Event-based QoS for a Distributed Continual Query System

Galen S. Swint, Gueyoung Jung, Calton Pu, *Senior Member IEEE*
CERCS, Georgia Institute of Technology
801 Atlantic Drive,
Atlanta, GA 30332-0280
swintgs@acm.org, {helcyon1, calton}@cc.gatech.edu

Abstract

Continual queries for information flows are attracting increased interest. However, two important characteristics of information flows are not adequately addressed by current continual query packages: distribution and Quality-of-Service (QoS). In this paper, we use Infopipes, an abstraction for information flows, and the Infopipes Stub Generator to create a distributed version of the Linear Road benchmarking application around an existing continual query server (CQ server). Then, we install a QoS event generator/monitor pair that observes system response latencies. Receipt of each query result generates a feedback event reporting the latency (response time) for that request. If latencies exceed a threshold as parameterized by a WSLA, then the Infopipes controlling the CQ server can attempt to restore latencies to an acceptable level by generating an adaptation event for the Infopipes managing the CQ server.

1. Introduction

Continual streams of information are becoming increasingly common as more devices are attached to networks [1]. For instance, a car with a GPS receiver will emit a string of location data points, or an environment monitor creates a steady stream of temperature reports. As such devices and sensors become more common, more emphasis is being placed on immediate data analysis as well as managing and controlling systems of these information flows. Since the systems are filled with dynamic information, Quality-of-Service (QoS) ascends in importance because information value can rapidly decline over time [1]. Furthermore, to extract value from the "raw" data these streams contain, several research projects such as STREAMS [2], CACQ [3], Aurora [4], NiagaraCQ [5], and TelegraphCQ [6] have proposed Continual Query (CQ) systems that operate on data as it is produced rather than warehousing the data and performing queries off-line.

This work was partially supported by NSF/CISE IIS and CNS divisions through grants IDM-0242397 and ITR-0219902, DARPA ITO and IXO through contract F33615-00-C-3049 and cooperative agreement N66001-00-2-8901, and Hewlett-Packard.

In this paper, we demonstrate an information flow CQ system that implements end-to-end QoS through events. Some streaming database projects (e.g., Aurora and TelegraphCQ) have begun to consider QoS as it pertains to query execution, but do not address the systemic picture wherein QoS extends beyond the database query engine itself. We demonstrate system-level QoS achieved through code generation and weaving of a QoS event mechanism into an Infopipes-enabled system. Our demonstration is built around a streaming database benchmark and the Stanford Stream Data Manager (STREAM) [7] for executing the queries. The code generator lets us reuse communication stubs developed during the course of the Infosphere project [8]. Furthermore, the code generation and weaving techniques allows us to reuse the quality of service code and event processing engine in new applications with little to no alteration. (In fact, this work re-uses some quality of service components developed for media streaming demonstration scenario [9].)

The demonstration system comprises three components: a data source, which generates new data; the STREAM CQ engine, which filters data based on a query script; and the end application, the consumer of the query results. We use a dataset and query subset from the "Linear Road" benchmark that simulates a stream of location tuples from highway traffic [10]. The benchmark core revolves around continual queries that dynamically set toll rates (e.g., high tolls to combat rush hour congestion). The benchmark comprises a data set, historical queries, and continual queries. So far, two continual query projects have used Linear Road in evaluations of their systems, but only running on a single machine. In this paper, Infopipes wrap the STREAM CQ engine, and we distribute remainder the application by wrapping data sources and sinks with Infopipes.

This paper is organized as follows. First, we describe in section 2 Infopipes and the Infopipes Stub Generator (ISG) used to encapsulate and extend the Linear Road benchmark and the STREAM CQ server. Then in Section 3 we briefly outline the benchmark and STREAM. Next, in Section 4 we describe how we the wrapped the STREAM system and used Infopipes to distribute, add QoS events, and implement adaptive, end-to-end quality of service to the system. Then, in Section 5 we provide a short evaluation. Finally, we introduce related work and conclude in Sections 6 and 7 respectively.

2. Infopipes and the ISG

One of the hallmarks of information flow systems like Linear Road is their communication-intense operation. In a complete Linear Road simulation, for example, each of several thousand vehicles sends location and velocity data to the query processing system, which later returns data on toll charges back to the vehicles. Therefore, a large amount of executing code is dedicated to creating, maintaining connections, waiting for new data, and then marshalling and unmarshalling data from communications software. For these applications, communication is a vital challenge that is not efficiently solved by handcrafting code in point-to-point segments. In this section we introduce Infopipes, an abstraction for applications like Linear Road, the Infopipes Stub Generator (ISG), and the AXpect weaver, a module of the ISG that we use to insert QoS code into Infopipe systems.

Infopipes are the abstraction designed by the Infosphere project to address the needs of these information flow systems. Broadly speaking, an Infopipe is a set of inputs and outputs and a mapping or computation that links the two. Infopipes are also composable, that is, an Infopipe can be constructed by linking together smaller Infopipes in parallel or serial fashion. Each Infopipe has a typespec that includes a description of the inports and outports which exchange application-level packets of information. In compositions, of course, there is also a specification for what pipes should be connected. We devised a code generator, the Infopipes Stub Generator, which offers a reusable, high-level abstract interface for building and composing Infopipe systems [11].

A XIP (XML for Infopipes) specification describes three major concepts in Infopipes systems: application packet datatypes, simple Infopipes, and composed Infopipes.

A developer uses a datatype element to describe application-level packets of data that are the units of exchange between Infopipes. The code generator maps these into a native-format, like a `struct` or `class`.

Second, the user can specify a simple Infopipe. Each simple Infopipe encapsulates a computational element unit (e.g. a data source or a query server) and has one or more inports and one or more outports, and each of these ports is assigned a datatype as defined in some separate datatype specification. For example, in creating an Infopipe to wrap the data source of the Linear Road data, we create an Infopipe with one outport that accepts data from a TCP connection and feeds the STREAM server

Finally, these simple Infopipes compose into complex Infopipes. Each complex Infopipe declaration states any simple Infopipes it includes as well as their connections expressed as inport-outport pairings. Each simple Infopipe description can be used multiple times, as a simple Infopipe participating in a composition for a complex pipe can be given a local name. That is, we allow developers to separate declaration of an Infopipe from its definition. . Figure 1 is an excerpt from an Infopipe specification for a simple Infopipe and a composed Infopipe.

Code generation from XIP is a multi-stage process handled by the ISG. First, the ISG assembles an expanded XIP document, XIP+, by retrieving missing sub-specifications from a repository and resolving connection information. The repository is inherent to the design of the ISG and stores previously defined datatypes, Infopipes, and compositions as XML fragments on disk. That way, the repository encourages datatype and Infopipe reuse between applications and even between differing Infopipes within the same application. After creating a XIP+ document, the ISG uses XSLT templates to generate and embed source code with the specification in the XIP+. At the last stage, the XML in the XIP+ is stripped and the resulting files are written to disk.

Were this all the information a developer provided, the ISG could generate communication infrastructure for an Infopipes system. That code consists of bootstrapping and initialization routines for communication packages, marshalling, and de-marshalling routines for data. However, this basic functionality does not address quality of service. Since QoS is an issue that varies greatly between applications or even instances of a single application, we use the flexible AXpect weaver module of the ISG to address QoS implementation [9].

The AXpect weaver allows developers that use the ISG

```
<pipe class="StreamReceiver" lang="C">
  <apply-aspect name="sla_stream_loadshedder.xsl"
                doc="streams.xml">
    <apply-aspect name="stream_feedback_read.xsl"
           namepipeloc="/hc283/stream_proj/fb0"/>
    ... </apply-aspect>
  <apply-aspect name="stream_receiver.xsl"/>
  <ports>
    <inport name="streamRcvPort"
             type="InputStream"/>
  </ports>
</pipe>
<pipe class="LinearRoadTest">
  <subpipes>
    <subpipe name="streamSource"
              class="StreamSource"/>
    <subpipe name="streamReceiver"
              class="StreamReceiver"/>
    ...
  <connections>
    <connection comm="tcp">
      <from pipe="streamSource"
             port="streamOutputPort"/>
      <to pipe="streamReceiver"
           port="streamRcvPort"/>
    </connection>
    ...
</pipe>
```

Figure 1. Excerpt from a XIP file illustratating a simple Infopipe and Infopipe composition.

to extend the basic library of generated code by inserting, or weaving, new code that is not germane to communications. The weaver has three key components: descriptive tags in generated code, a set of XSLT templates with source code that recognizes the tags, and statements in XIP. We placed XML tags in the content of the XSLT generator itself to demarcate major units of generated code. During generation, the ISG writes integrates these tags with the output along with the source code resulting in "marked up" source that is encoded with Infopipes semantics. For instance, the ISG generates tags for the C code that performs marshalling as `<jpt:outport point="marshal">` and it tags a C header file for an Infopipe with `<jpt:header point="pipe">`. A QoS developer uses these to insert QoS relevant code by writing XSLT templates that match on the correct semantic tags and inject code before, after, or around the marked section. Finally, in the XIP document the developer adds a few directives for the AXpect module to run the proper XSLT templates (these are the `apply-aspect` elements in Figure 1). This new code becomes part of the generated system, and appears each time the stub generator runs for that specification. The aspects can be re-used across multiple Infopipes or multiple Infopipes applications.

One major advantage to the AXpect weaver is that since it is invoked before the Infopipe generator writes code to disk, it has a holistic view of the source for a system. This feature is especially important for problems like QoS because it lets a developer insert new functionality simultaneously even though the end code will end up geographically dispersed in distributed system.

Of course, while it is possible for the XSLT aspects to "hardcode" all information needed by the aspect, it is useful for them to be able to draw information or operating parameters from outside documents. Since AXpect aspects are written in XSLT, the weaver can retrieve information from XML documents outside the code generator. A WSLA (Web Service Level Agreement) specification is an XML document that allows us to externalize measurements, events, and triggers for web services [12]. Adopting a WSLA format, we can write more generic aspects that refer to the WSLA for QoS parameters.

3. Linear Road and STREAM

The Linear Road benchmark has been developed by researchers to test new stream query systems [10]. It is based on a dynamic toll scenario under consideration by highway agencies in several states to help combat congestion and encourage drivers, through higher tolls, to travel at off-peak times. In the benchmark, there are a series of multi-lane highways, and each highway is divided into segments. Cars on the highway provide regular position and velocity updates which are tracked by the query system. Then as a car approaches a highway segment, it is notified of its toll calculated based on observed traffic volume. In the real world, a driver can opts to accept the toll and continue on the highway or exit. If the driver accepts the toll, then the toll system debits an account. Ultimately, the benchmark's measure is how many "highways" a query system can support before responses a returned too slowly – i.e., response latency is violated.

It is easy to see that QoS is a natural and important requirement for Linear Road: the toll calculator must execute efficiently and quickly to inform drivers in time for them to make safe decisions about whether to continue on the toll road or, if the toll is too high, then exit at the next segment. Consequently, latency becomes an important end-to-end property that must be monitored closely – each position report must be answered within a few seconds with updated toll information. Once latencies become too long, drivers may attempt unsafe exits, or the toll authority may be forced into a default toll policy at the expense of lost revenue opportunities.

The complete Linear Road package has a set of Data Generator processes that create a stream of location data to simulate car locations on the toll road. Along with location data, the Linear Road also creates a set of queries over the data it produces, including Position Reports, Account Balances and Daily Expenditures, and Travel Time Estimation. For a full benchmark, the system must execute static queries and continual queries simultaneously. The generated data is fed through Data Drivers to the query system. By default, Linear Road produces location data that correspond to a location report for every 30 seconds of "real" time that passes in the simulation.

To execute the Linear Road queries, we use STREAM [7]. Running it requires a continual query (CQ) script (see Figure 2) and a second file of system configuration parameters. The CQ script contains three types of information. First, it contains a description of the input tuples as an association of field names and types. Second, it contains a file path location from which STREAM can read tuples. Lastly, the script contains continual queries described in the Continual Query Language (CQL) [13].

```
table : register stream CommonInputStr
         (Vid integer, Speed integer,
          XWay integer, Lane integer,
          Dir integer, Seg integer);
source : /hc283/stream_proj/source/db0

vquery : select Vid, XWay*1000+Dir*100+Seg,
           Speed from CommonInputStr;
vtable : register stream SegSpdStr
           (Vid integer, Seg integer,
            Speed integer);
...
query : select Vid, Seg, Lav,
          Toll from OutStr;
dest : /hc283/stream_proj/out/lrtest0
```

Figure 2. Excerpt of a STREAM script.

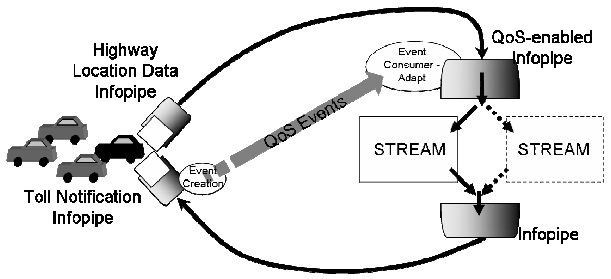

Figure 3. In the Linear Road benchmark, cars transmit location data to the STREAM CQ server. We add QoS feedback events from the cars to the Infopipes wrapping the STREAM servers.

CQL manipulates streams (a time-stamped and time-ordered sequence of tuples) and relations (time-varying sets of tuples) through various operators [4].

STREAM assumes the input tuples arrive in time-stamp order. Furthermore, STREAM is undergoing rapid development, and for this experiment a version was not available that implemented persistent storage. Therefore, we distill the benchmark to the essential continual query: calculating variable toll amounts on a volume basis. In this query, latency is incurred in internal buffers that STREAM automatically creates as part of the query plan. For this paper, we feed STREAM tuples in a serialized fashion from a generated simulation trace and then measure the latency as they are collected (see Figure 3).

As we mentioned earlier, STREAM also does not support distributed computation nor does it support QoS both of which are inherent to the Linear Road scenario. In the next section, we describe how we used Infopipes to add these features.

4. Distributed Linear Road

In this section, we describe how we used generated Infopipes and Quality of Service to create a distributed, adaptive version of Linear Road.

Our first task was to distribute Linear Road by wrapping each of the three application units (data source, STREAM server, and data sink) in an Infopipe. Currently, as did a prior Linear Road benchmark, we use a single data stream of highway information rather than creating a system of thousands of cars, each generating its own data. Instead, we feed the aggregated highway data through an Infopipe to a remote machine running the STREAM system, which then feeds the data out after query execution to an application sink, again via Infopipe. We wrap STREAM server with one input-only and one output-only Infopipe, streamReceiver (receiving from the car data source) and streamSender (sending back toll information to the cars), respectively. STREAM itself is hardcoded to read and write data only from files specified in the CQL. We worked around by embedding Unix pipe-

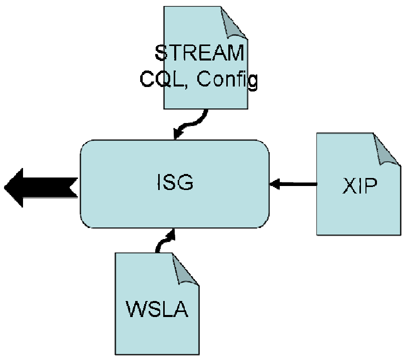

Figure 4. The ISG generator produces the Infopipes and QoS code that enables the distribution capabilities for the benchmark (left).

creation code in the wrapping Infopipes. By having Infopipes write and read on the Unix pipes, STREAM could be fed information from network sources.

An event-based model for controlling QoS is a natural fit since, generally, QoS is only relevant when some system event triggers quality evaluation. We program the QoS co-system as aspects to the Infopipes application, and use the AXpect weaver to integrate this code into the generation and deployment phase of the application. This achieves two goals. First, it allows logical, high-level separation of the QoS system. This type of abstraction enhances understandability. I.e., the developer can first create and debug a non-QoS system that has no QoS code to complicate the testing or performance profiling of the raw application. Second, abstracting the QoS system into a separate specification as standalone code offers the opportunity of re-use in later information flow systems. Furthermore, this method of packaging allows us to create a distributed QoS service which, in operation, is run distributed as three smaller pieces. In fact, a great deal of the QoS code we are applying to this problem was originally generated for an image streaming application.

If we run Linear Road with no QoS support, then it quickly violates the latency policy – the response with the toll amount is not returned quickly enough after a car sends its location. However, the latency is introduced primarily by the query engine and its buffers while the CPU remains relatively free. To detect this, we install a module on the data sinks that returns latency measurements. This behavior is defined within the WSLA. The receipt of a toll tuple at the DataSink that is late triggers a WSLA "Notification" event to be returned via the feedback channel.

If observed latency goes out of range, the monitor triggers an adaptation into a low-latency mode of operation. It does this by spawning a second copy of the query system, i.e., it spawns another STREAM instance. Based on highway numbers in the tuples, it splits the incoming tuple stream in two, and farms tuples from half the highways to the second server. This lowers latency because STREAM can use the second processor installed on the computer. Two servers means each query engine is handling fewer tuples; they can serve all tuples more quickly to reduce latency.

The AXpect weaver installs the QoS implementation on the base code – we insert QoS code automatically during code generation rather than after-the-fact manually. We wrote several small aspects, as it is easier to develop, deploy, and debug the QoS behavior by working with relatively small pieces of source code. The aspects were developed as follows:

1) The first aspect we develop measures latency of data streaming through the network. Since Linear Road tuples are generated with time stamps, the aspect need only observe the time stamps at the destination end of the application and calculate elapsed time.
2) Next, we introduce a feedback channel that returns timestamp data from the data sinks to the QoS monitor. A feedback event is generated every time that a data sink receives a data from the query server. For future flexibility, we relay the feedback information through both Infopipes connected to the query system. The feedback channel and event handler was reused, in fact, from an earlier project.
3) Next, on top of the control channel, we apply an aspect that reads the QoS specification from the WSLA and converts it into source code.
4) Inside the SLA code, we insert code that implements replication of the query server. In doing this, the aspect must create two things. First, a new Unix pipe must be created with a `mkfifo` call. Second, it emits new CQ script file that has been parameterized with the new FIFO name of the Unix pipe as the tuple source. For writing output tuples, the new query server can multiplex over the same FIFO as the original query server.
5) At this point, we write an aspect that distributes the tuples between the two query server copies. It does this by examining highway numbers and distributing half the highways to the replicated service, and half to the original service.
6) Next, avoidance for transient pricing errors. Since the new query server has no traffic information, it must be fed information for some time before tolls generated by the system are consistent with the original tolls. The aspect imposes a 60-second delay after starting to new STREAM server before accepting results from it.

5. Evaluation

We ran our evaluation of the distributed application on three PIII-800 dual processor machines. There was one machine for each of the system components (DataSource, QueryServers, and DataSink), and when we used two STREAM servers, they both resided on the same physical machine. We then evaluated the QoS-enabled system against a non-QoS-enabled system using the Linear Road benchmark for 1 to 32 highways.

First, we evaluated the system as a distributed system with no quality of service. We can see in the graph that in this case the latency grows slowly at first, but that at about 13 or 14 highways, latency rises dramatically indicating we have exceeded the optimal operating region for STREAM (see Figure 5). A second metric we use is "good" throughput. This is a measure throughput, but instead of counting all tuples processed, we only count those that arrive at the DataSink on time. Again, we see that throughput rises, but later falls as tuples become "late." In terms of good throughput, increasing amounts of input actually causes good throughput to fall (Figure 6).

After adding the QoS event monitor and adaptive code, the lower curve in the graph, we see how Linear Road is

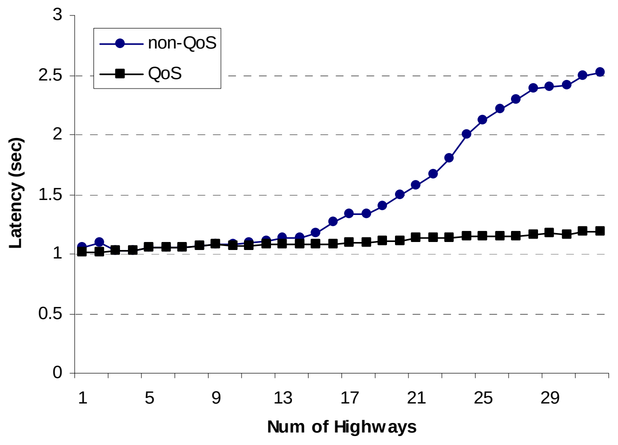

Figure 5. Average Latency, non-QoS and QoS. QoS-enabling keeps our latency under 1.5 seconds whereas with no QoS latency reaches 2.5 seconds.

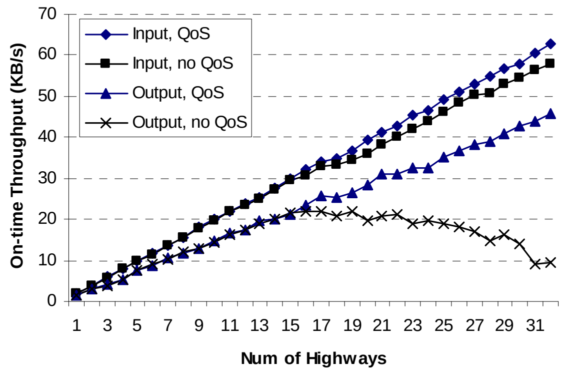

Figure 6. Throughput for input and "good" throughput for output. Higher input throughput under QoS results from more efficient servicing of buffers. Higher output throughput is from more tuples being "on time".

able to provide low latency information to the requesting cars. “Good” system throughput, that is throughput of tuples that are on-time, is dramatically improved, as the latency graph would indicate. Note, too, that for the input side throughput improves slightly because neither the system nor STREAM’s internal buffers become overwhelmed.

6. Related Work

As we mentioned, there are several continual query projects that are addressing QoS for their systems. However, none have yet demonstrated Quality of Service as it applies to the system beyond the query engine. Aurora [4] supports quality of service within the database engine itself, but it has not addressed quality of service as it applies to the broad system.

STREAM, upon which we built this example, is a lightweight query engine compared to Aurora since it does not add the Quality of Service adaptation nor does it require a heavy-duty ACID database to support it and provide persistent storage [2],[7].

The Berkeley TelegraphCQ project, like the Aurora project, has also addressed Quality of Service and even distribution or the query server itself [6], [14]. Still, it does not address the notion of Quality of Service outside the query engine as Infopipes allows us to do.

7. Conclusion

By wrapping the query system and application components in Infopipes, we were able to easily create a quality of service aware application. Using the Infopipes Stub Generator allowed us to create a distributed application with reusable communication stubs. By writing aspects for the AXpect weaver, we were able to introduce quality of service which not only reused QoS efforts code from earlier projects, but allowed us to concurrently generate the QoS code and communication stubs rather than insert the new QoS manually. Addressing QoS as a problem apart from application flow allowed us to use event-based servicing of latency violations rather than a continual information flow.

Future research will focus on more sophisticated QoS behavior, such as re-distributing information flows over multiple machines, more complete implementations of WSLA-reading code, or tunable queries.

8. References

[1] C. Pu, K. Schwan, and J. Walpole, “Infosphere Project: System Support for Information Flow Applications”, in ACM SIGMOD Record, Volume 30, Number 1, pp 25-34, (March 2001).

[2] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma,. “Query Processing, Resource Management, and Approximation in a Data Stream Management System”, In Proc. of the 2003 Conf. on Innovative Data Systems Research (CIDR), January 2003.

[3] S. Madden, M. Shah, J. M. Hellerstein, and Vijayshankar Raman, “Continuously Adaptive Continuous Queries over Streams”, SIGMOD 2002.

[4] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, “Monitoring Streams – A New Class of Data Management Applications”, VLDB 2002.

[5] J. Chen, D. DeWitt, F. Tian, and Y. Wang, “NiagaraCQ: A Scalable Continuous Query System for Internet Databases”, SIGMOD 2000.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, “TelegraphCQ: Continuous Dataflow Processing for an Uncertain World”. CIDR 2003.

[7] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, “STREAM: The Stanford Data Stream Management System”, *To appear in a book on data stream management edited by Garofalakis, Gehrke, and Rastogi*, 2004.

[8] P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, Calton Pu, “Infopipes: an Abstraction for Multimedia Streaming”, in *ACM Multimedia Systems Journal*, 8(5): pp 406-419, 2002.

[9] G. Swint, Pu, C, “Code Generation for WSLAs Using AXpect." Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004). San Diego, 2004.

[10] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker and R. Tibbetts, “Linear Road: A Stream Data Management Benchmark”, Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), August, 2004.

[11] G. Swint, C. Pu, and K. Moriyama. "Infopipes: Concepts and ISG Implementation." Proceedings of Second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (WSTFEUS'04). May 11 - 12, 2004. Vienna, Austria.

[12] A. Dan, D. Davis, R. Kearney, R. King, A. Keller, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, Web Services on demand: WSLA-driven Automated Management, IBM Systems Journal, Special Issue on Utility Computing, Volume 43, Number 1, pages 136-158, IBM Corporation, March, 2004.

[13] A. Arasu, S. Babu and J. Widom, “The CQL Continuous Query Language: Semantic Foundations and Query Execution”, Technical Report, Oct. 2003.

[14] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, “TelegraphCQ: An Architectural Status Report”. IEEE Data Engineering Bulletin, Vol 26(1), March 2003.