
</tr>
<tr>
<td>(3, 4) s</td>
<td>1 755.00</td>
<td>1 332.75</td>
</tr>
<tr>
<td>(4, 5] s</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>(5, ∞) s</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

5.2.4 Replacing an Slet at Runtime

The exoengine architecture encapsulates entities like operators and buffers and uses loose coupling between them, which enables dynamic changes to the processing mesh. We demonstrate this feature by replacing the car positions slet in the Linear Road workflow, while the benchmark is running, after 1 hour of the 3-hour-long run of the benchmark. We replace the MXQuery slet with a native implementation. The query executed by the car positions slet filters car position reports from the input and forwards them to the upper part of the workflow shown in Figure 5.7. Our native implementation performs the same functionality directly in Java instead of using the MXQuery engine and can be seamlessly plugged into the processing mesh. Replacing one instance of the MXQuery slet with a native implementation happens almost instantaneously and without adverse effects to the benchmark, as the average latencies With compared to Native in Figure 5.8 show (148.92 ms vs. 149.14 ms). Table 5.2 provides additional details of the corresponding experiment runs.

5.2.5 Scale Out

The encapsulation of entities like operators and buffers behind well defined interfaces abstracts from concrete implementations and thus allows to transparently introduce network communication between
components. We demonstrate this feature by distributing the centralized MXQuery engine across multiple machines and scaling the load factor $L$ of the Linear Road benchmark using data partitioning at the level of highways.

We use a cluster of 16 nodes connected through a switched gigabit ethernet network. Each node is equipped with two Xeon L5520 (quad-core, 2.26 GHz) CPUs and 24 GB RAM. The nodes run the 64bit version of Ubuntu 10.04 and Oracle’s JDK 6u22 with maximum heap size set to 5 GB. The maximum possible load for a single node of this setup is $L = 4.5$. However, applications designed for the exoengine architecture can easily be transformed into distributed systems by transparently introducing remote invocations between components running on different machines. For the Linear Road benchmark we chose the partitioning depicted by the dashed, red line in Figure 5.7. Every node handles the traffic of 4.0 highways (upper part of the figure). The toll balance (lower part of the figure) runs only on one node (the master) and the other nodes (slaves) update the shared toll store through a transparent remote service invocation, implemented by a remote connector. We fixed $L = 4.0$ as this configuration proved to spare enough capacity on the master node for the shared part for all setups up to 16 nodes.

Figure 5.9 shows how we scale the aggregate load (left y-axis) linearly with the number of nodes (x-axis) and the effect on the mean
latency of all tuples (right y-axis). For every node we add we can process another 4.0 highways, resulting in $L = 64.0$ being processed on 16 nodes. Overall tuple latency only increases significantly for the first few nodes being added and then flattens out. The impact of the distributed setup on the latency is twofold. First, updates to the toll store on the master by the slaves are synchronous in the current implementation and thus block local processing on the slaves until the update has completed successfully. The impact of this constant overhead on the total average tuple latency is proportional to the number of slaves ($0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$) and results in the steep increase when adding the first slaves. Second, the load of processing updates to the toll store on the master node increases with every slave added. This results in slightly increased latency on both the master node’s local traffic processing as well as responses to slaves’ update requests to the toll store and thus their local traffic processing as well. The steady, small increase in latencies reflects this effect.
The experiment shows that we can use XTream to scale out an application that was based on a centralized engine. We were able to linearly scale up the load of the Linear Road benchmark implementation on MXQuery with the number of nodes. The communication between nodes happens through XTream’s remoting bundle. Remote connectors appear to the MXQuery engine slet like a connector to a local channel. We have chosen a synchronous and straightforward implementation of the remote connector to capture all impacts of network communication. Depending on application semantics, e.g., asynchronous remote connectors with queues can be used to cut latency incurred by the synchronous network communication.

5.2.6 Heterogeneity

XTream enables the federation of heterogeneous stream processing entities on a common platform. We demonstrate this feature by combining MXQuery and STREAM into one application. Figure 5.10 illustrates the modified setup. We replaced the sink for the toll notifications with an slet that converts flat XML fragments into binary relational tuples ($X2R$). The STREAM engine ($SE$) processes the toll notifications according to its assigned query and emits the results to a sink that writes relational tuples to a file ($RS$).

The two engines used in this setup differ in terms of the query model (XQuery vs. CQL), data model (XML vs. relational tuples), implementation language (Java vs. C++), and processing model (purely pull-driven vs. thread-driven pull-based input and push-
based output). Each engine processes queries in its native format. No query transformation or translation takes place and the strengths of each engine and its specific query dialect are retained. Data is consumed and emitted by each engine in its native format as discrete items. Conversion slets, like slet X2R in Figure 5.10, convert between different data formats. They can be built using existing conversion tools and libraries. Federation of applications written and running on different engines typically happens at few, well-defined interaction points. Therefore, the effort to deploy conversion slets or provide custom conversion slets for proprietary data formats is manageable.

We measure the overhead introduced by adding the STREAM engine to the MXQuery based benchmark by running this modified setup of the benchmark and simply passing tuples through the STREAM engine using `select * from S` as query, where S corresponds to the TollNotR input channel. Connecting STREAM to the processing mesh and passing tuples through it adds only 0.5% overhead (148.92 ms vs. 149.65 ms) to the average latencies (compare `With` and `STREAM` in Figure 5.8) and the benchmark runs well within limits, as can be seen in Table 5.3.

The seamless integration of the pull-driven MXQuery engine written in Java with the pull/push-driven STREAM engine written in C++ demonstrates the suitability of the architecture for supporting different processing models and the composition of heterogeneous engines into new applications.

### 5.2.7 Implementation Effort

Since it is not possible to provide universally valid, hard numbers on the effort that is needed to implement certain functionality in software, we provide at least an intuition of the overhead and savings when implementing using the exoengine architecture.

The native car positions slet consists of two classes. One class implements `SletMain` (see Section 3.1.4 for details) and the other class implements the actual filtering functionality. The main class
Table 5.3: Tuple latencies for the federation with STREAM

<table>
<thead>
<tr>
<th>Latency range</th>
<th>With XTream</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 1) s</td>
<td>11 333 460.50</td>
<td>11 324 352.00</td>
</tr>
<tr>
<td>[1, 2) s</td>
<td>48 511.00</td>
<td>53 378.75</td>
</tr>
<tr>
<td>[2, 3) s</td>
<td>14 367.75</td>
<td>17 216.75</td>
</tr>
<tr>
<td>[3, 4) s</td>
<td>1 332.75</td>
<td>2 724.50</td>
</tr>
<tr>
<td>[4, 5] s</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>(5, ∞) s</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

consists of 55 lines of source code, out of which all but 9 lines have been generated automatically from the SletMain interface. Packaging of an slet implementation for XTream only requires to add one attribute to the manifest of the JAR file.

SletMain of the MXQuery slet consists of 300 lines of code. It uses the original MXQuery codebase and a set of interfaces and small helper classes that implement adapters between MXQuery’s and XTream’s interfaces for data exchange. These helper classes are again reused by the data loader and writer slets, the channels, and the native car positions slet.

The overhead of implementing an slet is moderate and limited to implementing the basic interfaces for management and data exchange. Implementing buffers as channels follows the same pattern and is equally simple. For the distributed experiment, we had to change certain data types of the MXQuery engine to implement the Serializable interface so that we could ship instances to other machines.

Similar to MXQuery, we have also ported the STREAM [32] engine to XTream using partial assimilation. The SletMain class of the STREAM slet consists of 70 lines of source code (only 23 actual implementation) and is accompanied by two Java classes interfacing with input and output ports (75 and 53 lines of code). The Java classes of the slet interface with STREAM, which is written in C++,
5.3 Summary

This chapter presented the evaluation of the proposed system model and architecture using XTream, our implementation of an exoengine platform. Evaluations on PlanetLab as well as on local machines and on a cluster validated the properties of the system. It supports processing data in large meshes spanning around the globe and running over unreliable links on unreliable nodes that act completely autonomously.

It also supports existing stream processing systems and their applications with negligible overhead. At the same time, it adds more flexibility in terms of interoperability and integration (federate MXQuery and STREAM engines in the same processing mesh), extensibility (extend basic interfaces with custom interaction between MXQuery engine and storage, replace an slet with a different implementation), deployment (deploy components as well-defined units to the runtime platform, handle query as individual configuration that is managed by the platform, use wiring provided by the platform instead of custom glue code), and functionality (run the distribution-agnostic MXQuery engine on a cluster and increase benchmark performance by an order of magnitude).
Chapter 6

Related Work

The concepts and ideas for a stream processing platform presented in this thesis also touch areas outside of core stream processing. This chapter presents related work in the area of stream processing as well as in these other areas and compares it with the approach presented in this thesis.

6.1 Data Stream Processing

Data stream processing has changed conventional queries into continuously running queries over data streams. Applications like financial market data processing or network intrusion detection require processing large volumes of continuously arriving data with high throughput and low latency. Stream processing turns the processing model of traditional databases (data is stored, queries arrive) upside down to support such applications: data arrives continuously at the stream processing engine (SPE) and triggers the evaluation of queries stored in the SPE. Within the last decade, data stream processing has gone from a fresh research idea (pioneered by Aurora [2], STREAM [32], and TelegraphCQ [13]) to a widespread and
intensively researched theme, with a number of commercial products already available [45, 48, 27, 4, 34, 14].

More recently, SPEs have also been extended with distributed operation capabilities. The research in this direction has to a large extent focused on providing SPEs with specific functionalities including load management (e.g., [1, 38]), fault tolerance and high availability (e.g., [1]), integration with sensor networks (e.g., [22, 3]), and performance tuning (e.g., [28]). These systems mostly target fixed cluster-based settings, where the dynamic wide-area architectural requirements that we consider such as loose coupling and heterogeneity of components, as well as flexibility of deployment and reconfiguration were not considered as equally critical.

XTream is like data stream management systems (DSMS's) in that it continuously processes data streams using a mesh of operators. XTream is unlike DSMS's in that: (1) it is an open and extensible system rather than a closed engine; (2) it supports operators written in different languages and processing models; (3) data dissemination is a first class element of the system (channels, connectors); and (4) it is a federated peer-to-peer system.

Also, typical DSMS’s are based on a (logically) centralized engine and mostly predefined operators. In contrast, we have developed a conventional data stream application (Linear Road Benchmark [5]) on top of XTream with performance comparable to that obtained with these systems. This is done by embedding a streaming engine [11] as an slet into an XTream processing mesh, thereby showing that XTream can be used to develop and run conventional data stream applications. Yet, the interface-based interaction that is at the heart of XTream’s design makes processing, storage, and communication distinct entities that can be independently extended, modified, replaced, and deployed in a distributed setting, unlike in existing DSMS’s.

Closer to our work, the XFlow Internet-scale distributed stream processing system proposes a loosely-coupled architecture for query deployment and optimization. XFlow focuses on an extensible cost model and can express and evaluate a variety of global optimization
metrics [36]. In contrast to XTream, XFlow does not provide any abstract programming models or techniques for building, hosting, or porting various SPE components.

6.2 Data Stream Processing Platforms

Like our work, there are also a few platforms proposed for facilitating the development of stream-based applications, such as System S, Auto-Pipe, MaxStream, or PIPES.

IBM’s System S [28] (now InfoSphere Streams [27]) includes a distributed runtime platform that facilitates the deployment and runtime of dynamic stream processing. The platform pursues similar goals in terms of deployment like the exoengine platform we propose. However, it is less extensible due to its focus on the ecosystem of System S, which includes a language and run-time framework (SPADE) and a semantic solver (MARIO). In contrast, XTream is an independent, pure middleware platform approach and usable for many different SPEs.

Similarly, Microsoft’s StreamInsight platform [4] (which is based on CEDR [6]) is a platform for developing and deploying complex event processing applications. It has the explicit notion of published streams for publishing results of one query to be consumed by another query. This notion resembles channels being explicit connection points to access intermediate or final results.

Auto-Pipe [12] is a development environment for streaming applications executing on diverse computing platforms consisting of a hybrid of multicore processors, GPUs, FPGAs, etc. The authors propose a coordination language X and a compiler that maps X programs into the native languages of the underlying platforms so that specific parts of applications can be run on the specific part of the underlying hardware platform that will provide the highest performance for them. This work focuses on diverse hardware platforms and a standardized coordination language, whereas we focus on diverse SPEs and support their particular properties.
The MaxStream architecture [9], on the other hand, integrates heterogeneous SPEs and databases behind a common declarative query interface, but without considering the lower-level virtualization and flexible wide-area deployment issues that our exoengine architecture tackles. An important part of the MaxStream project is SECRET, a model for analysis of execution semantics of stream processing systems [10]. This model allows users to analyze the behavior of different stream processing system with respect to, e.g., window semantics. MaxStream and SECRET complement the work on syntactic and technical interoperability of this thesis with their work on semantic interoperability.

PIPES [29] is a flexible and extensible infrastructure that provides fundamental building blocks (including runtime components like a scheduler, memory manager, and query optimizer) to implement a stream processing system for the Continuous Query Language (CQL) and a specific operator algebra. In contrast, the exo-engine approach proposes a generic model and platform to host and execute a variety of different and independent stream processing systems, which are not required to share a common query language, algebra, implementation language, or runtime components but can still share them where appropriate.

6.3 Elastic Stream Processing

The importance of elastic stream processing has also been recognized by related work recently [42]. It focuses on elastically scaling the performance of individual streaming operators on multicore machines. Like MapReduce, a restricted processing model enables automatic parallelization and deployment of stream processing applications on a large number of machines. In contrast, our work provides a more general architecture for distribution and a platform that can serve as basis for elastic stream processing.

StreamCloud [24] is a middleware layer that sits on top of streaming engines and focuses on how to parallelize continuous queries by
splitting them into subqueries and distributing them to nodes running a distributed SPE. The exoengine approach provides a platform that hosts different SPEs and could benefit from StreamCloud by integrating it as application builder.

6.4 Publish/Subscribe

Publish/subscribe systems [19] provide mechanisms and an infrastructure to disseminate and filter data from sources to sinks. They decouple senders from receivers by topics, however, they do not support a programming model for sophisticated in-network data processing, distributed operation in a peer-to-peer manner, access to intermediate results, or in-network storage.

XTream is like publish/subscribe systems in that it also provides mechanisms and an infrastructure to disseminate data from sources to sinks and thus decouple senders from receivers. Decoupling senders from receivers by topics can be modeled using channels. XTream is unlike publish/subscribe systems in that: (1) it supports sophisticated in-network data processing; (2) the dissemination network is peer-to-peer; (3) the staged processing allows subscribers to hook up to different parts of the processing pipeline; and (4) there is support for in-network storage and callback processing.

Publish/subscribe is used to refer to a wide variety of different systems [19]. Common to all of them is the ability to disseminate information using more or less complex routing predicates and strategies. It is trivial to implement a publish/subscribe system using XTream and the model we proposed. Routing and filtering is implemented as slets and the dissemination done through channels.

6.5 Peer to Peer

Peer-to-peer networks [44, 41] address the problem of information storage and sharing without resorting to centralized infrastructures.
However, they lack a data processing model supporting the tailoring, filtering, and processing of the information as it flows from sources to sinks. XTream shares the decentralized architecture of peer-to-peer systems when multiple instances of the platform federate to a distributed application. Its communication substrate forms a mesh of independent network links between peers.

6.6 Summary

The exoengine architecture is inspired by a number of ideas and technologies that revolve around processing and disseminating data. It casts some of these concepts into a pure middleware platform for distributed stream processing. Related work on stream processing platforms and elastic stream processing acknowledge the motivating need for sophisticated deployment and runtime support for stream processing applications. Related work on semantic interoperability or parallelization of continuous queries for elastic stream processing complements our work.
Chapter 7

Conclusion

This thesis presented a model and architecture for a distributed stream processing platform that facilitates interoperability, extensibility, integration, and deployment of stream processing systems and applications. The implementation of the platform and its evaluation validate the desired properties and guide future work.

7.1 Summary

Stream processing has enabled a new class of applications that process continuous streams of input data at a high volume and low latency. To facilitate the use of stream processing and further leverage the technique in its traditional application domains as well as beyond traditional stream processing domains, challenges in terms of interoperability, extensibility, integration, and deployment of stream processing must be met.

This thesis presented a generic model for stream processing and an architecture of a stream processing platform that support this model. The model answers the question *what are the fundamental properties common to all stream processing systems* by specifying
respective entities, properties, and conventions for a generic streamprocessing platform.

Chapter 2 presented the layered system model that considers stream processing at different layers of abstraction. For each layer, the requirements have been identified and a corresponding model has been derived. Processing and disseminating personal information, exemplified through a photo exchange use case, serves as an application domain that could benefit from stream processing but has very different requirements than traditional stream processing applications.

At the interface layer of the system model, application builders expose high-level interfaces that accept the definition of stream processing application. At the data processing model layer, applications are represented as a mesh of slets (operators) and channels (buffers/storage). Channels are views over the upstream processing mesh, support access to streaming data and data at rest, and serve as well-defined entities to access intermediate or final results for, e.g., federation of independent applications. At the implementation model layer, the processing mesh is implemented as a service- and component-based architecture. Individually managed and extensible components, supported in different implementation languages and processing models, as well as connectors, which encapsulate distribution in the model, facilitate deployment and interoperability.

Chapter 3 presented the implementation of the proposed architecture in the XTream platform and concretized implementation aspects not regulated by the model. The platform is implemented on OSGi, a service and module framework for Java, and is modular and dynamic itself. The implementation employs dynamic service binding with proactive caching, manages and persists wiring and configuration states, and provides transparent support for distributed operation using R-OSGi. Developers of components and application builders leverage these properties through concise APIs that hide the implementation details. Chapter 3 also exemplified the implementation of a reusable library component and discussed porting existing SPEs to the platform.
In Chapter 4, applications on personal data streams illustrated the origins of processing personal information as data streams and the requirements going along with this idea. While being no challenge in terms of data processing, these applications demonstrated that the actual challenges for building and running this kind of applications lie in platform support for flexible deployment and interoperability.

The proposed platform was evaluated in Chapter 5. A deployment on PlanetLab confirmed its fitness for processing and disseminating information in a mesh of unreliable nodes and network links, spread across the globe. The implementation of the Linear Road benchmark on the platform validated the low overhead incurred by the architecture and demonstrated the additional functionality gained. Operators were replaced at runtime; different SPEs with different implementation languages, processing, data, and query models were successfully federated; and a centralized, distribution-agnostic SPE was scaled out straightforwardly in a cluster.

Chapter 6 presented related work in the area of stream processing and other areas relevant to the platform. Orthogonal related work on, e.g., semantic interoperability or parallelization of continuous queries, complements the work presented in this thesis. Finally, we also identified similar work on platforms for deploying and running stream processing applications, which reinforces the motivation and relevance of this work.

7.2 Discussion

The work presented in this thesis succeeded in specifying a generic system model and architecture for stream processing. It also confirmed that the stream processing model is not limited to traditional stream processing applications. Instead, we also showed that the ability of stream processing to readily process data as it arrives in combination with explicit support for data buffering and forwarding satisfies the needs of processing personal information at a global
scale. The combination of these very different application areas in one model and system has proven to be sometimes challenging—to say the least—but in the end was a key factor for deriving the puristic and generic model.

The definition of applications on personal data streams in the first place and the breadth of applicability required to restrain the depth at certain aspects. For example, we ported only two existing stream processing engines to the platform, as shown in Section 5. Likewise, we fully implemented the automatic component transformation, management functionality, and completely carved out interfaces, abstract classes, and APIs for slets but not for channels or connectors, which is also mentioned in Section 3.1.3. However, we did not sacrifice generality as we carefully chose and focused on work that covers the design space well and substantiates the claims made. The chosen SPEs were very different in a number of aspects and implementing the slet part of our platform to full detail yielded more insight than implementing slet, channel, and connector parts of the platform to only 40% each. We also did not blend this work with related but orthogonal work like semantic interoperability of SPEs. Instead, we focused on the original goal and left orthogonal work to other projects, as identified in Chapter 6. Their findings can be combined with our findings as potential future work.

There will always be use cases for systems that are tuned to the last bit and operate in a closed environment. While such systems can neither afford nor benefit from an architecture that facilitates extensibility, interoperability, and deployment, we conclude that this thesis has demonstrated the feasibility of such an approach. In the light of feedback we got for this work, related research in other areas on similar approaches of structuring systems, and the recent proliferation of such solutions in highly relevant, industrial strength products like Oracle CEP [34], it is not only desirable but in the medium term inevitable to build systems that are easy to integrate, deploy, and extend. This thesis contributes its insights to this development.
7.3 Ongoing and Future Work

In the context of heterogeneous multi-core systems we explore a tight integration of operating system and managed language runtime systems like the Java VM in the Alpenrhein project. Applications for managed language runtimes are typically represented as intermediate bytecodes, which contain richer type information than binary programs (e.g., in the latter it is not possible to clearly distinguish an integer value from a pointer). In addition, managed language runtimes can profile and inspect the runtime behavior of applications. With this static and dynamic information, we expect to improve application performance through smarter resource allocation (e.g., co-locate data with processing on NUMA systems). With the tighter integration and information exchange between runtime and OS, we expect to improve overall system performance through better global optimization across all applications running on a system (e.g., automatic “partitioning” of the system along NUMA boundaries for multiple standalone applications) and richer and more flexible interactions (e.g., if the OS runs out of memory it can upcall into the runtimes and request garbage collection). A platform like XTream that hosts applications defined as meshes of entities that distinctively encapsulate processing, storage, and communication could directly communicate performance-relevant information about its applications (e.g., which slets are connected in the graph and thus exchange data and should be located closer to each other with respect to access to the shared data) to the managed language runtime and OS. We will use XTream in the Alpenrhein project as an example of an application running on top of a Java VM that directly communicates performance-relevant information instead of having to continuously profile it.

In the context of stream processing, the work presented in this thesis can serve as basis for future research. The generic model for stream processing, which explicitly separates concerns of processing (slets), storage (channels), and communication (connectors) into distinct entities of the model, facilitates research in the related but
orthogonal area of semantic integration of SPEs. When ported to
the platform as slets, channels, and application builders and thereby
fitting the model, different SPEs share a common denominator that
helps in understanding a specific aspect across different engines. For
example, load shedding in individual SPEs can be implemented eas-
ily in the exoengine architecture. The shedding mechanisms are im-
plemented in the components like in the original implementation and
the control mechanism orchestrates them through their management
interfaces. Except for different control interfaces, shedding compo-
nents of different SPEs can now be used uniformly and combined
right away, without struggling with technical challenges of integrat-
ing the different SPEs in the first place. Instead, the researcher can
focus on the actual semantic challenges of combining the load shed-
ding concepts of the involved SPEs and directly start experimenting
with them. The components’ generic parts (e.g., input and output
of channels, input and output ports) can support the experiments
by, e.g., extending them to capture profiling information in a “neu-
tral”, SPE-independent place in the mesh. Similarly, and related to
load shedding, a common platform and model can facilitate research
on quality of service, fault tolerance, or provenance across differ-
ent SPEs. Eventually, such work could result in a general model
for load shedding, QoS, fault tolerance, or provenance, respectively,
that becomes part of the exoengine architecture and system model.
Appendix A

Definitions for WCL

This section presents formal definition for the window configuration language (WCL) presented in Section 3.2.3 in Backus-Naur-Form (BNF), XML Schema, and JSON Schema. For each of the schemas it also illustrates an instantiation using a time-based window with notificationType PushWindow, a size of 30 seconds, and a slide of 5 seconds.

A.1 BNF

A syntactic description of the language in Backup-Naur Form and the example instantiation is given below.

A.1.1 Definition

\[
\begin{align*}
<windowConfig> & ::= <notificationType> <windowType> \\<windowSize> <windowSlide> \\
<notificationType> & ::= 'None' | 'Push' | 'PushWindow' \\
>windowType> & ::= 'None' | 'Count' | 'Time' \| 'Semantic'
\end{align*}
\]
APPENDIX A. DEFINITIONS FOR WCL

<windowSize> ::= <Unsigned Integer>
<windowSlide> ::= <Unsigned Integer>

where <Unsigned Integer> represents any integer > 0.

A.1.2 Instantiation

'PushWindow' 'Time' 30000 5000

A.2 XML Schema (XSD)

Using the XML Schema or XML Schema Document (XSD) [47, 8] to define the WCL, it is easier to express the semantic restrictions of the language elements. For example the <windowSize> and <windowSlide> attributes should always be > 0. Using XSD, it is particularly easy to parse and validate a given window configuration, which facilitates the Implementable property in Section 3.2.3.

A.2.1 Definition

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="windowConfig">
    <xs:simpleType name="notificationType">
      <xs:restriction base="xs:string">
        <xs:pattern value=" None | Push | PushWindow "/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="windowType">
      <xs:restriction base="xs:string">
        <xs:pattern value=" None | Count | Time | Semantic "/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="windowSize">
      ::= <Unsigned Integer>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```
A.3. JSON SCHEMA

```xml
<xs:restriction base="xs:integer">
  <xs:minInclusive value="0"/>
</xs:restriction>
</xs:simpleType>
<xs:simpleType name="windowSlide">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
</xs:schema>
```

A.2.2 Instantiation

```xml
<windowConfig>
  <notificationType>PushWindow</notificationType>
  <windowType>Time</windowType>
  <windowSize>30000</windowSize>
  <windowSlide>5000</windowSlide>
</windowConfig>
```

A.3 JSON Schema

It is also possible to represent the WCL in a Java Simple Object Notation (JSON) Schema [52]. JSON Schema, which has borrowed a lot of its ideas from XML Schema, allows to define a schema for a JSON representation of an object. As in the XML Schema representation of the WCL, the *pattern* keyword is used to define the domain of the language elements `<notificationType>` and `<windowType>` using a regular expression. Using JSON Schema has the same benefit as using XSD: It is easy to parse and process, thus facilitating the `Implementable` property.
A.3.1 Definition

```json
{
   "name": "windowConfig",
   "properties": {
      "notificationType": {
         "type": "string",
         "pattern": "None | Push | PushWindow",
         "required": true
      },
      "windowType": {
         "type": "string",
         "pattern": "None | Count | Time | Semantic",
         "required": true
      },
      "windowSize": {
         "type": "number",
         "minimum": 0,
         "required": true
      },
      "windowSlide": {
         "type": "number",
         "minimum": 0,
         "required": true
      }
   }
}
```
A.3.2 Instantiation

```json
{
    "notificationType": "PushWindow",
    "windowType": "Time",
    "windowSize": 30000,
    "windowSlide": 5000
}
```
Bibliography


[31] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996.


Curriculum Vitae

Education

2006–2011  Doctoral studies
Systems Group, Department of Computer Science
ETH Zurich, Switzerland

2000–2006  Diploma program in computer science
Department of Computer Science
ETH Zurich, Switzerland

1991–1999  High school
Bundesgymnasium Gallusstraße, Bregenz, Austria

Work Experience

2006–2011  Research assistant
Systems Group, Department of Computer Science
ETH Zurich, Switzerland

2010  Research assistant
Summer internship at Oracle Labs
Oracle America, Inc., Menlo Park, CA, U.S.A.

2005  Summer intern
AlmafinJaeger AG, St. Gallen, Switzerland