SPC: A Distributed, Scalable Platform for Data Mining

Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, Chitra Venkatramani

IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10598

ABSTRACT

The Stream Processing Core (SPC) is distributed stream processing middleware designed to support applications that extract information from a large number of digital data streams. In this paper, we describe the SPC programming model which, to the best of our knowledge, is the first to support stream-mining applications using a subscription-like model for specifying stream connections as well as to provide support for non-relational operators. This enables stream-mining applications to tap into, analyze and track an ever-changing array of data streams which may contain information relevant to the streaming-queries placed on it. We describe the design, implementation, and experimental evaluation of the SPC distributed middleware, which deploys applications on to the running system in an incremental fashion, making stream connections as required. Using micro-benchmarks and a representative large-scale synthetic stream-mining application, we evaluate the performance of the control and data paths of the SPC middleware.

1. INTRODUCTION

The widespread deployment of digital systems has led to a large increase in the number of sources of streaming digital data such as text and transactional data, digital audio, video and image data, instant messages, network packet traces, and variety of sensor data. The Stream Processing Core is a distributed stream-mining middleware, built to support applications composed from user-developed processing elements that ingest, filter, and most importantly, mine data streams. SPC supports the stream-mining paradigm, where users submit *continuous* inquiries expressed as processing flow graphs that are evaluated over continuous streams of data. Similarly, results of the evaluation are "streamed" continuously to the user as they become available.

Our contributions in this paper are two-fold. First, we discuss a simple, yet very flexible programming model supported by SPC for developing stream-mining applications. Second, we describe the distributed architecture of SPC middleware and its relationship to the programming model and how it addresses the scalability needs of large-scale stream-mining applications.

Many stream processing systems in the literature [1, 2, 17] are designed for applications employing only relational operators (and their adaptations to continuous streams) without any support for user-defined operators, which is required in many application domains. The SPC programming model supports both relational operators and user-defined operators alike. Most other stream-processing systems also precompile the processing flow-graph before deploying it into the runtime. SPC, on the other hand, allows for flow-graphs to specify the streams to process using a subscription-like model. This enables stream-mining across raw data streams and data streams generated by other applications concurrently using the middleware. This dynamic nature fits the stream-mining model where information sources are ever-changing and long-running stream-mining inquiries are looking for "relevant" information in the continuously changing set of data streams. Finally the SPC architecture capitalizes on several programming model features to achieve scalability and performance efficiency. For example, SPC uses a two-tiered data routing approach which may accord substantial savings over a traditional pub-sub model for data dissemination.

The key design points addressed by the SPC architecture are: (i) data processed by applications is streamed in nature, (ii) applications are not restricted to using relational operators, (iii) applications are inherently distributed due to their resource requirements, (iv) support for discovery of new data streams, as they are created in the system, is essential. That is, applications may dynamically tap into relevant streams possibly created by other applications, (vi) since applications share streams, processing elements belonging to different applications need to share a common vocabulary to annotate discovered data in streams, and finally, (v) volume of incoming data is very large and hence the system must be highly scalable.

The rest of this paper is organized as follows: Section 2 describes and contrasts SPC with other stream processing systems. Section 3 describes the SPC programming model, which is the cornerstone of the middleware and the gateway for application developers into the runtime environment. Section 4 provides an overview of the SPC architecture. Section 5 provides a performance study of a few critical system metrics by employing micro-benchmarks and a large-scale synthetic application where the middleware is deployed and stress-tested on 85 multiprocessor nodes in a cluster environ-

DM-SSP'06 August 20, 2006, Philadelphia, PA USA

ment. Finally, Section 6 concludes the paper and discusses ongoing work towards evolving SPC's architecture.

2. RELATED WORK

Recently a large number of projects aimed at supporting streamed data processing have been started. Not surprisingly, many of these initiatives are more slanted towards extending current database technology, although stream data processing has been an active area of research in the programming languages as well as in the high performance computing communities.

Current projects that aim to process unstructured information, such as GATE (General Architecture for Text Engineering) [10] and UIMA (Unstructured Information Management Architecture) [11], focus on mining text documents for information. These systems are designed with a focus on flexibility in assembling *text* mining applications using generic algorithms and not necessarily with a focus on large-scale distributed multi-operator mining over multi-modal data.

On the other hand, the InfoSphere project [14, 18, 19], aims at defining system support for writing information-driven applications, focusing explicitly on defining the desired QoS-level between distributed components that interact through streams. Unlike in SPC, application composition is done at compile-time.

On the high performance front, DataCutter [7] is a middleware for decomposing applications into processing *filters* responsible for implementing *subsetting* and *reduction* operations over streams of buffers. The lack of support for dynamic topologies and the lack of support for multiple coexisting applications are the main differences between DataCutter and SPC.

In the database community, projects such as Aurora [1], TelegraphCQ [8, 17], Borealis [2, 9], and STREAM [4, 5] have been making progress in providing support for streamed data manipulation from a database-centric perspective. That is, data is manipulated as in relational databases, by specifying queries (sometimes with time windows) over regular relational data as well as streamed data.

While SPC shares a few directions with some of the projects we discussed, it distinguishes itself from these systems in three main ways: (i) SPC provides a programming model to write processing elements whose semantics are beyond relational operators; (ii) SPC supports dynamic application composition and stream discovery, where multiple applications can symbiotically interact; (iii) SPC adopts a design that enables scaling to a very large number of computational nodes supporting multiple coexisting applications.

3. PROGRAMMING MODEL

The Stream Processing Core includes a programming model and development environment that enables the implementation of distributed, dynamic, and scalable applications. SPC's programming model includes APIs for declaring and creating processing elements as well as tooling to assemble, test, visualize, debug, and deploy applications. Unlike other stream-processing middleware, the SPC programming model supports non-relational operators and relational operators alike. The programming model has been shown to be flexible and comprehensive. Applications from various domains have already been implemented, including the Linear Road benchmark [13] and a semantic video filtering application [15, 16].

SPC has been built to answer *inquiries*, which are high-level user requests for information. An inquiry is translated into one or more processing flow graphs, which we refer to as a job or an application. Jobs comprise one or more processing element. The *processing element (PE)* is the fundamental building block for creating applications that use the services provided by the SPC and implements an application-defined operator. PEs communicate through *ports* and are able to read, process, and write streams of data. A PE receives data from a collection of streams through *input ports*, process and writes resulting data to *output ports*, thus creating new streams. Each port is associated with a declared *format*, which is a set of attribute types provided for each PE port. The format for an input port defines the structure of an SDO the PE can ingest through that port. An input port subscribes to data it intends to consume using a *flow specification* expression. It includes properties of streams that the PE intends to receive as well as a predicate to filter SDOs from the streams. The format for an output port defines the structure of SDOs in the stream produced by the output port. Each output port produces a stream, which is a contiguous group of stream data objects (SDOs) conforming to the *format* for the output port. That is, SDOs are structured messages consisting of annotations and an associated payload.

The rest of the discussion in this section centers on the process of creating processing elements and assembling applications.

Since one of the key features of SPC is the sharing of streams across applications, all SDO annotations are typed using types declared in a *global type system*. This enables applications to be composed from PEs written by different developers and different applications to share and cooperate through streams.

3.1 Global Type System

The global type system is a library of agreed-upon types. Types are like C-programming language *typedefs* where each type definition has a type name and may contain one or more typed features. Type definitions may also be hierarchical in that they inherit all the features of their parent type. The global type system in SPC defines a few basic types such as `string`, `int` and `float` and many other complex types are derived or composed from these basic types. For example, the description for a type named com.ibm.systems.SourceIPAddress is shown in Figure 1. This type has a description and the declaration of a feature called `value`, which is of type `String`. This type is used by the stream definition associated with the DemultiplexedEmailTraffic port depicted in Figure 2.

In SPC, the type system definitions are created by and maintained in a central repository. C++ and Java classes corresponding to each type definition, with *getters* and *setters* for each feature within the type are automatically generated to facilitate programmatic access to instances of types carried by the SDOs.

3.2 Processing Element Description

Since a PE is a basic building block of an application, it is essential that it is described adequately to enable either a human or an automated tool to compose applications. The

```
<typeDescription>
    <name>com.ibm.systems.SourceIPAddress</name>
    <description>A source IP address</description>
    <features>
        <featureDescription>
            <name>value</name>
            <description>IP address as string</description>
            <rangeTypeName>String</rangeTypeName>
        </featureDescription>
    </features>
</typeDescription>
```

Figure 1: Excerpt of the definitions in the global type system. The definition for type com.ibm.systems.SourceIpAddress

SPC programming model requires a PE developer to create a static descriptor tied to the PE implementation. This describes the PE's ports as well as its deployment requirements such as its library dependencies, resource requirements, executable location, and command line arguments.

The description of the ports is key to the dynamic application composition performed by SPC. Both input and output ports have a declared *format*, which consists of a set of type names constraining the properties of SDOs that can be manipulated through those ports. An input port format specifies the types of the annotations that the PE implementation will look for in every incoming SDO on that port. An output port format declares to the system the types of annotations that will be present in all the outgoing SDOs produced by that port.

An excerpt of a descriptor is depicted in Figure 2 for a PE that demultiplexes email traffic into different protocols (e.g., SMTP, POP, IMAP). In particular, the descriptor contains the properties for the streams this PE should consume through the EmailTraffic input port and the information about the streams it produces through the output port DemultiplexedEmailTraffic. It also includes runtime configuration such as command line arguments (args) and executable (exec).

3.2.1 Output Port Stream Description

The stream is another basic concept in the programming model. A stream consists of SDOs that are produced by a single output port. It is defined by the *format* of the output port and has a name, assigned by the application composer. Streams can be declared to be private or public. Only public streams are available to all applications in the system and can be tapped into by specifying the stream name in the flow specification. This matching process is done by the SPC middleware during runtime (additional information about this matching process is found in Section 4.1) .

3.2.2 Input Port Flow Specification

As described earlier, in order to enable sharing of streams among applications and allow for dynamically tapping into a changing set of streams and mining them, SPC allows PEs to use a subscription-mechanism for the input ports. The PEs specify the streams they are interested in consuming using a flow specification which is associated with each input port. It is important to note that the PE-developer specifies the *format* which defines the types to be found in the SDO, whereas the flow specification is filled in during the application composition process to specify the streams and SDOs that must be routed to the PE. The flow specification

```
<pe> <executable> <main>@exec@</main>
<arguments>@args@</arguments> </executable>
<input>
 <port name="Port1">
  <flowspecification><STREAM EmailTraffic]></flowspecification>
  <format>
   <attribute>com.ibm.systems.SourceIPAddress</attribute>
   <attribute>com.ibm.systems.DestinationIPAddress</attribute>
   <attribute>com.ibm.systems.SourcePortNumber</attribute>
   <attribute>com.ibm.systems.DestinationPortNumber</attribute>
   <attribute>com.ibm.systems.DataSource</attribute>
   <attribute>com.ibm.systems.IPTransportType</attribute>
   <payload>text</payload>
  </format>
 </port>
</input>
<output>
 <port name="Port2">
  <streamname="DemultiplexedEmailTraffic"/>
  <format>
   <attribute>com.ibm.systems.SourceIPAddress</attribute>
   <attribute>com.ibm.systems.DestinationIPAddress</attribute>
   <attribute>com.ibm.systems.SourcePortNumber</attribute>
   <attribute>com.ibm.systems.DestinationPortNumber</attribute>
   <attribute>com.ibm.systems.DataSource</attribute>
   <attribute>com.ibm.systems.IPTransportType</attribute>
   <attribute>com.ibm.systems.EmailProtocol</attribute>
   <payload>text</payload>
  </format>
 </port>
</output>
</pe>
```

Figure 2: Excerpt of a PE descriptor with an input port `Port1` and an output port `Port2` producing a stream named `DemultiplexedEmailTraffic`

```
# Stream specification
    STREAM IPTraffic EmailTraffic;
# SDO filters
    com.ibm.systems.IPTransportType:value*='Chat' OR
    com.ibm.systems.IPTransportType:value*='Email' OR
    com.ibm.systems.IPTransportType:value*='AVT' OR
    com.ibm.systems.IPTransportType:value*='Voice'
```

Figure 3: A sample flow specification expression

expression has two optional components: a stream filter and a predicate to filter SDOs within the matching streams. In the descriptor shown in Figure 2, input port Port1 has a flow specification subscribing to a stream named `EmailTraffic`). Any stream that has that name will be multiplexed into that port by the SPC at runtime. Hence, by defining a flow specification expression, a single input port can have several streams be routed to it. In addition to the streams to connect to, the flow specification can indicate that SDOs must be filtered according to a particular predicate, using an extended Java Message Service (JMS) syntax [12]. Figure 3 demonstrates how an input port can subscribe to two specific streams and filter out SDOs according to a disjunctive predicate.

The example in Figure 3 shows the *precise* description of the streams that should be routed to an input port (e.g., `IPTraffic`). The absence of named streams in the flow specification (or the specification of stream properties alone) implies a *loose* subscription. In other words, if a *loose* specification is provided, consumers and producers are matched solely based on port formats on an ongoing basis (i.e., every time a new stream becomes available).

3.3 PE API

PEs are built as dynamically loadable modules that run in the context of a container. Multiple PEs can be managed by a single PE container, which effectively hooks it up to the middleware. PEs are currently implemented in C++[1] by inheriting from a simple `PE` abstract class, where a `init`, `process`, and `shutdown` methods must be customized and event handlers may optionally be installed.

The `init` method may be used by the PE-writer to initialize its processing. Similarly, the `shutdown` method is automatically called by the PE container when the PE is requested to terminate and may be used for providing cleanup. The `process` method is where the actual data processing, i.e., reading SDOs from streams, their manipulation, the execution of data analysis operations, and the generation of output SDOs takes place. The event handler method, if implemented, is invoked in order to asynchronously communicate external runtime events to the PEs. One typical example is to inform a PE when no PE is subscribed to its stream. This allows the PE to scale down its resource utilization and go into a "quiet" mode.

A PE will typically perform some (or all) of the following operations: read from streams through one or more input ports, process data, and create one or more streams. An input port is the object used for reading SDOs from the stream (or streams) matching the port's flow specification. Input ports can be opened and closed (using `open` and `close` methods), SDOs can be synchronously read (using the `readNext` method), and input ports can be *selected* (employing the `select` method). The select operation is similar to the Unix `select` system call, that is, once SDOs are ready to be consumed through a port, the port is flagged as such. The programming interface for output ports is similar, with the exception that only SDO writing operations can be performed (using the `writeNext` method).

As previously seen, the SDOs that are created or ingested consist of two parts, a set of structured attributes and an opaque payload. The structured attributes are manipulated through a set of convenience classes that implement *getter* and *setter* methods for each type used by the PE as described in Section 3.1. The opaque payload is manipulated as a blob of a particular size.

3.4 Application Composition

PEs are the building blocks of applications. In some cases, applications have well-defined boundaries where a collection of PEs will act in concert in order to analyze data and produce results for a user. In other cases, applications will leverage and co-exist symbiotically with other pre-existing (or yet to exist) applications by consuming streams generated by them or by producing streams useful to others. In SPC, an application consists of a collection of PEs and some additional configuration metadata. The application description may specify the stream connections that will exist among the PEs, by configuring the actual flow specifications for each input port for every PE in that application. Note that the actual stream connections are not completely known *a priori* because input ports flow specifications do not necessarily hard-wire consumer and producer PEs. In other words, the flow specification for the input port establishes a predicate that will, at runtime, result in the streams and SDOs that an input port can consume.

[1]Java support is currently under development.

For example, when an application that consists of a source PE that advertises a stream *S1* and a sink PE that has a flow specification of *S1* is submitted to SPC, it will be placed in the runtime environment and a connection between them will eventually be made. Likewise, other PEs belonging to other applications may eventually tap into *S1*, when the middleware identifies that a newly created PE has a flow specification that matches *S1*.

4. ARCHITECTURE

In this section, we describe the architecture of the SPC system. Many basic requirements drove the design of the SPC architecture. As stated earlier, SPC supports large-scale distributed streaming applications and is hence designed to run on large-scale clusters. Applications need to tap into streams based on stream characteristics, using a subscription-like model. This flexibility in the programming model requires the middleware to establish connections on an ongoing basis, providing dynamic information discovery. The harmonious coexistence of applications sharing common computational resources requires monitoring and enforcement of resource consumption limits and policies. SPC's components that host the PEs enforce the resource limits determined by a global resource manager. Finally, applications running on behalf of users having different isolation requirements needs to be supported. For this, the SPC architecture employs PE-execution containers that host, manage, monitor and control a PE's life cycle.

Figure 4(a) depicts the SPC architecture where three nodes (nodes 1, 2, and 3) are allocated exclusively for some of the middleware daemons and two nodes are allocated for the PEs (node 4 and 5) as well as their managing components. Some of the SPC components such as the Dataflow Graph Manager are standalone daemons. Others such as the the PE Container and the Data Fabric Server are distributed across all the SPC nodes. These components support a logical topology of PEs and their stream interconnections, an example of which is shown in Figure 4(b).

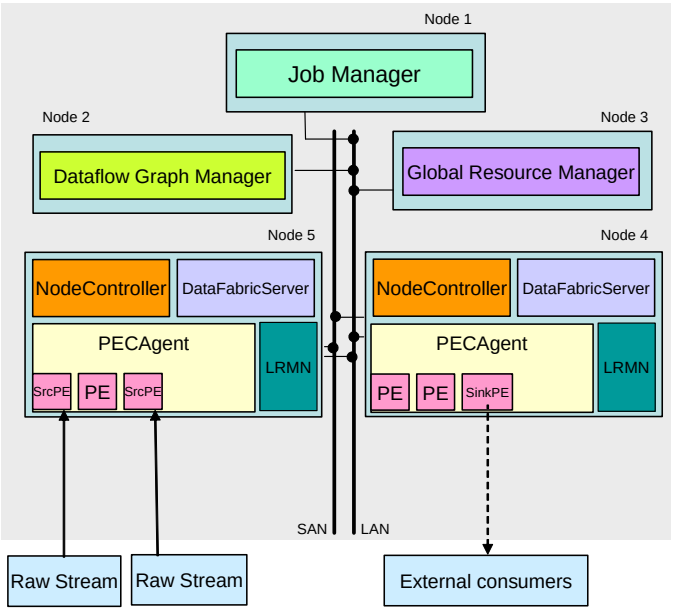
Each of the system components was developed as a multithreaded Corba service in C++ and the current inter-component communication infrastructure relies on remote procedure calls and asynchronous notifications. The SPC has been under development at IBM Research for over a year and our initial prototype is functional on an 85-node cluster. The current SPC code base consists of around 200 Kloc. The various SPC components are described in the following subsections.

4.1 The Dataflow Graph Manager

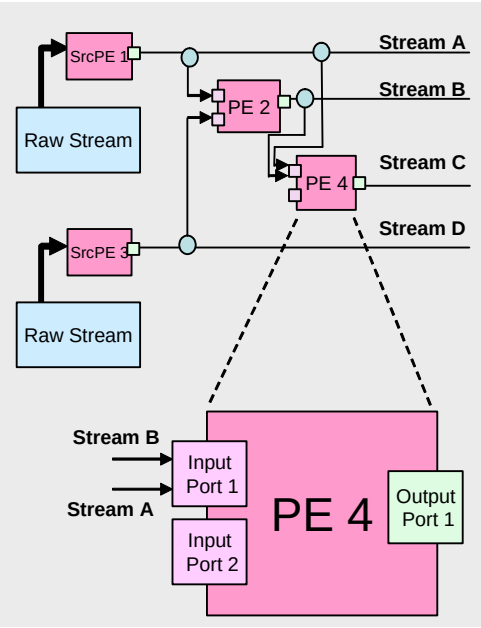
The Dataflow Graph Manager (DGM) manages the *global dataflow graph* in terms of PEs and stream connections among them. In order to provide this service, the DGM keeps a directory of existing PEs, streams, and the current graph topology. Some of the key functions of the DGM are described below.

4.1.1 Dynamic Stream Connections

As described in Section 3, SPC allows application composers to declare their input port flow specifications using a subscription model which is evaluated at run time to determine stream connections. This is one of the main tasks of the DGM. It achieves this by matching stream descriptions of output ports to flow specifications of input ports



(a) SPC distributed components. Nodes 4 and 5 host components that a typical processing node requires. Nodes 1, 2, and 3 each host an infrastructure control component



(b) Dataflow graph – a sample logical topology

Figure 4: The SPC physical and logical architecture. The physical interconnect can be provided by a Local Area Network and a Storage Area Network

whenever new PEs start running or leave the system.

Operationally, when a new PE is instantiated, the DGM determines all the stream connections to its input and output ports as follows – (1) *Subscription Lookup:* The DGM inspects the flow specifications of input ports of every PE already in the system for matches against streams produced by the new PE. The matching process also considers port format compatibility, i.e., an input port's format has to be a subset of any candidate output port format, to match. (2) *Stream Lookup*: Conversely, the DGM inspects the streams produced by every PE already in the system for matches against the flow specifications of the input ports of the new PE. Once matches are detected by either process above, the DGM asynchronously notifies the Data Fabric of the corresponding stream connections.

4.1.2 Two-Tiered Routing

As also seen in Section 3, the flow specification expression has two components (stream and SDO-filter) which enables the system to employ a two-level routing structure where – (1) candidate streams are matched against the stream portion of an input port's flow specification expression, allowing logical circuits to be established between consumers and producers; (2) SDOs from these matched streams are then filtered, at the communication substrate level (i.e., the Data Fabric), based on the SDO-filter, before delivery to the PE. In other words, DGM establishes logical circuits between PEs generating streams and PEs interested in consuming them. The physical connections are made by the Data Fabric upon notifications from the DGM.

Unlike traditional pub-sub scheme where single-tiered routing is usually employed, our design choice was made based on two assumptions. First, we believe that there is a more balanced ratio of publishers and subscribers in streaming systems, which makes it possible to pre-establish connections among them and then performing the per-SDO matching of SDO-filter portion of the subscribers' flow specifications within the stream. Second, stream data objects belonging to one stream are assumed to have many attributes in common as opposed to pub-sub systems, where message formats are assumed to be arbitrarily variable. Indeed, the stream format is an invariant and is used to establish circuits between producers and consumers.

4.1.3 PE Reuse

As we previously stated, another key design point is the support for the dynamic composition of applications. Because applications are built employing PE building blocks, it is possible to leverage code and expertise previously

Application writers have access to PE libraries, where fundamental operations such as relational operators, signal processing filters, translators, and others are available for mixing and matching by new applications. Reuse can be exercised when applications are designed and written or composed and is put to use when the application is deployed. At application dispatch time, DGM identifies commonalities between applications already in the system and the new application in terms of the PEs they employ. That is, the reuse mechanism is triggered every time a new application is overlaid on the global dataflow graph and its PEs are instantiated. The DGM creates a PE *signature* based on the PE configuration. Using the PE's signature the DGM attempts to find other candidate PEs in the system for reuse. Candidate PEs are then directly compared with the new PE descriptor. If *equivalence* can be established, the job submission client is instructed to not deploy the new PE. The DGM

instead simply increments the reference count for the reused PE and associates it with yet another job.

4.2 The Data Fabric

The Data Fabric is the data transport substrate in the SPC and is implemented by a collection of distributed servers (DFS), one per node in the cluster. It is responsible for routing, filtering, and flow-smoothing. It provides PEs with access to SDOs via ports and transports SDOs within streams.

Figure 5 depicts the Data Fabric internal architecture with multiple PEs placed on two nodes.

4.2.1 Data Routing

The DFS is responsible for determining the best routing strategy based on the topology of the network as well as the subscriptions to a stream. As seen in Section 3, PEs can have input ports through which they read SDOs and output ports to which they write SDOs. Each input port is associated with a flow specification, which has a stream-filter part and an SDO-filter part. The Data Fabric is responsible for setting up the circuits based on stream connection notifications from the DGM. Once connections are in place, it multiplexes data from various streams that match the stream-filter part into one input port. It then delivers SDOs that match the SDO-filter part of the flow specification to PE's input ports. The DF uses a subscription *Matcher* module for this task.

The filtering of SDOs based on the subscriptions of downstream consumers may be done either at producer or at each of the consumers of the data. In the current system implementation, the matching is done at the producer-end. However, the location of the matching process is flexible to help the DFS to adapt to available bandwidth and computation resources. For example, if all the subscriptions are evaluated at the source, the network bandwidth consumption potentially decreases, but the CPU cost for matching could be quite high.

4.2.2 SDO Transport

The DF chooses dynamically the most efficient transport for delivering SDOs from the same stream to different PEs, depending on factors such as topology, reliability levels, stream popularity, and number of consumers. Figure 5 shows examples of three transport layers available in the current prototype – (i) *Pointer Transport*: SDOs are transferred from the producing PE to the consumer PE by simply transmitting a memory pointer. This is suited for communication among PEs within the same container, where PEs share a common address space; (ii) *Shared Memory Transport*: SDOs are transported using shared memory segments when PEs run in different execution containers co-located in the same node; (iii) *Network Transport*: SDOs are transported from the producing PE on one node to the consuming PE on another node. This transport layer can be implemented on top of TCP, UDP, and IP-multicast. The former is available in the current prototype while the latter two are being implemented. When a connection between a producer and consumer needs to be reliable, we rely on the storage subsystem if necessary.

4.2.3 Traffic Isolation

As will be seen in Section 4.3, PEs run in execution containers, which provide process isolation. Since PEs run user-written code, their interaction with the Data Fabric needs to be mediated. Hence, the Data Fabric is split into two parts – the DFLocalRouter and the DFRemoteRouter. The DFLocalRouter is responsible for routing traffic among PEs that reside within the same PE execution container. The DFRemoteRouter is a standalone process responsible for routing traffic among PEs in different PE containers, for keeping the routing tables, and communicating with other system daemons such as the Dataflow Graph Manager.

4.2.4 Flow Balancing

Fluctuations and mismatches in SDO producing/consuming rates may result in data loss that can be prevented by performing local optimizations in buffer and CPU allocations. This is accomplished by the nano-scheduler component of the Data Fabric. The reader is referred to [3] for a complete description.

4.3 PE Execution Container

Every PE runs within the confines of a container called a PEC. The PEC provides isolation of PEs from each other and also monitors and manages the PE life cycle. The current level of containment is provided by a process address space. However, if necessary, a PE can be further isolated by running it, and its PEC, within a different virtual machine. Isolation may be required due to an explicit request since a PE may be running in debug mode for instance. If less containment is needed, for example, for PEs that are part of the same job, they can share the same container and be managed by the same PEC. Once the PE starts, the PEC monitors the PE CPU-usage and memory and reports them to the global resource manager (a complete description of resource management issues is outside the scope of this paper). In the current prototype, the PEC loads the PE dynamically from a library and runs each PE in its own thread.

5. PERFORMANCE EVALUATION

In this section, we demonstrate the SPC architecture by employing a large-scale synthetic application using the SPC prototype as well as a set of micro-benchmarks. Our studies were conducted on a cluster of 85 dual-processor hyper-threaded Intel Xeon 3.06 GHz nodes with 2 GB of RAM interconnected with switched Gigabit Ethernet as well as with a Storage Area Network (SAN), which is available to a subset of the nodes. The cluster runs SuSE Linux Enterprise Server Version 9. The aim of this study is to highlight some of the critical performance aspects as they relate to the overall middleware architecture. For an application-oriented experimental study based on the Linear Road benchmark [6], we refer the interested reader to [13].

5.1 Experimental Methodology

The experimental results are presented in two parts. In the first part, we were primarily concerned with extracting a performance profile of the control infrastructure between the system components that support the processing elements of a large scale application. The first objective was to characterize the costs associated with deploying and removing a large number of applications from the system. These operations involve primarily interactions between the submission client and the Dataflow Graph Manager using two remote procedure calls: instantiate_job and condemn_job. There are

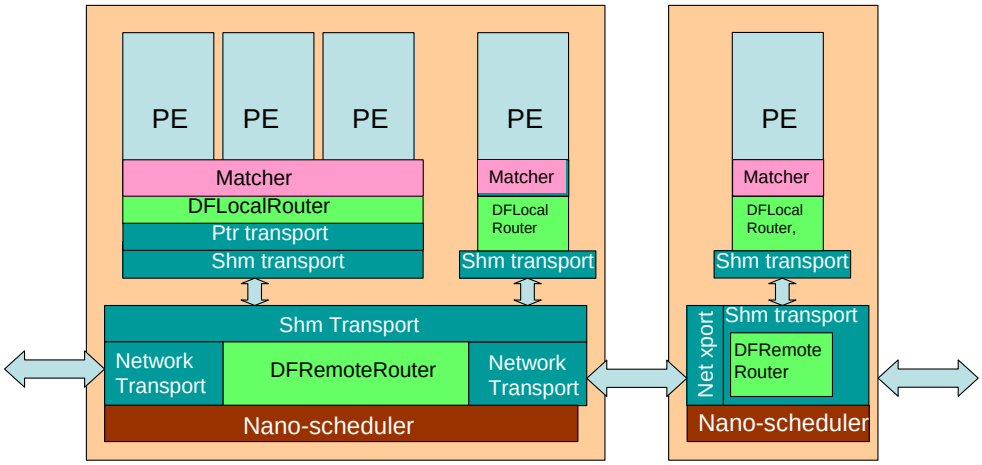


Figure 5: The Data Fabric internal architecture.

also remote calls from the Data Fabric servers on behalf of PEs connected – lookup_streams and lookup_subscriptions. These calls are used for establishing the initial stream connections between a new PE and the existing topology (both as a producer as well as a consumer). Secondly, we were interested in understanding how the process of setting up stream circuits is affected as new PEs join and leave the dataflow topology. These are notifications from the Dataflow Graph Manager to multiple Data Fabric Servers using these RPCs – new streams to input ports (new_stream_notif), new subscribers to output ports (new_subscriber_notif), removal of a stream (remove_stream_notif), and removal of a subscription (remove_subscription_notif).

All these interactions are affected by the number of PEs in the system, the job inter-arrival and inter-departure rate, the number of service threads in all the daemons, as well as the network traffic generated by the streams. Since the DGM is in the middle of these interactions, we chose to collect the following metrics: (1) average RPC response time – these are RPCs coming into the DGM from the job submission or DFS remote requests. The timings we report capture the amount of time spent inside the DGM including queuing, execution, and other overheads associated with the operation; (2) average processing time – these are RPCs made out from DGM to DFS daemons as part of the stream setup and tear-down. In this case the timings we report were collected inside DGM and reflect end-to-end communication costs between DGM and a DFS. When collecting the experimental data, we refrained from *sending* data through the application, since the goal was to obtain a performance profile for the middleware control infrastructure. The results we present are averaged across three runs, where two control knobs were employed: (1) the job inter-arrival and inter-departure time, characterized by an exponential distribution with a preset mean and (2) the number of DGM service threads available for asynchronous operations (typically for issuing and processing DFS RPCs). The experimental results were obtained while the cluster had only a modest amount of background workload.

The second part of our experimental study consists of micro-benchmarks that are relevant to PE-writers as they affect the performance of individual PEs. We measured (1) how much effective bandwidth is available to PEs as function of the size of the payload an SDO carries, (2) the latency an SDO incurs as it propagates through a chain of PEs, and (3) how efficient the Data Fabric's SDO filtering capability is.

5.2 A Synthetic Case Study Application

The first part of our experimental study employed a large-scale synthetic application we refer to as the Large Scale Demonstration (LSDemo, for short). LSDemo was conceived to demonstrate the various features of SPC – the capability to host a large set of jobs resulting in a large number of PEs and streams hosted on a large cluster of nodes. LSDemo also showcases the middleware ability to dynamically compose applications. That is, many of its jobs have connections among PEs that are loosely specified, i.e., the stream connections are made solely based on the compatibility of input and output port formats.

The challenge was not only to design a realistic large-scale stream processing scenario, but also to demonstrate the ingest and processing of an aggregate bandwidth of 10 Gb/s of raw data from various data sources. The LSDemo consists of over 700 PEs grouped into around 100 jobs, running on 85 nodes. LSDemo ingests content generated by a synthetic workload generator, consisting of Internet protocol packets (e.g., email, instant messaging chat applications, voice and video traffic, among others), de-multiplexes the traffic into specific protocols, extracts features such as IP addresses, subject names, email addresses and, eventually, correlates data across protocols for finding information nuggets such as messages from a particular sender or detecting buzzwords by sifting through the communication traffic. Additionally, some PEs in LSDemo are tasked with identifying information tidbits that can be used to influence the behavior of upstream PEs by, for example, restricting the data a particular PE ingests. Although the application employs PEs for all these operations, the operations themselves were not fully implemented. Instead, the results (or ground-truth) were encoded in the incoming data traffic generated by the workload generator. The actual operations carried out by

the PEs consisted merely of forwarding SDOs. The Data Fabric performed the content-based routing and filtered and forwarded SDOs according to rules encoded in the SDO-part of their flow specification expressions.

Figure 6 provides a bird's eye view of the complete topology (left-side), and two partial zoomed in views. The figure on the top right-side depicts a network with a source PE (responsible for acquiring data from a workload generator and pumping it into the data processing PEs) and a collection of de-multiplexer PEs responsible for separating categories of traffic (e.g., email from chat) and another layer of PEs that separates SDOs on a per protocol basis (e.g., SMTP from IMAP, ICQ from AIM, etc). The bottom left-side screenshot shows a closer view of PEs (represented as circles) and their input and output ports (represented as squares). The connections between ports are the streams. The popup message in that figure shows status information for a particular PE.

5.3 Large-scale Demo Results

The results depicted in this section were obtained by submitting the jobs for the LSDemo application using a single job submission client session, simulating an artificial bulk load/unload operation. An inter-arrival/inter-departure delay drawn from an exponential distribution was imposed between jobs (referred to as jiad in the charts). After a period of 30 seconds, the application was torn down by terminating jobs with a random inter-departure time drawn from the same exponential distribution. This configuration was chosen because it strenuously exercises most of the control paths in the middleware.

The results shown in Figure 7 consist of a parameter sweep where we varied the job inter-arrival and inter-departure rate as well as the number of threads that the DGM allocates to process asynchronous work (e.g., interactions with Data Fabric Servers).

Figure 7(a) illustrates the service time for calls issued by the job submission client (instantiate_job and condemn_job) as well as calls issued by Data Fabric daemons as they initially setup stream connections between processing elements (lookup_streams and lookup_subscriptions). The general trend is that job inter-arrival/inter-departure time impacts the service time, demonstrating as expected that, the more the requests-per-unit-time the DGM has to handle, the slower it becomes. Overall, the service time is below 2 ms, which translates in very fast bulk load/unload operations. Indeed, the dispatch of the 700+ PEs can be done in approximately 30 seconds and, during this time, the topology becomes stable (i.e., all stream connections are in place).

Figure 7(b) depicts the response time for calls issued from DGM to the Data Fabric Servers as new streams and subscribers become available (or leave the system) and new stream connections are established or removed between Data Fabric Servers on behalf of PEs. The results show that while the number of DGM service threads is a performance factor, for the LSDemo application no more than 4 threads are required, which means that at most 4 concurrent interactions happen between the DGM and several DFS daemons. In most cases, a slower pace of job submission/termination implies quicker response times from the Data Fabric as each daemon is facing fewer requests per unit of time. The fact that for removed_stream_notif and removed_subscriber_notif requests, the Data Fabric daemons take longer to service

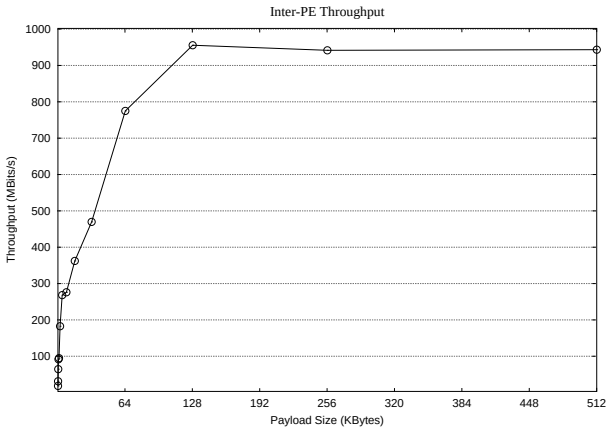


Figure 8: Effective bandwidth between two PEs on different nodes

the notifications as the inter-arrival and inter-departure time increases is interesting. This counter-intuitive behavior is explained by the underlying Corba RPC control communication implementation, which harvests connections that go unused for long periods of time, which means that, at teardown time (when these requests are issued), longer response times are observed. In other words, as the inter-arrival and inter-departure rate increases, the Corba connection harvesting threshold is reached and the Corba channels must eventually be restored in order to carry out the notifications.

In general, as seen for the LSDemo, the service and response times observed are considerably small, varying from a few hundred microseconds to a few thousand microseconds, which considerably aid in providing quick bulk load and unload capabilities. During normal operational conditions the service time is typically much smaller, because the control infrastructure is under much less stress as, among other issues, there is less contention for data structure locks.

5.4 Micro-benchmarks

In the second part of our study, the aim was in showcasing elements of the system design that affect application writers and users from the standpoint of individual PE performance.

One of the critical elements is the actual bandwidth PEs can effectively use. Figure 8 demonstrates that the bandwidth between two PEs running on two of our cluster nodes is a function of the size of the payload size carried by an SDO. The data was obtained by running 30-second measurement sessions where SDOs were sent as quickly as possible and the number of bytes per second for the serialized SDOs was averaged on the receiver side. For larger payloads (>128 KBytes), the maximum bandwidth is close to the rated maximum obtainable using a 1 Gbps Ethernet interconnection.

Application performance is also affected by the latency experienced by SDOs transported from the producing PE to the consumers. The results shown in Figure 9(a) indicate the variation in the latency incurred by an SDO as a factor of the number of PEs in series that it traverses. This experiment was performed with synthetic PEs running in three configurations, thereby evaluating the three different transport layers currently available in the Data Fabric. The first configuration uses Pointer Transport where the PEs are run-

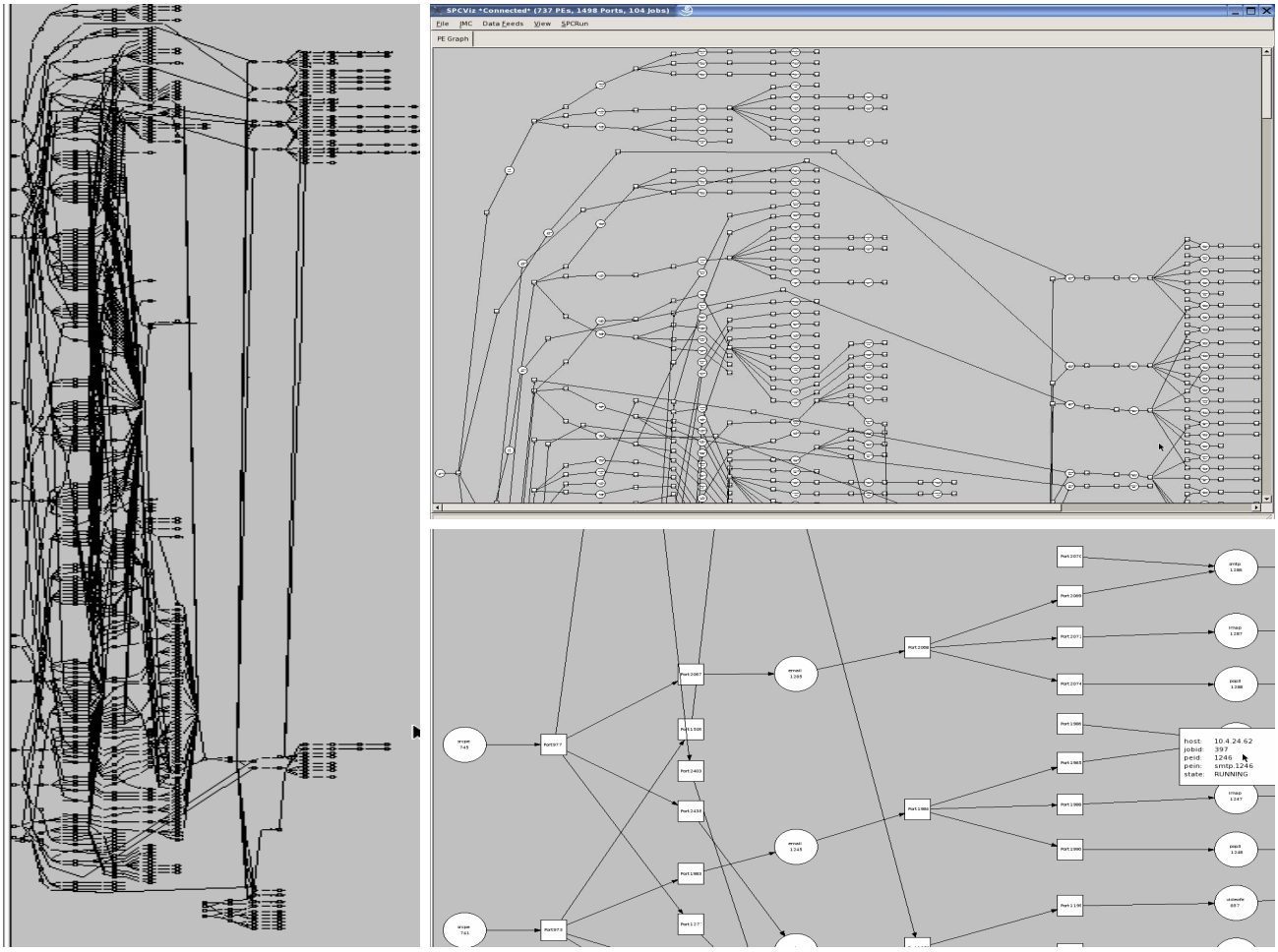
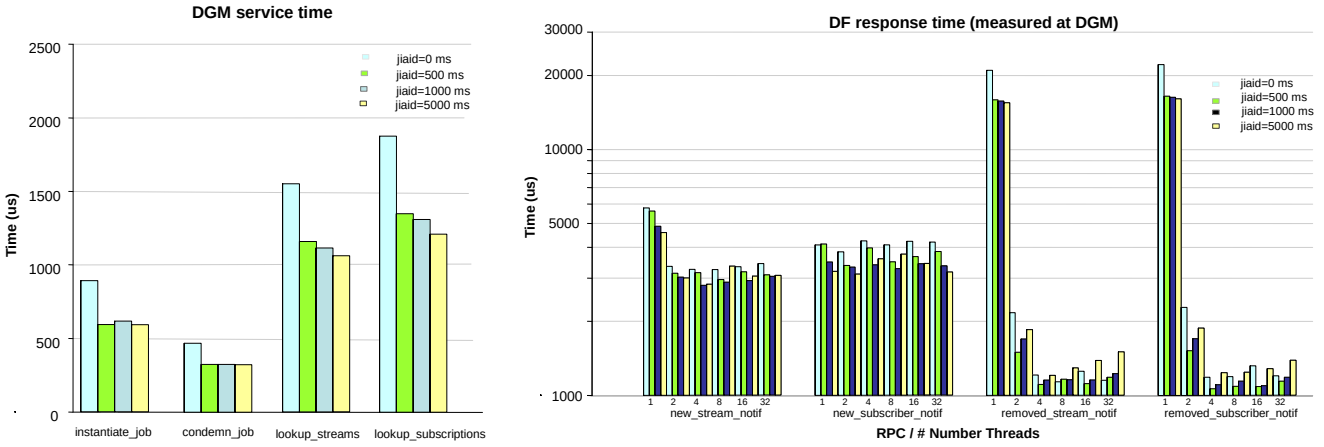


Figure 6: Large scale synthetic application

ning as threads in a single PE container and, hence, a simple pointer transfer is required to transmit the SDO. The second uses Shared Memory Transport where PEs run in independent PEC processes on the same node and we use a shared memory-based protocol to transfer the SDOs. Note that in our implementation, we ensure that there is only one writer per shared segment to provide lock-free access to memory. This results in different shared segments for PEs' input and output and consequently data copies from one to the other to transfer the SDO. In the third case (Network transport – TCP), each of the PEs runs in a different processing node. As expected, the latency increases with the above cases since the data transfer overheads increase. In the experiment, the transmission of 100 SDOs was paced over 5 seconds such that it avoided the situation where SDOs get *bunched* together in the DFS' outbound transmission queue. This allowed us to capture the latency of individual SDOs due to the transmission overhead, but not due to queuing. The latency was determined by measuring the time needed by an SDO to reach the final PE in the series.

Data filtering is one of the building block operations for streaming data applications. The micro-benchmark whose results are depicted by Figure 9(b) illustrates SPC's performance when a receiving PE defines a flow specification for one of its input ports. We employed a configuration where the producing and receiving PEs are located in the same node on different PECs, which required using the Shared Memory transport. In this experiment, we created SDOs whose structure consisted of 10 to 100 instances of the same attribute type. A flow specification expression consisting of a conjunction of clauses (to avoid a short-circuited evaluation) over individual attribute instances in the SDO was evaluated against each incoming SDO (100 SDOs sent over 5 seconds). When the number of clauses in the flow specification expression is more than the number of attributes, the flow specification is evaluated multiple times. The results show that expression evaluation is a linear function of the number of instances of an attribute an SDO carries as well as of the number of clauses in the expression.

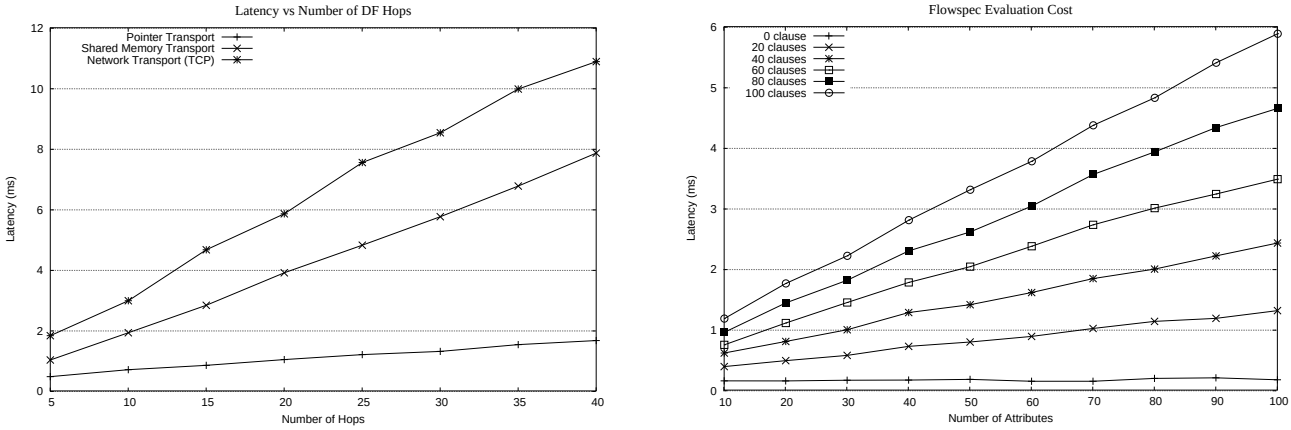
Our study has highlighted the middleware performance in a controlled setting, depicting many of the important aspects that are responsible for providing an effective environment to applications. We are now in the process of exposing additional control knobs and performance counters that will enable much more in-depth understanding and automatic control of performance issues in the face of multiple appli-



(a) Remote calls serviced by DGM

(b) Remote calls issued by DGM and serviced by DF daemons

Figure 7: Inter-component communication



(a) Per SDO latency overhead as SDOs propagate through a chain of PEs

(b) Per SDO flow specification evaluation costs

Figure 9: Data Fabric overheads

cations in large configurations. One such approach is the automatic adjustment of stream rates [3]. We believe that such an approach is essential for providing an autonomic computing environment of thousands of nodes hosting tens of thousands of PEs.

6. CONCLUSIONS AND DIRECTIONS

In this paper we presented the Stream Processing Core, middleware designed for supporting large-scale stream mining applications. Creating and tapping the knowledge and actionable information available from the huge number of raw data sources coming from the increasingly interconnected digital world requires massive processing capabilities. In this context, SPC was designed to support processing elements that will filter, analyze and, as a collective entity, mine these streams for information. We described the SPC programming model and its unique features such as the flow specifications that allow stream mining applications to tap into an ever-changing array of data streams available at run time. We also described the SPC architecture in detail and demonstrated that our still evolving prototype is able to efficiently support a non-trivial large scale application. Using micro benchmarks, we provided experimental evidence showing that the system has been engineered for providing high performance, both in the control infrastructure and in the data dissemination layer.

We are currently in the process of evaluating how architectural decisions such as stream-based routing as well as PE reuse impact real large-scale applications and aid in addressing the middleware scalability requirements. We are also implementing a replica-based, fault tolerant, Dataflow Graph Manager for greater scalability. In order to support larger volumes of data efficiently, we are implementing multicast-based transport in the Data Fabric. Finally, on the programming model side, the support of Java processing elements, additional and improved tooling for the design and implementation of processing elements coupled with GUI-based interfaces for application assembly are currently all underway.

Acknowledgments

SPC is part of a larger project being developed as a collaborative effort amongst many groups within IBM Research. The authors wish to thank Nagui Halim, the principal investigator, and several other project team members for many formative discussions.

7. REFERENCES

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), August 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2005 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, 2005.

[3] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, Lisbon, Portugal, July 2006.

[4] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, R. Motwani, U. Srivastava, and J. Widom. STREAM: The stanford data stream management system (demonstration description). *To appear in a book on data stream management edited by Garofalakis, Gehrke and Rastogi*, 2004.

[5] A. Arasu, B. Babcock, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM International Conference on Management of Data (SIGMOD 2003)*, San Diego, CA, June 2003.

[6] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases Conference (VLDB 2004)*, Toronto, Canada, 2004.

[7] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11), October 2001.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, , and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, 2003.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003.

[10] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, Philadelphia, PA, July 2002.

[11] T. Gotz and O. Suhre. Design and implementation of the UIMA common analysis system. *IBM Systems Journal*, 43(3), 2004.

[12] M. Hapner, R. Burridge, R. Sharma, and J. Fialli. Java message service – version 1.0.2b, August 2001. Sun Microsystems.

[13] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM International Conference on Management of Data (SIGMOD 2006)*, Chicago, IL, June 2006.

[14] R. Koster, A. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the 2001 ACM Multimedia Workshop on Multimedia Middleware*, Ottawa, Canada, October 2001.

[15] C.-Y. Lin, O. Verscheure, and L. Amini. Videodig project. http://www.research.ibm.com/VideoDIG.

[16] C.-Y. Lin, O. Verscheure, and L. Amini. Semantic routing and filtering for large-scale video streams monitoring. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME 2005)*, Amsterdam, Netherlands, July 2005.

[17] S. R. Madden, J. M. H. Mehul A. Shah, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM International Conference on Management of Data (SIGMOD 2002)*, Madison, WI, June 2002.

[18] C. Pu, K. Schwan, and J. Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.

[19] G. Swint, G. Jung, and C. Pu. Event-based QoS for a distributed continual query system. In *Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005)*, Las Vegas, NV, August 2005.