J Internet Serv Appl (2011) 1: 165–181
DOI 10.1007/s13174-010-0016-5

ORIGINAL PAPER

A lightweight and extensible platform for processing personal information at global scale

Michael Duller · Gustavo Alonso

Received: 9 November 2010 / Accepted: 29 November 2010 / Published online: 11 December 2010

Abstract Advances in digital devices, computing, and networking have led to an ever increasing number of personal information being exchanged across the globe. Typically, this is done through centralized, web-based applications like Flickr, YouTube, Twitter, or Facebook. In this paper we propose an alternative architecture for the dissemination of personal information at a global scale. Our solution runs both within a data center as well as on a pool of personal devices such as mobile phones, desktop and laptop computers, or Internet gateways. Our approach leverages idle resources available in millions of devices, allows for much more flexible applications than the predefined services available on the web, and permits users to exchange personal information in a peer-to-peer manner with the possibility but not the requirement to store the personal data in a data center.

Keywords Personal information · Application platform · Distributed system · XTream

1 Introduction

Pervasive access to the Internet as well as the proliferation of mobile devices for both producing (e.g., cameras, recorders) and consuming (e.g., displays, 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, 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 is typically limited. Some services provide an extension mechanism for custom applications (e.g., Flickr App Garden [10], Facebook API [9]). 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.

M. Duller (✉) · G. Alonso
Department of Computer Science, ETH Zurich,
Universitatstrasse 6, 8092 Zurich, Switzerland
e-mail: michael.duller@inf.ethz.ch

G. Alonso
e-mail: alonso@inf.ethz.ch

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.

Privacy: Not everybody feels comfortable with giving away information to a service operated by a big corporation just to be able to exchange it easily with friends. An increasing number of users are becoming aware of and concerned with the loss of control, usage agreements 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 (e.g., at night) in peoples' homes like Internet gateways (e.g., wireless routers, cable modems, etc.), 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 the data.

Nevertheless, we cannot assume that everybody has 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.

2 The XTream vision

2.1 Background

There is a body of work that deals with different parts of the challenges we identified. However, these initiatives are mostly orthogonal to each other and they do not provide a complete system solution.

Peer-to-peer networks [19, 22] address the problem of information storage and sharing without resorting to centralized infrastructures. Unfortunately, they lack a data processing model supporting the tailoring, filtering, and processing of the information as it flows from sources to sinks. On the other extreme, *publish/subscribe systems* [8] implement routing mechanisms that disseminate information to end users. However, publish/subscribe systems lack a programming model and efficient mechanisms for intermediate storage and processing. *Data stream processing* [1, 2, 4, 6] has changed conventional queries into continuously running queries over data streams, but does so only in (logically) centralized settings as well as only for predefined data and operators. Other techniques such as *mash-ups* [21, 23] or *situational applications* [20] focus only on client side processing and cannot be easily generalized.

The purpose of the work presented in this paper is to develop a complete software stack that combines all these ideas into a coherent system, thereby creating the opportunity to explore how a global data processing and dissemination platform may work in practice.

2.2 A data processing and dissemination platform

Our vision for *XTream* is that of a global scale, collaborative, data processing and dissemination platform. XTream has been designed as a dynamic and highly distributed mesh of data processing stages connected through strongly typed channels. The mesh connects heterogeneous data sources and sinks through standard interfaces and supports in-network data processing and storage. Figure 1 illustrates the processing model consisting of a mesh of processing elements (π*-slets*; *slet* is the short form for *streamlet*) and channels that connect sources (adapted by α*-slets*) and sinks (adapted by ω*-slets*). The mesh is extensible and configurable, with the ultimate goal of providing a declarative programming language as the means to specify how data should be disseminated and processed. Slets operate under standard interfaces and are language and OS independent. The channels constitute the underlying messaging fabric for the entire system. Slets and channels are dynamic elements that can be added and removed at runtime, with the system taking care of the necessary steps to ensure continuous operation. The overall design aims at providing a coherent software stack that does not preclude scalability and extensibility.

An important part of the XTream vision is the generalization of the data stream processing model beyond current applications (e.g., stock tickers, data feeds, sensor data) to support a more general class of *pervasive streaming applications* that encompass a wider range of heterogeneous information sources and almost any form of data exchange [7].

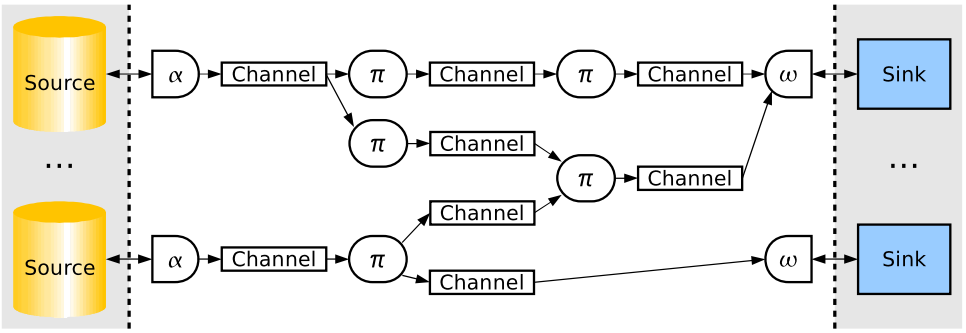


Fig. 1 Data processing model of XTream

2.3 Contributions

XTream uses techniques from publish/subscribe systems and data stream processing. It also borrows ideas from systems such as P2 [13] or commercial platforms like Yahoo Pipes [23]. XTream is heavily inspired by social networks and Web 2.0 ideas. Yet, XTream is unique both as a complete system as well as in terms of the applications it supports and the scope for programmability and extensibility. In addition, XTream contains design aspects that are also important contributions on their own. First, XTream provides a complete system model for both processing and dissemination, without imposing any restrictions on the implementation of the corresponding elements. Second, XTream carefully separates concerns so that data processing, data management, and communication are treated separately. Third, XTream supports distribution as a first class property of the architecture with disconnected operation being a property supported by XTream, not an error condition. Finally, XTream is to our knowledge the first large scale data processing and dissemination platform that is designed as a Service Oriented Architecture. This allows for easy exchange and adding of implementations of individual components, their loose coupling to support dynamic changes, their lifecycle management, and maintaining and updating individual configuration parameters for every instance at runtime.

3 XTream system overview

XTream can be seen from four different perspectives: the end user, communities of users, developers, and service providers. Figure 2 illustrates a simple application built on our platform that processes and exchanges photos. Throughout the paper, we will explain this example step by step and use it to illustrate key aspects of the system.

3.1 XTream: End User

For each single user, XTream is intended as a platform for managing information. XTream turns arbitrary sources of data into data streams and pushes them through the mesh of processing stages to deliver them to the user at, potentially, several end devices. XTream 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 cameras, voice and text messages, reports on calls/SMS/e-mails from several addresses, RSS feeds, e-mail, etc. XTream 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. XTream supports rules to decide which path the information should follow (e.g., depending on time of day, content, or user settings). Finally, XTream also supports the storage and caching of the data, offering functionality for querying the data in different ways.

In the photo application in Fig. 2, 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 *LowRes*, which is fed by a scaling slet *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 turning the user's own setup already into a distributed application.

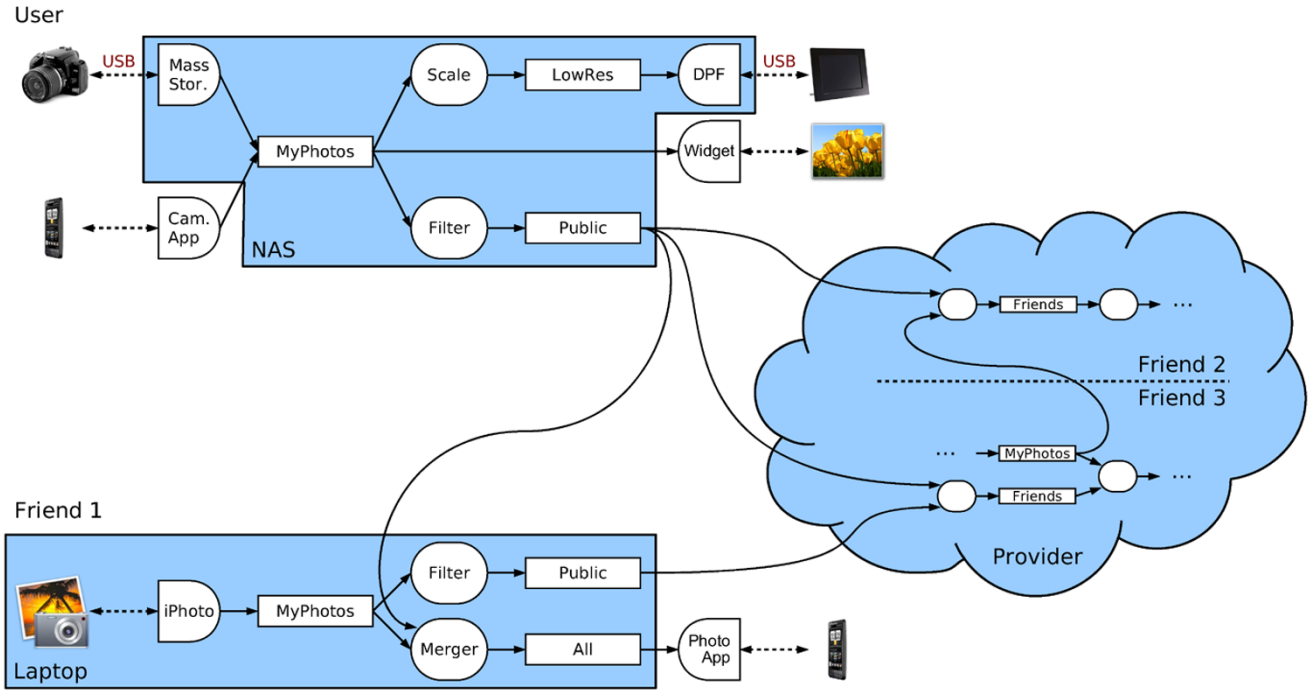


Fig. 2 Photo processing example application

3.2 XTream: communities of users

For communities of users, XTream offers the possibility of linking the *personal data processing and dissemination mesh* of each user with those of other users, thereby building an even larger mesh. XTream 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 Sect. 5.9).

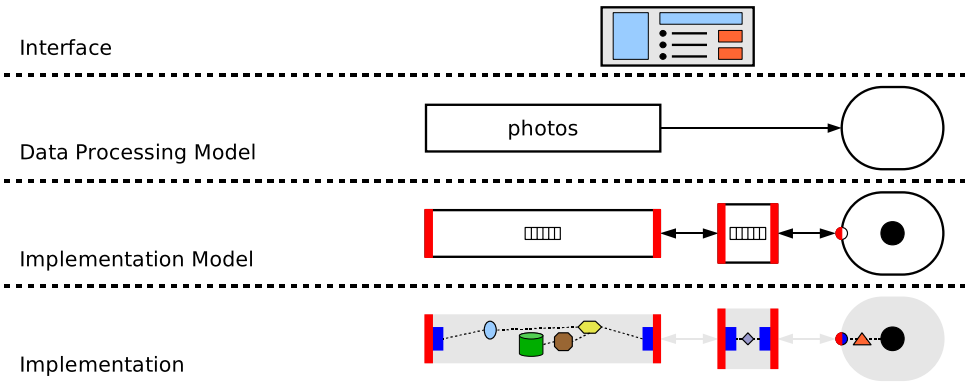
Through standard interfaces in the processing stages and the communication channels between processing stages, XTream gives users the option of publishing any of the final or intermediate results of their personal processing mesh while other users can connect to that information and feed it into their own meshes. The scenarios that we are targeting include forwarding of notifications to other persons depending on conditions (e.g., user sets a *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, XTream 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, XTream 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 Internet. XTream has been designed to function and operate at such Internet scale.

In the photo application in Fig. 2, another user (*Friend 1*, on the lower left) accesses the public photos and merges them with her own photos (*MyPhotos*) into the channel *All*. This channel holds 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 has no connection to her laptop, which is hosting her personal processing mesh.

3.3 XTream: Developers

For developers, XTream 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., splitters, routers, duplicators) processing steps can be built independently of how they are combined by users. Communication between the slets happens through channels. Channels are strongly typed and support all the data management in the system. 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, which treat data as push, pull, or both. Channels also support callbacks to send request back along the reverse path of the processing and dissemination mesh. This

Fig. 3 Layered XTream design



allows slets to request specific data from given sources in either push or pull mode, regardless of whether the source supports push or pull. The architecture of XTream has also been conceived as a modular system where slets and channels can be added or removed at runtime, with XTream dealing with the corresponding dynamic changes. Thus, XTream 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 Fig. 2, examples for slets that can be developed separately and then deployed and integrated into a processing mesh by different people are the ω-slets for the mobile phone, desktop photo widget, and digital photo frame. They are custom with respect to the sink they adapt (phone, widget, frame) but once implemented they can be reused by all users who want to use the same kind of sink and they can be connected to any channel containing photos.

3.4 XTream: service providers

Users of XTream who do not have a device at hand that is connected to the Internet all the time can use the services of XTream platform providers. These companies offer access to an XTream platform 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 XTream 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 of XTream saves resources on the provider side (only one OS instance with the XTream platform running on top is required) and 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 Fig. 2, *Friend 2* and *Friend 3* host their XTream meshes at a provider. The meshes of the two friends frequently exchange data and are thus located on the same physical machine.

3.5 Interface

Channels and slets are the model and mechanism used in the background to implement applications. Typically, they are not visible to the user but hidden behind a convenient interface that provides a declarative specification of the processing mesh using, e.g., a visual language. This final element of XTream is not explored in this paper. The separation of concerns between elements of the system, the clean interfaces between components, and the architecture of the system all contribute to simplifying the task for developers, providers, and the end user for whom using XTream should be no more difficult than using a web browser.

3.6 XTream architecture

The emphasis of XTream is on extensibility and flexibility. Therefore, XTream follows a layered architecture representing an application at different levels of abstraction (Fig. 3). Different optimizations can be performed at each level.

On top, application builders (Sect. 5.8) provide a high-level language and interface to build applications. One level below, applications are represented in the data processing model of XTream (Sect. 4), describing the data format, flow, and access between slets and channels. One level further below, the implementation model (Sect. 5) formalizes the implementation entities of an application, including the services provided by slets, channels, and connectors (a class of components introduced at this level) and their interaction. At the bottom level, the actual implementation of the application is illustrated for our prototype, which is written in Java (Sect. 5.2).

In this paper, we elaborate in detail on the lower three layers. Application builders are discussed only in the context of their interaction with the framework.

4 XTream data processing model

4.1 System elements

The XTream data processing model is based on slets and channels (Fig. 1). Slets interact with channels by using *input* and *output ports* and every port connects to exactly one channel. *α-slets* on the left and *ω-slets* on the right adapt data sources and sinks, respectively. Inside these boundaries, a mesh of *π-slets* is used to process the data with the channels in charge of the dissemination and distribution. The model is made of six elements:

Source: A data source is *any* external device, application, or software component that creates data. External refers to the property that it is not part of XTream's 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 their data. This can range from a shared memory region 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: An α-slet adapts a source by wrapping it into an slet. α-slets have no input ports, they consume data only from the source they adapt. If the source can be divided into well-defined subparts (e.g., folders of an e-mail account), each subpart corresponds to one output port of the α-slet.

ω-slet: An ω-slet adapts a sink by wrapping it into an slet. ω-slets have no output ports, they output data only to the sink they adapt.

π-slet: A π-slet processes data it receives through one or more of its input ports and outputs the result—if any—through one or more of its output ports.

Channel: A channel transports and optionally buffers data between producing and consuming slets. Any number of input and output ports can connect to a channel.

4.2 Data format

The XTream data processing model does not impose any specific data format per se. α-slets packetize data from external data sources into discrete parts—*items*. They optionally have to convert or embed data into a general data type of the implementation language of the system, e.g., a Java object, if the source is implemented in a different language.

Items typically represent a unit of information that corresponds with the stream's content, like, e.g., a single photo, video clip, or e-mail message. Every item is associated with metadata which includes the unique identifier of the slet that created the item, a unique identifier for the item in the scope of the slet that created it, a type description of the item in the implementation language of XTream, one timestamp indicating when the item was created, and an optional item-specific content date.

Items in XTream can either be short-lived, for immediate processing and subsequent discarding, or long-lived, for temporary or persistent buffering of data.

4.3 π-Slets

π-slets can execute arbitrary operations on data accessed through upstream channels, including filtering, joining, and transformation. Slets can be parametrized with configuration information that influences their behavior. We discuss the implementation model of π-slets in Sect. 5.

In the photo application in Fig. 2, the *Scale* slet can be parametrized with the desired result size and compression settings which facilitates its reuse in other applications where different photo sizes are needed.

4.4 Channels

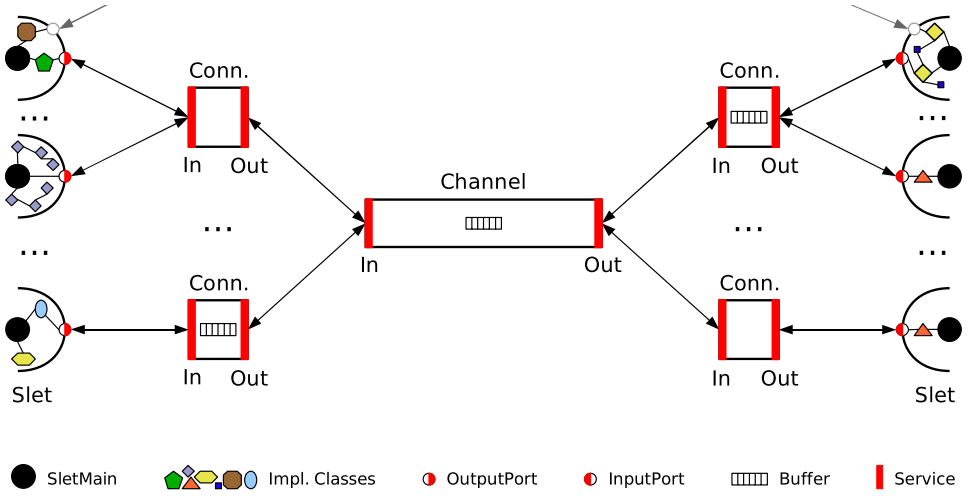
The XTream data processing model distinguishes between two different data access facets: the stream and the query data path. Channels implement both facets and allow for push and pull access to the stream data path and pull or query-like access to the query data path. Slets can access data from channels 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 input port of a downstream slet that connects to a channel specifies its buffer requirements and its access method. The channel 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).

In the photo application in Fig. 2, the *All* channel of *Friend 1* keeps photos that are at most one week old for the ω-slet connecting to the mobile phone.

In contrast to data arriving on the stream data path, which arrives by virtue of the upstream mesh, data delivered on the query data path arrives as response to explicit requests. By default, a pull request on the query data path returns all items in the channel. If the channel is operating in materialized mode, the results are returned from the channel's buffer. Else, the channel fetches all items from the upstream mesh. Optionally, the pull request can be accompanied by a query object and if the upstream slets understand the query, selectivity can be pushed towards the sources. The result of a pull request on the query data path is returned in a buffer that is (logically) separate from the stream data path.

Fig. 4 Implementation model



Conceptually, a channel holds a copy of all data produced by the upstream slets it is connected to. This conceptual perspective resembles views in traditional database systems. Channels can either materialize their contents or request data on demand when being polled by a downstream slet.

In the photo application in Fig. 2, 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 the channel 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. Materialization allows to trade storage requirements off against result latency.

4.5 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 query data path, the request will be forwarded to all connected ports and the results merged. When multiple ports are connected to the output of one channel, the channel replicates data to all logical buffers of the connected ports. Though variations of channels could be implemented that support, e.g., round-robin distribution to multiple downstream slets, this is not done and not allowed. Selective data routing decisions taken in the channel 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 Fig. 2, 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*.

5 Implementation model and prototype

5.1 Overview

XTream is implemented using a Service Oriented Architecture (SOA). Applications developed over XTream are made up of individual components with well-defined interfaces that are composed into a data processing mesh. Figure 4 illustrates the implementation model of XTream by depicting all implementation details from slets emitting items on the left, to a channel, to slets consuming items on the right. Note that, compared to the higher level data processing model, *connectors* have been added as another class of components. Furthermore, the internal implementation of slets, their input and output ports, as well as buffers have been made explicit in the illustration. 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 a channel, respectively. For the sake of readability, only service interfaces used in data exchange have been depicted using thick, red bars. Service interfaces for managing components are not explicitly depicted but the component as a whole represents the respective service.

The following sections elaborate the general implementation aspects of the individual components as well as their interaction. We also present implementation details of our prototype.

5.2 OSGi

Our prototype implementation of XTream is based on the OSGi Service Platform [14]. OSGi is a widely used (e.g., Eclipse IDE, BMW 3 series cars) framework for module

management and service composition for Java. In OSGi, modules 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 takes care of the dynamic dependencies that arise from this process.

Services are implemented as ordinary Java classes, which 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 instance of the class that was registered. Thus, OSGi provides loose coupling and dynamic service composition within a Java VM.

The open source project R-OSGi [18] extends OSGi to support dynamic service composition across multiple Java VMs. When accessing a remote service with R-OSGi, a proxy bundle is generated on the fly and installed in the local framework. This proxy bundle registers a proxy service that is identical to the original service. Calls to the methods of the local proxy service are converted to remote procedure calls to the original service. XTream is implemented on top of OSGi and uses R-OSGi to interact with remote frameworks.

5.3 Component logic

π-, α-, and ω-slets and channels are also entities in the implementation model and reflect their roles in data processing as discussed in the preceding section. In addition, ports and connectors have been added as explicit entities. Every component exposes one or more specific service interfaces for interaction and components are subject to lifecycle management by the platform.

5.3.1 π-Slets

π-Slets have any number of input and output ports. They can create and remove ports during initialization or at runtime. Individual properties can be assigned to every instance of an slet. These properties, a set of key/value pairs, customize the behavior of the particular instance.

A π-slet can normally 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 update to 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 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 update to the *configuration*. The operations are:

Table 1 Slet processing model

Operation	Stream	Store	Config.
Access Stream Data	✓	✓	
Access Store Data	✓	✓	
Update State	✓	✓	✓
Update Configuration	✓	✓	✓
Emit Items	✓		
Return Result		✓	
Modify Ports			✓

Access Stream Data: If the slet's ports are connected to channels that provide buffers/windowing on the stream data path (see Sect. 4), it can access these buffers.

Access Store Data: The slet can pull on any channel connected to its input ports.

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

Update Configuration: The slet can update its 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.

Looking at the first two rows of the above 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 variants to data and thus is the key to the integration of push, pull, and hybrid data sources.

π-slets expose an *Slet* service interface used for managing their properties and state. Additionally, every port of an slet offers an *InputPort* or *OutputPort* service providing methods for pushing data to the port or querying it, respectively. Ports expose their requirements with respect to buffer policy and access method in their service properties.

In our prototype, slets can be implemented using any number of Java classes. One class must implement the interface `SletMain`, which defines methods that are called at initialization or state transitions. In Fig. 4, the solid black disks represent instances of these main classes. The colored shapes represent instances of other classes used by a particular slet implementation.

Multiple instances of the same slet can exist and each is instantiated by creating a new instance of the class implementing `SletMain`. During initialization, an object of

type `SletUtil` is passed to the slet. This interface provides methods for creating and destroying ports, updating configuration and state, and logging. A callback object (implementing `SletInputReader` or `SletOutputFeeder`) has to be supplied for every port that is created by an slet to handle requests (push or access on query path) arriving at the port.

5.3.2 α- and ω-slets

The processing model for α- and ω-slets is very similar to that of π-slets, with some exceptions. Push or pull requests can arrive through the external source or sink they wrap 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 query 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.

In the photo application in Fig. 2, α-slets adapt the user's photo camera and mobile phone. In the example of the camera, the α-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 XTream mesh themselves that interacts with the mesh on the NAS through a remote connection, as discussed in Sect. 5.9.

5.3.3 Channels

Channels forward and buffer data between slets. Thus, their processing model is dictated by the processing model of slets. When a pull request arrives, they either return their contents (if fully materialized) or forward the pull request to all connected upstream slets and return the results to the requesting slet.

Channels provide a *Channel* service interface for management. Every channel also exposes a *ChannelInput* service interface used by input connectors and a *ChannelOuput* service interface used by output connectors. These interfaces provide methods for data exchange between connectors and channels. Furthermore, channels also form the well-defined interface for remote data exchange and every channel has a unique URI.

5.3.4 Connectors

Connectors provide a level of indirection in the interaction between channels and slets. Slets' ports connect to connectors instead of directly to channels. Connectors provide a *Connector* service interface for management. In addition, connectors for channel inputs expose *CICI* and *CICO* service interfaces (ChannelInputConnectorIn and -Out) and connectors for channel outputs *COCI* and *COCO* service interfaces (ChannelOutputConnectorIn and -Out).

In a local setting, connectors only forward requests between channels and ports. Thus, they can be omitted as optimization and channels and ports interact directly with each other. In a distributed setting, connectors provide means for accessing a channel in a remote framework as well as for implementing optimizations like, e.g., local caches, where the connector itself can fulfill the connected slets' buffer requirements. Figure 5 illustrates distributed operation using remote connectors and Sect. 5.9 discusses its details.

5.4 Component interaction

Interaction between slets' ports, connectors, and channels happens through services and unique identifiers. Every component in XTream has a unique identifier in an instance of an XTream platform, including every port which has a unique identifier in the scope of the slet it belongs to. These unique identifiers in conjunction with a unique identification of a platform can thus serve as a URI for the component. Upon being connected, a port or a connector saves the identifier of its peer connector or channel, respectively, in its own service properties.

5.4.1 Accessing the target service

When interacting with the peer component(s), i.e, invoking a method of its/their service, 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 service objects are cached by the tracker and do not change. Thus, the service oriented design only adds the cost of one additional dereferencing of one native Java object reference.

Figure 6 shows the result of a micro benchmark[1] that illustrates the advantages of using caching and proactively updating services when implementing a dynamic binding. We

[1]Times captured using `System.nanoTime()` on dual-socket dual-core Opteron 275 machines with 4 GB RAM using Sun's 64 bit JDK 1.6 on 64 bit Linux.

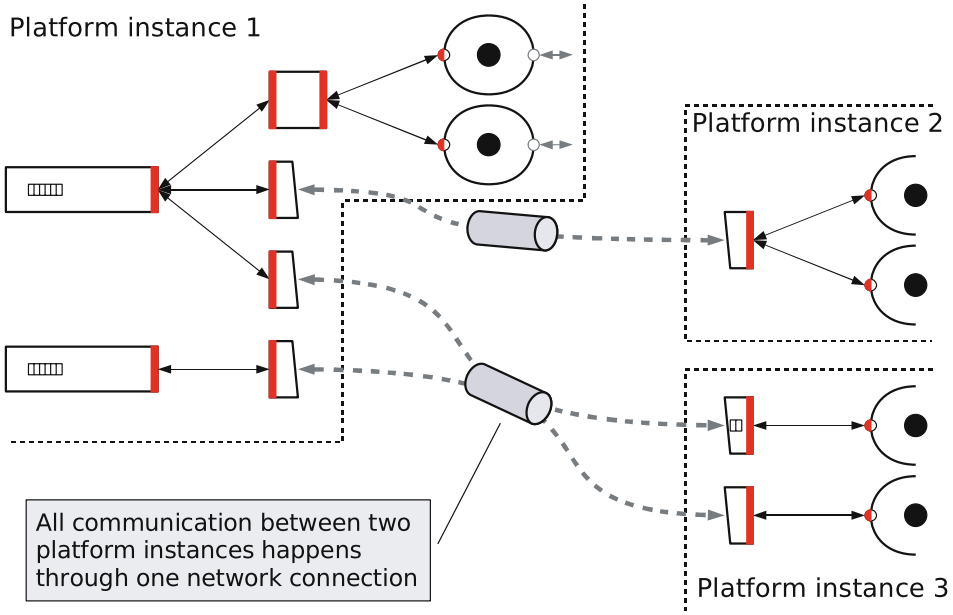


Fig. 5 Distributed operation using connectors

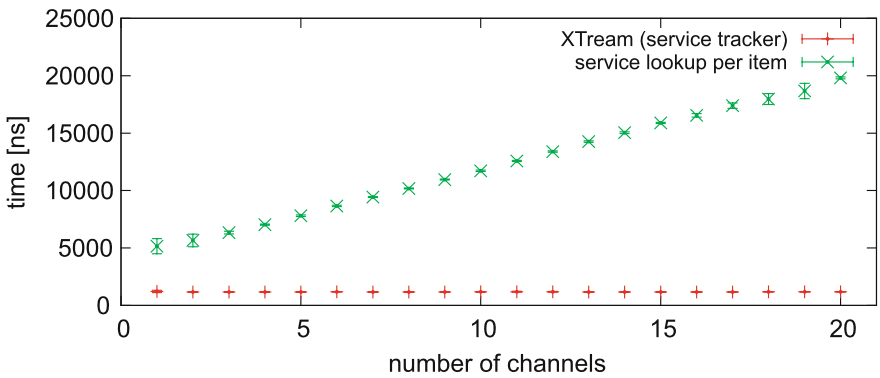


Fig. 6 Data path on local machine

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 4 illustrates this chain. 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 lookup. 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 (not even involved in the chain 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.

5.4.2 *Recording component bindings*

Figure 7 illustrates the binding of components. It shows a set of components, their services, and their respective service properties. When a port is connected to a connector, the unique identifier of the connector is communicated to the port, which in turn saves it in its service properties. 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 and channels are connected 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

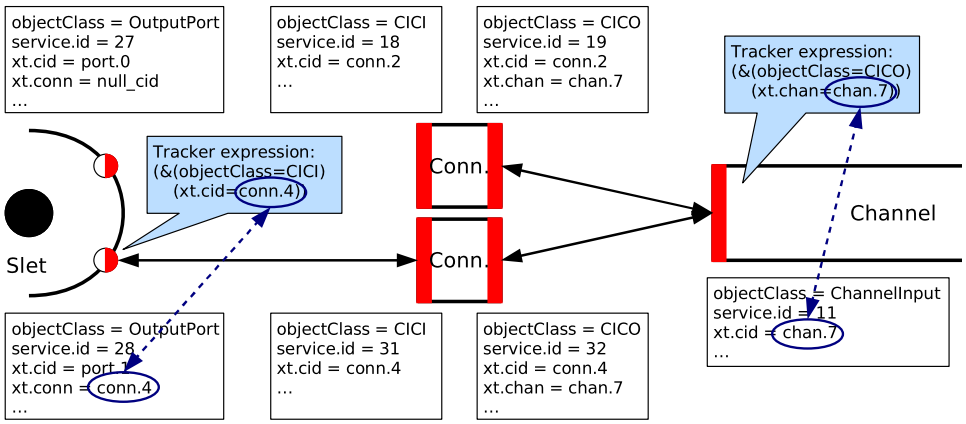


Fig. 7 Component bindings

only one distinct location—the ports' or connectors' service properties—that specifies a distinct interaction 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, a 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 *xt.conn* or *xt.chan*) 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 7 gives two examples of expressions for the service trackers; 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 *xt.cid*) or the unique identifier of the component they are connected to (value of *xt.conn* or *xt.chan*).

5.5 Data management

5.5.1 Data type

Our prototype of XTream is implemented in Java. 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.

5.5.2 Item container

We define a class `XTreamItem` as a container holding metadata (see Sect. 4.2) and the data item object. 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.

In addition, a container also holds the date when it has been created and an optional date that reflects the content's date (e.g., date an e-mail has been sent or a photo has been taken).

5.5.3 Channel buffer sharing amongst slets

To supply an efficient implementation of a buffer that optimally serves one or more requirements in terms of what backing data structure and storage type (RAM, disk) to use, is a separate field of research. To evaluate and illustrate the implementation issues in terms of how these virtual buffers are accessed, we have implemented a memory-backed channel that uses one central buffer to serve different requirements of downstream slets, e.g., *n* most recent items, items newer than *m* minutes, or keep *all* items until consumed by downstream slets.

Every call to the channel or connector through a downstream connector or port is accompanied by the caller's unique identifier, which allows the called connector or channel to identify the correct virtual buffer to return. The caller's identifier is automatically transmitted as the port and connector implementations are provided by XTream and slet implementors only need to use them. The identifier is, like every object in Java, passed by reference, resulting in only four or eight additional bytes in argument size.

5.5.4 Item sharing amongst local components

To reduce unnecessary copy operations inside XTream, 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 XTream's model requires item content to be immutable. When an item is contained in a channel that is also accessed by remote frameworks, a deep copy of the item will be sent to the remote frameworks.

In the setting of a platform-as-a-service provider, this potentially expands to the optimization of sharing items of meshes of independent users if they exchange data and the provider places 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 Fig. 2.

5.6 Lifecycle management

The lifecycle of every slet, connector, and channel is individually managed. The primary 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 framework, or moving a component to a remote framework. The component itself, however, cannot tell the difference and only has to implement the transition from *running* to *suspended* and back. Typically, this only is an issue for α- and ω-slets because they interact with external entities, while most π-slets do not require explicit transition processing.

In our prototype, every type of slet is a separate bundle in OSGi, referred to as *slet class bundle*. 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 properties. Channels and connectors are provided by one or more respective bundles per XTream platform that may provide different implementation variants.

OSGi manages the lifecycle of bundles, which includes that bundles that were active when the OSGi framework has been shutdown will be restarted when the framework is started again. We use this feature in combination with OSGi's configuration admin service, which provides persistent configuration for services, to fully and automatically restore the state of an XTream framework as it was at the time when the framework was shutdown, including bindings of ports to connectors and connectors to channels as well as configuration and state of slets. It is thus possible to restore a full XTream framework 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.

5.7 Monitoring and management

XTream provides a management module in charge of composition and management of all components in one instance of the XTream framework. It provides a service interface with methods to, e.g., create a new instance of a component, connect two components, or change the properties of an slet.

It would be possible to allow application builders (see below) to directly execute these tasks on the SOA environment, but 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 remote access to a framework. The management module acts as synchronization point for all operations modifying the state of the framework, 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 and unloaded at runtime, e.g., to save space or be replaced with a different version.

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. In our prototype, we make heavy use of OSGi service trackers to track slets, connectors, and channels as well as the clients of the monitoring bundle.

5.8 Application builders

The preceding sections have specified the implementation model of slets and which mechanisms are in place to compose slets and channels into applications. The entities that actually program slets and use the provided mechanism to compose actual applications are called *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 [23].

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.

5.9 Distributed operation

In a distributed setting, every instance of the XTream framework has its own management module which is also remotely accessible. Interaction between instances of the

framework happens in a peer-to-peer manner and on the level of services. The management services of the participating frameworks are accessed remotely by an application builder to coordinate the placement of a distributed application or relocate parts of it.

Channels contain intermediate and final results and are like a view of their upstream processing mesh (see Sect. 4). 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, our prototype's remote operation bundle uses 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) [12], for Bluetooth, it uses Bluetooth's service discovery mechanisms.

Regardless of how the remote channel has been identified, connection to it is realized using *remote connectors*, which are illustrated in Fig. 5. One-half of a remote connector is installed in the framework where the channel resides and implements either the *CICO* or *COCI* service interface. The other half of the connector is installed in the framework where the channel needs to be accessed and implements either the *CICI* or the *COCO* service interface. The two parts communicate with each other and act as one logical connector. Typically, the latter part implements a local buffer that satisfies the buffer requirements of all the connected slets. 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.

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

Local memory buffers: Logical buffers of remote channels can be accessed as local memory because they are proxied physically in the local connector part.

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 query data path, and keeping a full and up-to-date copy of a channel's buffer.

Transparency of composition: Remote connectors have the same interfaces as local connectors, thus they can be managed and used like a local connector.

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 Fig. 2, 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.

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 [18].

While the withdrawal of remote channels 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. Thus, the management module provides a monitoring service interface that 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 framework. 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 our prototype. Figure 8 plots transmission times 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

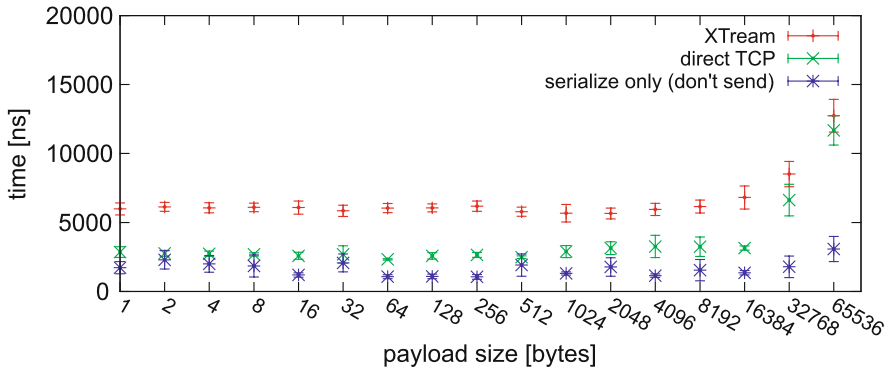


Fig. 8 Data path including a gigabit ethernet link

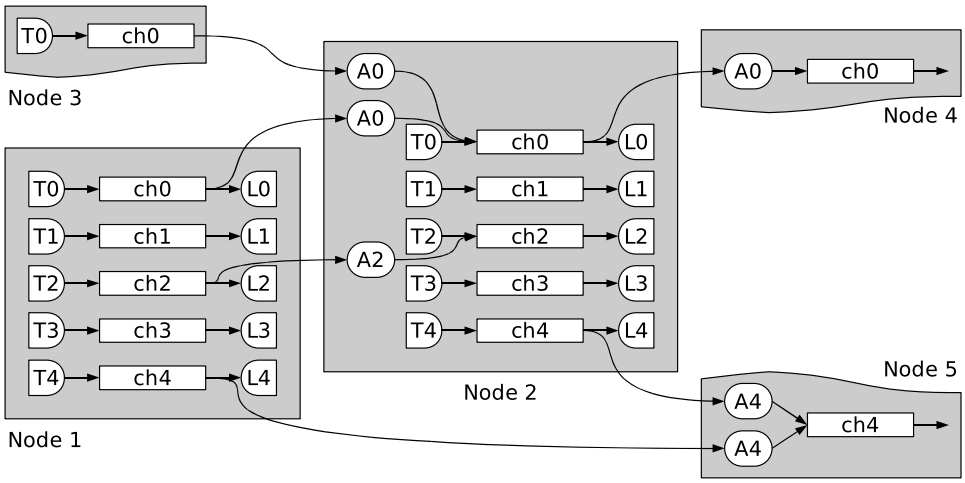


Fig. 9 A part of the synthetic PlanetLab application

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 Sect. 5.4 and Fig. 6.

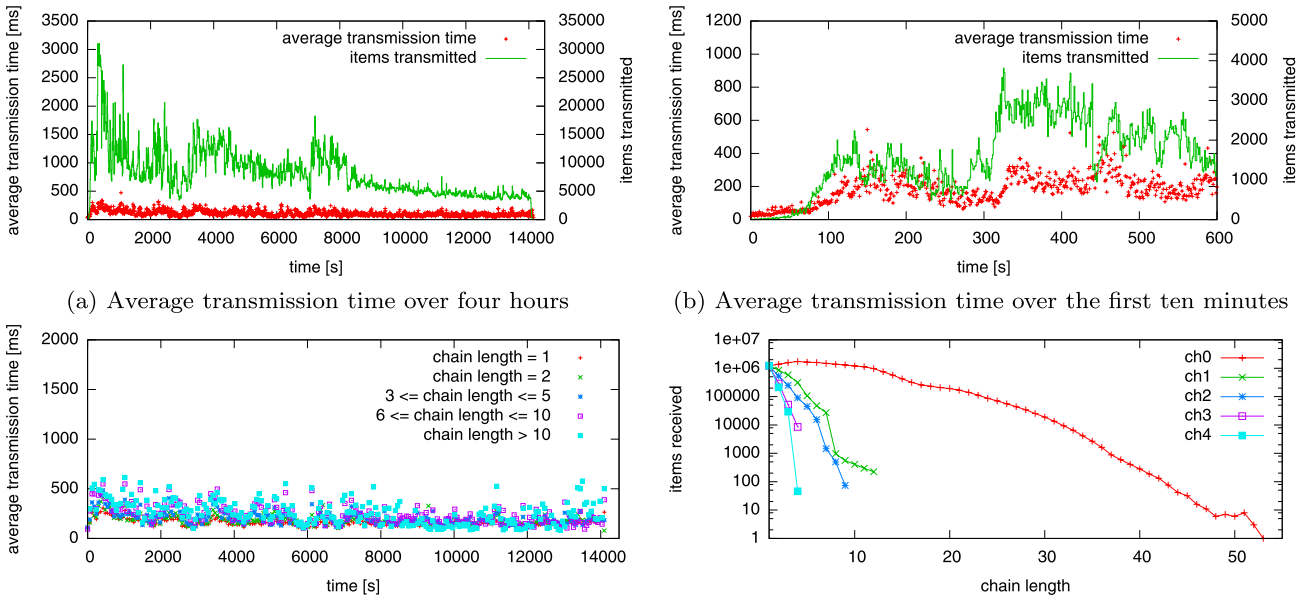
6 Evaluation on PlanetLab

The microbenchmarks presented in the previous section show that the careful design of XTream imposes low overhead and confirms its lightweight nature. In this section, we complement them with an experiment on PlanetLab [16] that shows XTream's ability to process and disseminate data on the Internet. PlanetLab 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.

We selected 200 nodes from 200 distinct PlanetLab sites and deployed Sun's JRE 1.6, our XTream prototype, 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 9 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 push chains with a source Tx, a channel ch$x$, and sink Lx. 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 autonomously chooses zero to four random nodes to connect



(a) Average transmission time over four hours

(b) Average transmission time over the first ten minutes

(c) Average transmission times per chain length

(d) Numbers of items received per channel and chain length

Fig. 10 Analysis of PlanetLab experiment

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 1, 0.3 for 2, 0.2 for 3, and 0.1 for 4. On the connecting node, remote and local channels are connected by an slet Ax that appends local hostname and timestamp to items. These slets also drop items already seen to break loops.

We ran the experiment for 4 hours and then collected the logged data. Figure 10a shows the average transmission time of an item between two hops over the duration of the experiment, grouped into bins of 10 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 experiment. Figure 10b examines the first ten minutes of the experiment in more detail, showing again the average transmission time of an item between two hops but this time grouped into bins of one second. We can see that though the load triples after 5 minutes and after another 3 minutes halves again, average transmission time stays at roughly 200 ms. In both figures, the average transmission time as well as the load clearly vary and exhibit jitter, which reflects the unpredictable nature of the testbed as well as nodes failing.

Figures 10c and 10d analyze the experiment with respect to the length of a processing chain. Figure 10c 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 10d gives the number of items received per channel for individual chain lengths in a 10-hour run of the same experiment. Subchains logged by upstream nodes are excluded from this figure. The different chain lengths per channel follow from the different probabilities for connecting to a channel. In total, around 1.2 million unique items were created per channel, which is illustrated at the value for chain length 1, representing the local sink that logs all data created by the local source.

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 spontaneously 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.

7 Discussion and related work

XTream borrows ideas and techniques from a wide variety of research areas and systems. Its strength is to put all these ideas and techniques into a coherent system.

Data streams 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 standard languages (currently Java); (3) dissemination is a first class element of the system and treated as an orthogonal concern; and (4) it is peer-to-peer. Typical DSMS's (Aurora [1] and Borealis [2], TelegraphCQ [6], and STREAM [4]) are based on a (logically) centralized engine and mostly predefined operators. In contrast, we have developed a conventional data stream application (Linear Road Benchmark [3]) on top of XTream with performance comparable to that obtained with these systems. This is done by embedding a streaming engine [5] as an slet into an XTream processing mesh, thereby showing that XTream can be used to develop 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.

Publish/subscribe XTream is like publish/subscribe systems in that it provides mechanisms and an infrastructure to disseminate data from sources to sinks and thus decouple senders from receivers. 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 [8]. 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 with the routing and filtering implemented as slets and the dissemination done through channels. The advantage of using XTream comes from the SOA architecture, the ability to perform optimizations at all levels (e.g., slet and channel placement, buffer management), and late binding to implementations (XTream uses the most efficient transport, e.g., local invocations, shared memory, TCP connections, depending on the endpoints).

RSS-mashups XTream bears certain similarity to systems like Yahoo Pipes [23] and Damia [21]. Ultimately, we could think of having something similar to these systems for specifying XTream applications at a high level. Unlike these systems, XTream is completely decentralized, does not rely on a central infrastructure, and is extensible with a richer interface for developers.

Global data streaming XTream resembles global data streaming systems such as XPORT [15], HiFi [11], and Hourglass [17]. These systems propose an overlay network that allows the placement of operators for processing data streams. They assume well-connected, dedicated machines. Unlike XTream, they do not provide a complete stack, lack interfaces, are not extensible, and do not have a clean separation of processing, storage, and communication.

8 Conclusion

In this paper we presented XTream, a complete system and software stack for data processing and dissemination at Internet scale. XTream defines an application model for building applications on personal information streams. Slets written by different developers can be added at any time and seamlessly integrated into applications because of common interfaces defined by XTream. The ability to run on users' devices and also as platform as a service hosted at providers is an important property of the system. Applications are not restricted to a specific platform and advanced users can leverage resources of their local devices. It also enables everyday users to define, run, and access their personal information processing applications without bothering about the *wheres*, *hows*, *whens*, and further challenges of deploying and running a distributed application. We demonstrated the feasibility and potential of the idea using microbenchmarks and through a large scale deployment on PlanetLab. Our experimental results demonstrate the advantages of using SOA as a design principle, giving XTream a degree of flexibility and openness lacking in DSMS's, publish/subscribe, and global data streaming systems. Unlike previous work in these areas, XTream provides a platform for development that supports the combination of existing systems: e.g., a publish/subscribe platform extended with data streaming capabilities or a peer-to-peer data stream processing system.

References

1. Abadi DJ et al (2003) Aurora: a new model and architecture for data stream management. In: VLDB
2. Abadi DJ et al (2005) The design of the Borealis stream processing engine. In: CIDR
3. Arasu A et al (2004) Linear road: a stream data management benchmark. In: VLDB

4. Arasu A et al (2004) STREAM: the Stanford data stream management system
5. Botan I et al (2007) Extending XQuery with Window functions. In: VLDB
6. Chandrasekaran S et al (2003) TelegraphCQ: continuous dataflow processing for an uncertain world. In: CIDR
7. Duller M et al (2007) XTream: personal data streams. In: SIGMOD
8. Eugster PT et al (2003) The many faces of publish/subscribe. ACM Comput Surv
9. Facebook API http://developers.facebook.com/
10. Flickr App Garden http://www.flickr.com/services/
11. Franklin MJ et al (2005) Design considerations for high fan-in systems: the hifi approach. In: CIDR
12. Guttman E (1999) Service location protocol: automatic discovery of IP network services. IEEE Internet Comput. 3:71–80
13. Loo BT et al (2005) Implementing declarative overlays. In: SOSP
14. OSGi Service Platform http://www.osgi.org/
15. Papaemmanouil O et al (2006) Extensible optimization in overlay dissemination trees. In: SIGMOD
16. Peterson L et al (2003) A blueprint for introducing disruptive technology into the Internet. SIGCOMM Comput Commun Rev
17. Pietzuch P et al (2006) Network-aware operator placement for stream-processing systems. In: ICDE
18. Rellermeyer JS et al (2007) R-OSGi: distributed applications through software modularization. In: Middleware
19. Rowstron AIT et al (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware
20. Rozlog M (2008) Situational applications. http://www.ddj.com/architect/206102105
21. Simmen DE et al (2008) Damia: data mashups for intranet applications. In: SIGMOD
22. Stoica I et al (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: SIGCOMM
23. Yahoo Pipes: http://pipes.yahoo.com/