Virtualizing Stream Processing

Michael Duller[1], Jan S. Rellermeyer[2], Gustavo Alonso[1], and Nesime Tatbul[1]

[1] Systems Group, Department of Computer Science,
ETH Zurich, Zurich, Switzerland
{michael.duller,alonso,tatbul}@inf.ethz.ch
[2] IBM Austin Research Laboratory, Austin, TX, U.S.A.
rellermeyer@us.ibm.com

Abstract. Stream processing systems have evolved into established solutions as standalone engines but 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 but introduces different requirements on how stream processing solutions can be integrated, deployed, extended, and federated. To address these needs, we present an *exoengine* architecture and the associated *ExoP* platform. Together, they provide the means for encapsulating components of stream processing systems as well as automating the data exchange between components and their distributed deployment. The proposed solution can be used, e.g., to connect heterogeneous streaming engines, replace operators at runtime, and migrate operators across machines with a negligible overhead.

Keywords: stream processing, federation, virtualization.

1 Introduction

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 supports such applications using a model whereby 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 research idea (e.g., Aurora [2], STREAM [20], and TelegraphCQ [9]) to a widespread solution, with several commercial products already available [27, 29, 16, 4].

Stream processing has proven to be useful for many applications. However, its applicability is still limited in terms of interoperability and deployment.

Interoperability refers to the integration of heterogeneous SPEs [28]. 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 book keeping and stock control software need to be coordinated even across large geographic distances. From these scenarios we derive

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 269–288, 2011.
© IFIP International Federation for Information Processing 2011

Petitioner has added a romanette to page (i).
Otherwise, it leaves the original page numbering.

PAN Exhibit 1013

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.

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.

To address these challenges we explore the possibility of *virtualizing* any component of a streaming engine (operators and buffers as well as entire engines) so that they can be automatically deployed, managed, and composed in a flexible and dynamic manner. The aim is to build a generic middleware *platform* that allows to (1) encapsulate existing engines (either for embedding into applications or for composition); (2) support combinations of heterogeneous operators; and (3) provide the functionality needed in a distributed platform (e.g., support for operator migration, replacement, data routing).

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). The architecture proposed is inspired by the concept of the exokernel for operating systems [12, 19]. Like exokernel, it implements as few policies and makes as few assumptions as possible to support a wide range of different SPEs well.

Middleware has already proven to be useful in providing additional features for data processing systems like, e.g., TP monitors or message queueing systems for traditional databases. In fact, several engines have been extended with middleware platforms: IBM's System S [17] or Yahoo's S4 [21]. These systems are built as extensions to one particular SPE. Our approach is a pure middleware system that is engine-independent. In that way, we do not impose engine-specific semantics or a processing model, but cater to dynamic and distributed operation, deployment, and lifecycle management and provide interoperability between heterogeneous SPEs with potentially different semantics. From our own work, we have explored semantic aspects of integrating heterogeneous SPEs in MaxStream [6], and wide-area, stream-based processing of personal information in XTream [11], both through middleware-based solutions.

In this paper, we present the design, use, and implementation of *ExoP*, an architecture for stream processing. ExoP provides well-defined, extensible interfaces for encapsulating stream processing entities (operators, buffers) and building applications on top of these. It also supports dynamic composition, dynamic data routing, and component lifecycle management.

The results presented in the paper validate the potential for the ideas behind ExoP as they cannot be achieved with any other system we are aware of. For instance, we have ported two existing, different SPEs into ExoP. One of them is the MXQuery engine [7]. We use the implementation of the Linear Road benchmark [5] with MXQuery to show that ExoP has a negligible overhead (0.7 %; see Sect. 5.4), and yet, it provides the flexibility to the implementation that we claim: dynamic and distributed deployment and component lifecycle management. We show that we can replace at runtime part

PAN Exhibit 1013

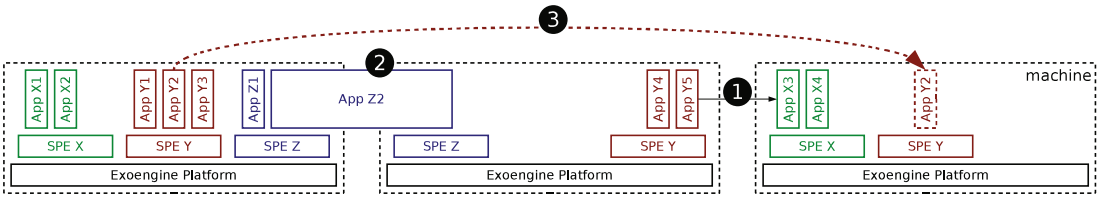

Fig. 1. Exoengine vision

of the system with a native implementation of the operators without interruption in service (see Sect. 5.5) and that we can turn the originally centralized implementation of the Linear Road benchmark into a distributed implementation (see Sect. 5.6) to achieve one order of magnitude improvement in performance; a load factor of 64, which is equal to the fastest published results [30] of a highly optimized, distributed implementation. The other engine ported to ExoP is the Stanford STREAM system [20]. In the paper, we show that we can deploy the engine on our platform, supply queries as part of our configuration mechanism, and federate STREAM and MXQuery (see Sect. 5.8) with a minimal development effort (see Sect. 5.7).

2 Exoengine Architecture

Figure 1 illustrates the vision of the exoengine architecture. It virtualizes stream processing and thus enables ❶ multiple applications written for different engines/query languages to exchange data across machine (platform) boundaries; ❷ single applications to run across multiple machines; and ❸ applications to be migrated to other machines.

2.1 Layers

The exoengine architecture considers stream processing applications at different layers of abstraction (Figure 2). On top, a high-level abstraction, e.g., a streaming query language, presents the *interface* to the system. This interface is provided by *application builders* (see Sect. 4.2). In the example shown, the interface is CQL [20], a language for continuous queries. 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 2.2 provides more details. The data processing model is generic enough to fit different flavors of stream processing (e.g., push vs. pull driven engines) 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 (see Sect. 2.3). The implementation model grasps elements as individually managed components, wires them using loose coupling, enables remote operation through connectors, and thus enables *flexible deployment*.

PAN Exhibit 1013

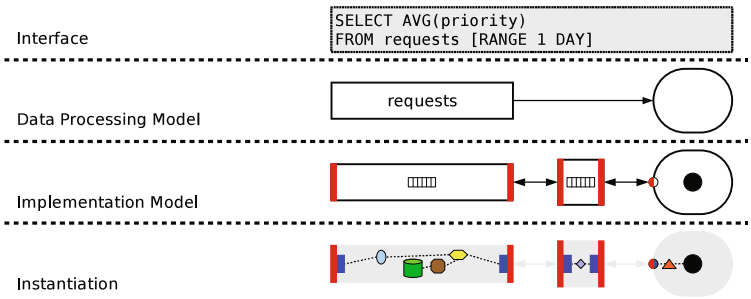

Fig. 2. Layers of abstraction

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 either implemented manually as part of the application implementation or generated by the application (e.g., by a query compiler).

2.2 Data Processing Model

The two fundamental building blocks of stream processing are operators, which process data, and buffers, which forward and buffer data (e.g., to form finite windows over infinite streaming data). The data processing model of the exoengine architecture thus considers operators, which we call *slets*, and buffers, which we call *channels*. Figure 3 illustrates the model as a mesh of channels (rectangles) and slets (ovals). Slets have input and output ports, and each port can connect to one channel. In the figure, ports are implicitly illustrated as the places where arrows enter or leave slets. π-*slets* process data and behave like conventional operators. They can have any number of input and output ports. At the edges of the processing mesh, α-*slets* adapt data sources and ω-*slets* adapt data sinks, providing clean interfaces for exchanging data with the platform. α-slets have no input ports, they only receive data from external sources. Likewise, ω-slets have no output ports and only send data to external sinks. Sources and sinks can be any external device, application, or component that emits or consumes data, respectively.

Data is processed as discrete *items* (tuples) that flow from the sources on the left through the application mesh to the sinks on the right. Slets can transform data that arrives at an input port in any way, including dropping it, aggregating it into internal state, or creating and emitting new data. 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 exoengine, data tables in databases) with the view definition. In databases, the view definition is a query and in the exoengine architecture it is the part of the mesh that is connected to the channel's input. Channel implementations can persist their contents, resembling materialized views in databases.

Processing can be push- or pull-driven to support all existing systems and to be able to combine them. Thus, the puristic, basic interfaces for data exchange define two methods:

PAN Exhibit 1013

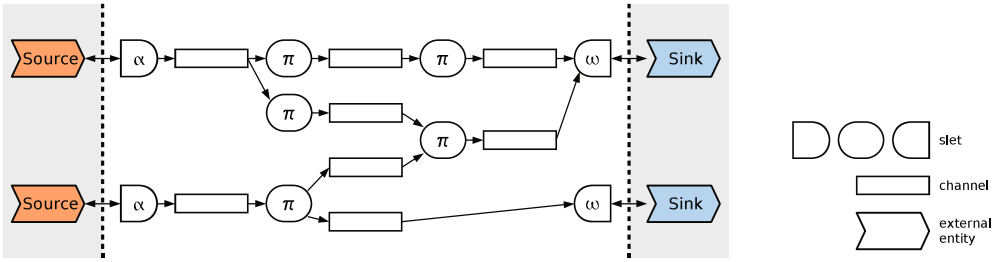

Fig. 3. Data processing model

push(item): push an item to the input of a channel or an slet
pull(): request items from the output of a channel or an slet, return a set of 0. . . n items.

Advanced functionality and optimizations can be implemented as extensions to the interfaces, including channels that provide individual windows for connected slets based on count, time, or explicit eviction from the buffer by the slet (semantic windows); sharing multiple windows and the materialized view from the same physical buffer; or allowing pull requests that are augmented by a query to push selectivity towards data sources. We use channels that materialize their contents or provide windows to slets for the Linear Road implementation presented in the evaluation (see Sect. 5).

2.3 Implementation Model

The implementation model captures implementation aspects, which include actual component interaction and distributed operation. It is based on service-oriented software design and the flexible and loose coupling between components, which facilitates dynamic changes to the processing mesh. It adds an additional type of component, *connectors*, as an indirection between slets and channels used to capture distributed operation in the model. Figure 4 illustrates the implementation model. It depicts all implementation details from slets emitting items on the left, to *input* connectors (*Conn.*), to a channel, to *output* connectors, to slets consuming items on the right. Ports, buffers, and the implementation of slets have been made explicit in the illustration.

Connectors are part of the virtualization strategy to enable distribution. They can be omitted as an optimization if channel and slet reside on the same instance of the platform. In the distributed case, where a channel residing on one instance of the platform is accessed by slets residing on another remote instance, *remote connectors* encapsulate communication between the instances of the platform. One half of the remote connector is installed on the platform instance of the channel and the other half is installed on the remote platform instance, where it represents (*proxies*) the channel. For every remote platform instance that accesses the channel, one remote connector is used to serve all slets on that instance. Section 4.3 with Fig. 6 illustrates distributed operation.

Using connectors as local proxies of channels can also improve performance and robustness. *Smart connectors* can, e.g., 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

PAN Exhibit 1013

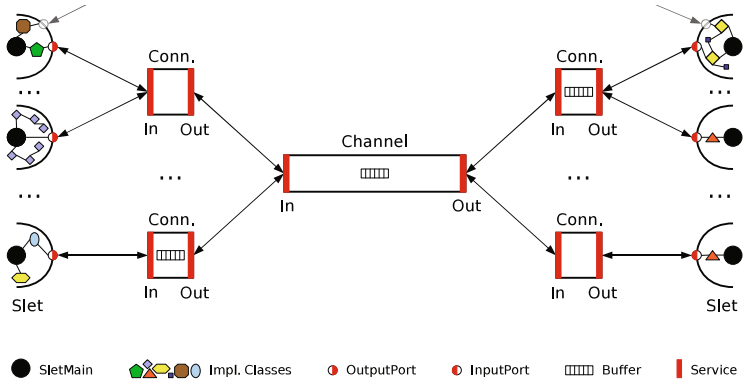

Fig. 4. Implementation model

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 interfaces.

In Fig. 4, 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. Service interfaces used in data exchange are depicted using thick bars. To exchange data, components call the *push(item)* method on the *InputPort* or *In* services, or the *pull()* method on the *OutputPort* or *Out* services.

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.

2.4 Component Life Cycle Management

Every component running on top of an exoengine platform also provides a management service. At runtime, the platform interacts with a 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), 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).

3 Stream Processing with the Exoengine Platform

One way to use our platform is to implement from scratch fully distributed, heterogeneous data streaming applications. This implies writing or synthesizing each operator and the additional components for our platform. As an example, we are in the process of implementing a wide-area, peer-to-peer stream processing infrastructure for exchanging personal data in a streaming manner across collections of devices and locations [11].

PAN Exhibit 1013

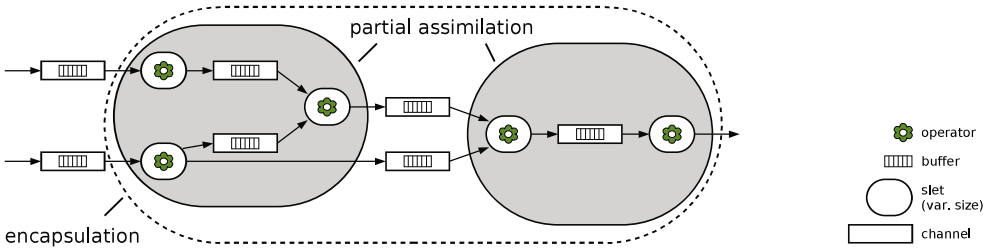

Fig. 5. Wrapping alternatives

3.1 Porting Existing 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 our platform. We distinguish three levels of granularity for porting existing SPEs: individual operators ("assimilation"), bare stream processing engines ("partial assimilation"), and complete applications ("encapsulation").

If a streaming 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, respectively. This allows reusing operators and buffers without changing the semantics of the underlying engine, while opening up all three possibilities illustrated in Fig. 1. Assimilation is illustrated in Fig. 5 by individual operators and buffers being wrapped as slets and channels.

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 (Sect. 5) to wrap existing XQuery [7] and STREAM [20] engine implementations. Partial assimilation reduces the porting effort, while still providing access to intermediate results and allowing distributed deployment. It opens up all three possibilities illustrated in Fig. 1 but at a coarser granularity (e.g., at the level of individual queries) compared to full assimilation. In Sect. 5.6 we show the effectiveness of turning a centralized application composed of multiple instances of a partially-assimilated XQuery engine into a distributed application and, thereby, enabling it to handle higher load. Partial assimilation is illustrated in Fig. 5 by the two big, gray slets containing multiple operators and buffers/internal state, which are not explicitly visible to the exoengine platform.

In encapsulation, we wrap an entire application with all the engines and queries into a single slet. This 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, dashed slet in Fig. 5, which encapsulates everything.

3.2 Extensibility

The exoengine model generally matches stream processing applications. The functionality provided by the interfaces (*push(item)* and *pull()*), however, is only suitable for

basic data exchange and not sufficient to implement a full-fledged SPE. The interface between operators and storage/buffer instances in an SPE is typically richer and additionally supports, e.g., index-based access to data or bulk access to multiple tuples at once. The exoengine architecture supports any kind of interaction between slets and channels and thus allows to keep intrinsic implementation details of existing SPEs.

Slet and channel implementations can extend the service interfaces for data exchange (thick bars in Fig. 4). 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. In the local case, connectors are empty and simply omitted. In the distributed case, standard remote connectors only need to pass method calls through. Thus, they are created automatically by the platform by inspecting the extended interfaces of the components using, e.g., reflection. Distribution-aware implementations of SPEs for the exoengine architecture can provide implementations of smart connectors for enhanced remote operation.

In addition to extensions of interfaces on the data path, implementations of components can also extend their management interfaces 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 richer configuration mechanism.

4 Platform Implementation

In this section, we discuss aspects of implementing an exoengine platform conforming to the data processing and implementation models presented in the previous sections. We also discuss how to implement applications running on top of it and present our prototype of an exoengine platform, ExoP. Though we have implemented the aspects discussed below in our prototype, they are applicable to any platform implementation.

4.1 Component Implementation

Components for the exoengine 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. 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 architecture and potential extensions thereof. In addition, every component implements the respective management interface (slet, connector, or channel) and potential extensions to it.

An implementation of the exoengine platform provides the generic parts of slets, connectors, and channels as a library. These generic parts contain the necessary and recurring glue code that deals with registering a component's services with the underlying service framework, creating and destroying slet ports, exchanging configuration and state with the exoengine platform, and retrieving the connected components' input

PAN Exhibit 1013

(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 without having contact with the details of the underlying service-based implementation of the exoengine platform.

4.2 Application Builders

Applications are created, modified, and removed by *application builders*. These provide the programming interface and abstraction to the system, typically through a high-level, declarative interface like a streaming query language or a graphical user interface.

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. A mesh of slets and channels can have cycles and it is the responsibility of the application builder to ensure that this does not cause adverse effects.

Similar to the implementation of components, application builders do not need to know the details of the underlying service-based implementation of the platform. Instead, a management service provided by the platform 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 a response to these methods, the management service creates instances of slets, channels, and connectors; assigns a unique component identifier and configuration (e.g., the query that a particular slet is executing or whether a channel should persist data); and registers them under the corresponding service interfaces. The platform persists the configuration of every component and whether the component is active. This information is reused when an application or parts of it are restarted (e.g., migration). Then, components are recreated, receive the persisted configuration, and are started automatically, relieving application builders from these tasks.

4.3 Distributed Operation

Application builders access remote platforms through a remoting service provided by the platform. Similar to the management service, it 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 or 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 between suspending and resuming a component.

PAN Exhibit 1013

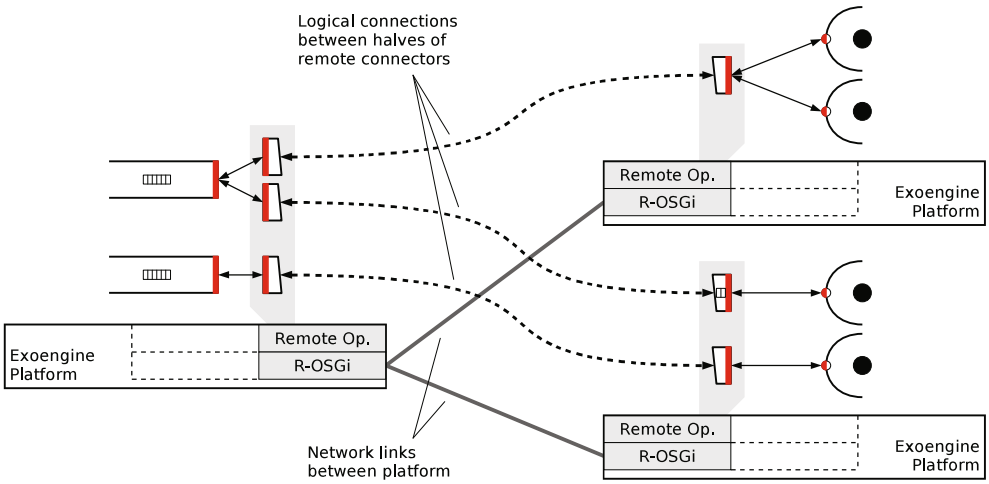

Fig. 6. Distributed operation using connectors

Every instance of the platform maintains one connection to every remote platform instance. Through this connection, all communication takes place, as is illustrated in Fig. 6. Remote connectors are provided by the *remote operations* component of ExoP. By default, network failures result in the removal of the remote connector service, which appears like any other dynamic change to the processing mesh. Smart connectors can override this behavior and remain registered and thus connected.

4.4 Prototype

We have implemented the ideas presented in this paper in *ExoP*, our prototype of an exoengine platform. ExoP is implemented as a componentized, service-oriented system using Java and OSGi. This section presents some of its implementation details.

OSGi. ExoP is based on the OSGi Service Platform [22]. OSGi is a widely used (e.g., Eclipse IDE, application servers) framework for module management and service composition for Java. 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 handles the dependencies that arise in the process.

Services are implemented as 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 object registered as the service. OSGi provides loose coupling and dynamic service composition within a Java VM. The open source project R-OSGi [25] extends OSGi to support dynamic service composition across multiple Java VMs.

ExoP is implemented as a set of OSGi bundles and uses R-OSGi as the communication fabric to interact with remote platforms (see Fig. 6). ExoP is modular and dynamic itself and a subset of its bundles (e.g., management) can be (un)loaded at runtime.

Component Implementation. ExoP provides an API that facilitates the implementation of slets and channels. Using the example of slets, one class of an slet implementation must implement the interface *SletMain*, which defines methods that are called at initialization or state transitions. In Fig. 4, this class is represented as solid black disk.

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. Through this, slets interact with the platform to create and destroy ports, and to update configuration and state.

ExoP's component model for slets, channels, and connectors extends OSGi's lifecycle management facilities. The API hides the details of the service-based design and allows developers to concentrate on the actual logic of the slet. Interactions with the OSGi platform, like registering services or persisting configuration, is implemented in ExoP and happens behind the API. For example, when an slet calls the 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 *InputPort* service interface and with the identifier as service property.

When an instance of an slet is created, a configuration object for that instance is created and the configuration is persisted with OSGi's Configuration Admin service, resulting in a callback to the particular instance of a Managed Service Factory implementation. The factory then creates an instance of a generic *SletImpl* and an instance of the specific *SletMain* slet implementation (supplying the instance of *SletImpl* as *SletUtil*), calls initialization and start methods on it, and eventually registers the *SletImpl* under the *Slet* service interface and with a set of service properties (including the unique Slet instance identifier) with the OSGi service registry. These steps are implemented in ExoP's management bundle and hidden behind its *ComponentManager* service interface.

Component Binding and Interaction. 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 a higher cardinality. When a port is connected to a connector, the connector's unique identifier is saved in the port's "connected to" property. Likewise, for connectors and channels, the channel's identifier is saved in the connector's property.

When a component wants to interact with the component(s) it is connected to (i.e., call a method on their service interface), it fetches the matching components according to the specific "connected to" property and unique identifiers. For example, a channel fetches all connectors that have the channel's unique identifier in their specific "connected to" properties, while a connector fetches the channel with the unique identifier that is saved in the connector's specific property. Even though the setup of the mesh typically hardly changes, properties need to be matched for every interaction between components, which is a rather expensive operation. Therefore, we make heavy use of OSGi's service tracker, which pro-actively tracks and caches matching components, similar to a proxy. The details of setting up and persisting component bindings with unique identifiers, service properties, and service trackers are implemented in ExoP's management bundle and hidden behind its *WiringController* service interface, providing straightforward methods to connect and disconnect components.

PAN Exhibit 1013

5 Evaluation

We have ported the MXQuery engine and the Linear Road benchmark (LRB) implementation presented by Botan et al. [7] to ExoP using partial assimilation (see Sect. 3.1). The evaluation measures the overhead of the exoengine approach, demonstrates the features and benefits gained by porting MXQuery to ExoP (i.e., capability for dynamic modifications and extensions, distributed operation, federation with different SPEs), and describes the porting effort.

5.1 The 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) enters 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 a as measure of performance impact.

5.2 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, 3-hour-long run with an input load of $L = 5.0$.

5.3 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 query. 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.

The MXQuery engine was wrapped as a π-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

PAN Exhibit 1013

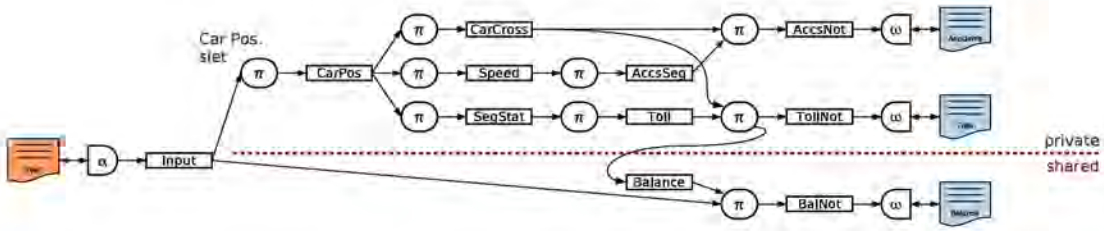

Fig. 7. Linear Road benchmark implementation on ExoP

interfaces of ports and channels. Furthermore, 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 was wrapped as α- and ω-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 with custom glue code was turned into an application builder. The application builder registers the slets (MXQuery, data loader, data writer) and the channel implementations with ExoP. It then instructs the platform to create respective instances thereof and to 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 ExoP's configuration mechanism. Once the application builder has completed setting up the Linear Road benchmark implementation in ExoP (as illustrated in Fig. 7), processing starts. With the exception of the daily expenditures query[1], the implementation is the same as the original one presented by Botan et al. [7] and consists of 9 instances of the MXQuery engine, each processing a different query.

5.4 Overhead of the Exoengine Architecture

Since every additional layer potentially adds overhead to a system, we first measure the overhead incurred by the modular and dynamic design employed by our platform. Figure 8 compares the original implementation of the Linear Road benchmark ("Without") with the ExoP version ("With"). Both implementations can handle the load well and the overhead added to the average processing time of tuples is negligible (147.90 ms vs. 148.92 ms). Table 1 provides additional details about the experiment runs. It counts output tuples grouped by processing time (bins of 1 second) in the 2nd and 3rd column.

5.5 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 LRB workflow after 1 hour of the 3-hour-long run of the benchmark with a native implementation, while

[1] We have removed the daily expenditures query from both the original and the ExoP implementation due to its negligible impact on performance and the effort required to deploy the historical data.

PAN Exhibit 1013

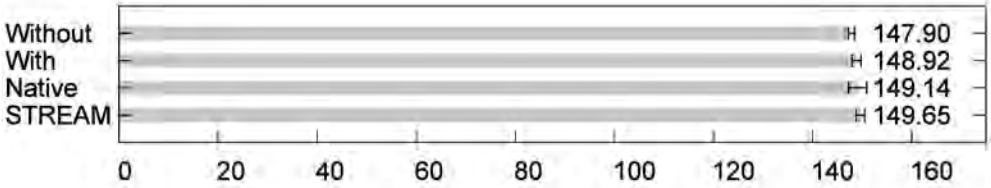

Fig. 8. Average tuple latencies [ms]

the benchmark is running. 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 Fig. 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 Fig. 8 show. Table 1 again provides additional details of the corresponding experiment runs in the 3rd and 4th column.

5.6 Distributed Deployment

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 16 cluster nodes on a switched gigabit ethernet. Each node has two Xeon L5520 CPUs (quad-core, 2.26 GHz) and 24 GB RAM. They run Ubuntu 10.04 64 bit and Oracle's JDK 6u22 with maximum heap size set to 5 GB. The maximum possible load for a single node is $L = 4.5$. However, applications designed for the exoengine architecture can easily be transformed into distributed systems by introducing remote invocations between components running on different machines. For the Linear Road benchmark, we chose the partitioning depicted by the dashed line in Fig. 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 spared enough capacity on the master node for the shared part of all experiment setups up to 16 nodes.

Figure 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. The overall tuple latency only increases significantly for the first few added nodes 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

PAN Exhibit 1013

Table 1. Tuple count grouped by processing time

Time	Without ExoP	With ExoP	Native after $1h$	STREAM
$[0,1)\ s$	11 329 044.00	11 333 460.50	11 333 349.00	11 324 352.00
$[1,2)\ s$	52 042.75	48 511.00	48 009.75	53 378.75
$[2,3)\ s$	14 830.25	14 367.75	14 373.00	17 216.75
$[3,4)\ s$	1 755.00	1 332.75	1 940.25	2 724.50
$[4,5]\ s$	0.00	0.00	0.00	0.00
$(5,\infty)\ s$	0.00	0.00	0.00	0.00

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 small, steady increase in latencies reflects this effect.

The experiment shows that we can use ExoP 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 ExoP's communication system, used by remote connectors which appear to the MXQuery engine slet like a connector to a local store. We chose a synchronous and straightforward implementation of the remote connector to capture all impacts of network communication. Depending on application semantics, asynchronous remote connectors, with queues, can be used to cut latency.

5.7 Developing with the Exoengine

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 inkling 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 Sect. 4.4 for details) and the other class implements the actual filtering functionality. The main class consists of 55 lines of source code, out of which all but 9 lines have been generated from the *SletMain* interface. Packaging an slet implementation for ExoP only requires the addition of 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 with a set of interfaces and small helper classes, which 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 only had to change certain data types of the MXQuery engine to implement the *Serializable* interface so that we could ship instances to other machines. The remote invocations along the boundaries of OSGi services happen transparently with R-OSGi.

PAN Exhibit 1013

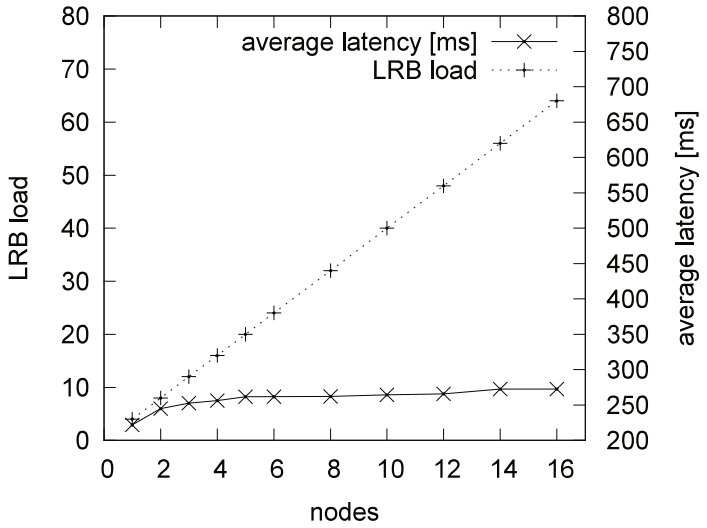

Fig. 9. Scale-out of MXQuery-based LRB implementation on ExoP

The MXQuery and the STREAM engines [20], storage implementations of MXQuery, data loaders and writers were ported by us and are reusable as a library. Additional applications for these engines can therefore be deployed right away.

5.8 Heterogeneity

ExoP 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 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 Fig. 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 to the average

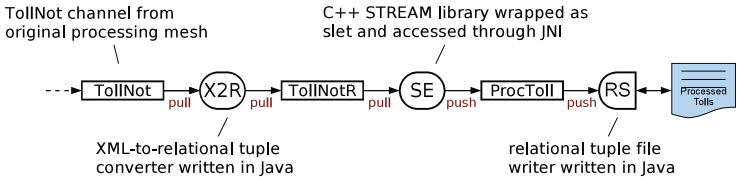

Fig. 10. STREAM engine attached to toll notifications

latencies (compare "With" and "STREAM" in Fig. 8) and the benchmark runs well within limits, as can be seen in the 5th column in Table 1.

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.

6 Related Work

The fundamental difference between the research [2,20,9] and commercial products [27, 29, 4, 16] mentioned in the introduction and our exoengine architecture is that it is not yet another SPE. Rather, it provides a platform for facilitating application development, deployment, integration, and management of existing or new-to-be-built SPEs.

Related work on distributed SPEs focused on functionalities like load management (e.g., [1,24]), fault tolerance and high availability (e.g., [1]), integration with sensor networks (e.g., [14, 3]), and performance tuning (e.g., [17]). 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. Closer to our work, the XFlow Internet-scale distributed stream processing system proposes a loosely-coupled architecture for query deployment and optimization, focusing on an extensible cost model [23]. XFlow does not provide any abstract programming models or techniques for building, hosting, or porting various SPE components.

There are a few platforms proposed for facilitating the development of stream-based applications, such as System S, Auto-Pipe, MaxStream, or PIPES. System S [17] includes a distributed runtime platform that facilitates dynamic stream processing. The platform pursues similar goals in terms of deployment as exoengine does but 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 constrast, exoengine is an independent, pure middleware approach, and as such, is usable for many different SPEs. Auto-Pipe [8] 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 parts of applications can be run on the platforms that will provide the highest performance for them. This work focuses on diverse hardware platforms, whereas we focus on diverse SPEs.

PAN Exhibit 1013

The MaxStream architecture [6], 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. PIPES [18] 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 exoengine 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.

The importance of elastic stream processing has also been recognized by related work recently [26]. This work focuses on elastically scaling the performance of individual streaming operators on multicore machines, whereas our work provides a more general architecture for distribution and a platform that can also serve as basis for elastic stream processing. Yahoo's S4 [21] provides an architecture and platform for processing streaming data similar to MapReduce [10] for stored data, and the similar key property of a specific, simple processing model that enables automatic parallelization and deployment on a large number of machines. StreamCloud [15] 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. The exoengine approach provides a platform that hosts different streaming engines and could benefit from StreamCloud by integrating it as application builder. Lastly, in our XTream project [11], we explore how an exoengine-like platform and stream processing in general can facilitate personal information processing and dissemination at global scale.

Publish/subscribe systems also provide mechanisms and an infrastructure to disseminate and filter data from sources to sinks [13]. They decouple senders from receivers by topics, which can be modeled by channels in our architecture. However, they do not support sophisticated in-network data processing, distributed operation in a peer-to-peer manner, access to intermediate results, or in-network storage, as stream processing with the exoengine does.

7 Discussion and Outlook

In this paper, we have proposed a new architecture for implementing data stream processing applications by virtualizing components of stream processing systems and deploying them on a common middleware platform. While being radical in its puristic approach inspired by the exokernel architecture, the non obtrusive nature of the approach—it does not dictate a specific query language, algebra, operator implementation, or scheduling model—allows to leverage any existing stream processing system and its particular strengths. In contrast, yet another concrete implementation of a stream processing system would require a much more radical reimplementation of existing applications. The exoengine architecture defines the fundamental elements of stream processing (slets/operators, channels/buffers) using extensible interfaces to allow rich interaction between specific slet and channel implementations of a particular system, while retaining basic data exchange capabilities with other systems.

PAN Exhibit 1013

Depending on the granularity of the integration of streaming systems with the exoengine platform (see Sect. 3.1), the benefits range from the automatic management of deploying and executing an encapsulated application and its federation with applications for other engines (through its ultimate inputs and outputs) to the reuse of engines or individual operator implementations and the ability to replace them with a different implementation at runtime, as demonstrated in Sect. 5.5. The architecture transparently provides data transport in a distributed setup and allows to run centralized engines in a distributed setting, as demonstrated with the scale-out experiment in Sect. 5.6.

Finally, we have provided brief conceptual instructions for building an exoengine platform using SOA in Sect. 2.3 and discussed concrete implementation aspects as well as showed a concrete prototype implementation in Sect. 4. The prototype confirms that the dynamic nature and additional indirections (ports, connectors) can be implemented efficiently with negligible overhead, as validated in Sect. 5.4.

Future work includes automatic state capturing of slets for migration (currently slets need to implement serialization and deserialization of internal state to allow migration); deriving common, generic channels and slets (e.g., round robin distributor) and providing them as a base library (similar to the platform providing common functionality for deploying, running, and managing configuration and state); and extending the exoengine's area of application to elastic/cloud computing—the holy grail of dynamic operation, automated management, and distributed deployment.

Acknowledgments. Part of this work was funded by the Swiss National Science Foundation SNF ProDoc program and by the Microsoft ICES initiative.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: CIDR (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. The VLDB Journal 12(2), 120–139 (2003)
3. Aberer, K., Hauswirth, M., Salehi, A.: A Middleware for Fast and Flexible Sensor Network Deployment. In: VLDB (2006)
4. Ali, M.H., Gerea, C., Raman, B.S., Sezgin, B., Tarnavski, T., Verona, T., Wang, P., Zabback, P., Ananthanarayan, A., Kirilov, A., Lu, M., Raizman, A., Krishnan, R., Schindlauer, R., Grabs, T., Bjeletich, S., Chandramouli, B., Goldstein, J., Bhat, S., Li, Y., Di Nicola, V., Wang, X., Maier, D., Grell, S., Nano, O., Santos, I.: Microsoft CEP server and online behavioral targeting. Proc. VLDB Endow. 2, 1558–1561 (2009)
5. Arasu, A., Cherniack, M., Galvez, E.F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark.. In: VLDB (2004)
6. Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., Lee, C., Mundada, G., Shan, M., Tatbul, N., Yan, Y., Yun, B., Zhang, J.: Design and Implementation of the MaxStream Federated Stream Processing Architecture. Tech. Rep. TR-632, ETH Zurich Department of Computer Science (2009)
7. Botan, I., Kossmann, D., Fischer, P.M., Kraska, T., Florescu, D., Tamosevicius, R.: Extending XQuery with Window Functions. In: VLDB (2007)

PAN Exhibit 1013

8. Chamberlain, R.D., Franklin, M.A., Tyson, E.J., Buckley, J.H., Buhler, J., Galloway, G., Gayen, S., Hall, M., Shands, E.B., Singla, N.: Auto-Pipe: Streaming Applications on Architecturally Diverse Systems. IEEE Computer Magazine 43(3), 42–49 (2010)
9. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR (2003)
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI (2004)
11. Duller, M., Alonso, G.: A lightweight and extensible platform for processing personal information at global scale. Journal of Internet Services and Applications 1, 165–181 (2011)
12. Engler, D.R., Kaashoek, M.F., O'Toole Jr., J.: Exokernel: An Operating System Architecture for Application-Level Resource Management. In: SOSP (1995)
13. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
14. Franklin, M.J., Jeffery, S.R., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., Hong, W.: Design Considerations for High Fan-In Systems: The HiFi Approach. In: CIDR (2005)
15. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Valduriez, P.: StreamCloud: A Large Scale Data Streaming System. In: ICDCS (2010)
16. IBM InfoSphere Streams, `http://www-01.ibm.com/software/data/infosphere/streams/`
17. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In: SIGMOD (2006)
18. Krämer, J., Seeger, B.: PIPES - A Public Infrastructure for Processing and Exploring Streams. In: SIGMOD (2004)
19. Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., Hyden, E.: 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)
20. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: CIDR (2003)
21. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed Stream Computing Platform. In: ICDMW (2010)
22. OSGi Service Platform, `http://www.osgi.org/`
23. Papaemmanouil, O., Çetintemel, U., Jannotti, J.: Supporting Generic Cost Models for Wide-Area Stream Processing. In: ICDE (2009)
24. Pietzuch, P.R., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.I.: Network-Aware Operator Placement for Stream-Processing Systems. In: ICDE (2006)
25. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Pasquale, F. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)
26. Schneider, S., Andrade, H., Gedik, B., Biem, A., Wu, K.L.: Elastic Scaling of Data Parallel Operators in Stream Processing. In: IPDPS (2009)
27. StreamBase Systems, Inc., `http://www.streambase.com/`
28. Tatbul, N.: Streaming data integration: Challenges and opportunities. In: NTII (2010)
29. Truviso, Inc., `http://www.truviso.com/`
30. Zeitler, E., Risch, T.: Scalable Splitting of Massive Data Streams. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 184–198. Springer, Heidelberg (2010)

PAN Exhibit 1013