Distributed Stream Management using Utility-Driven Self-Adaptive Middleware

Vibhore Kumar, Brian F Cooper, Karsten Schwan
Center for Experimental Research in Computer Science
Georgia Institute of Technology, Atlanta, GA-30332
{vibhore, cooperb, schwan}@cc.gatech.edu

Abstract

We consider pervasive computing applications that process and aggregate data-streams emanating from highly distributed data sources to produce a stream of updates that have an implicit business-value. Middleware that enables such aggregation of data-streams must support scalable and efficient self-management to deal with changes in the operating conditions and should have an embedded business-sense. In this paper, we present a novel self-adaptation algorithm that has been designed to scale efficiently for thousands of streams and aims to maximize the overall business utility attained from running middleware-based applications. The outcome is that the middleware not only deals with changing network conditions or resource requirements, but also responds appropriately to changes in business policies. An important feature of the algorithm is a hierarchical node-partitioning scheme that decentralizes reconfiguration and suitably localizes its impact. Extensive simulation experiments and benchmarks attained with actual enterprise operational data corroborate this paper's claims.

1. Introduction

Many emerging business applications require real time processing and aggregation of updates produced by geographically distributed data sources. Examples range from internet-enabled sourcing and procurement tools like TraderBot [25], to those focused on enterprise-wide inventory management using RFID tags [24], to the operational information systems used by large corporations [9]. These applications, which aggregate, use and make sense of large volumes of business-data are now coping with a rapidly mounting problem: how to efficiently manage, configure and optimize a distributed information system that extends across the enterprise while maximizing business value (i.e., the priorities and policies set by the enterprise). The large number of system components, high rate and complexity of stream data, and large number of changing system conditions make the configuration problem too difficult for human system administrators.

We are developing self-adaptive middleware for distributed stream management that aims to 'autonomically' align automated system administration with the goals of the enterprise, to maximize business utility rather than basic metrics like bandwidth or latency. We focus on two aspects of self-management: self-configuration and self-optimization of the deployed information system. The approach first deploys a stream-composition framework for a particular application (represented as a data-flow graph), with little explicit human input. Then, as the system is processing information, middleware automatically reallocates resources and reconfigures the deployed data-flow graph to maximize business value despite the changing availability of bandwidth, variability in end-to-end delay, availability of CPU cycles, and changes in other system parameters. Although focused on self-management for a specific distributed stream management middleware, the concepts and the algorithms presented in this paper can be used in a wide-range of distributed systems to reconfigure and reallocate resources for the purpose of maximizing business value.

Our goal is to use business value, expressed as utility functions, to drive the management of distributed data stream processing systems. The approach uses in-network stream-aggregation coupled with self-managing autonomic techniques. Self-management is decentralized to ensure scalability and robustness. This exploits the fact that data in our target applications is usually routed using overlay networks, such that updates from distributed data sources arrive at their destination after traversing a number of intermediate overlay nodes. Each intermediate node contributes some of its cycles towards processing of the updates that it is forwarding. Thus, our approach attempts to deploy a data-flow graph as a network overlay over the enterprise nodes. A resulting advantage is the distribution of processing workload and a potential

reduction in communication overhead involved in transmitting data updates. The task of configuring and optimizing the deployed overlay is done by the middleware's autonomic module, which focuses on the goal of maximizing business value from these distributed resources.

We have chosen to investigate the concepts of self-management using a stream management middleware because it closely resembles many enterprise-wide applications, which involve real-time interaction (in particular, information-flow) and are inherently distributed in nature. In addition, our stream management middleware with a large number of underlying nodes and resources provides an excellent platform to validate autonomic concepts. Our current research is driven by Delta's operational information system, the deployment and reconfiguration phases for which are shown in Figure 1 and is described next.

1.1. Example: Operational Information System

An operational information system (OIS) [10] is a large-scale, distributed system that provides continuous support for a company or organization's daily operations. We have been studying one example of such a system: the OIS run by Delta Air Lines, which provides the company with up-to-date information about all of their flight operations, including crews, passengers and baggage. Delta's OIS combines three different types of functionality:

- *Continuous data capture* – for information like crew dispositions, passengers, airplanes and their current locations determined from FAA radar data.
- *Continuous status updates* – for low-end devices like airport flight displays, for the PCs used by gate agents, and even for large databases in which operational state

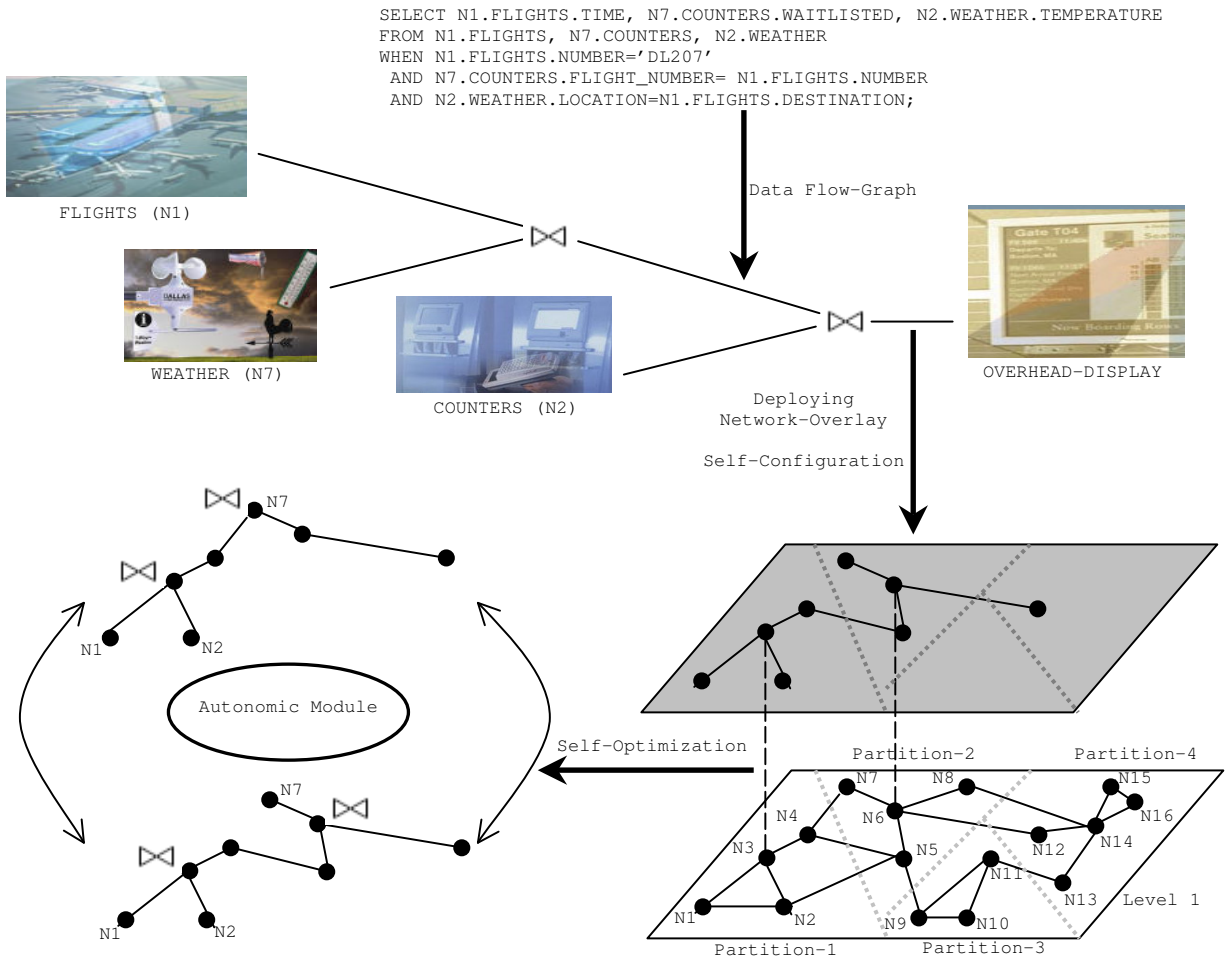

Figure 1. Steps involved in deploying the stream overlay

changes are recorded for logging purposes.

- *Responses to client requests* – an OIS not only captures data and updates/distributes operational state, but it must also respond to explicit client requests such as pulling up information regarding bookings of a particular passenger. Certain events may also generate additional state updates, such as changes in flights, crews or passengers.

The key challenges addressed by this paper are to simplify deployment and maintenance of continuous data-flows in the OIS. This is achieved by incorporating self-management into the system, thus relieving the administrator of tedious configuration and optimization problems. The graphs for various data flows in the system are expressed in high-level language constructs, and the middleware performs the tasks of initial deployment and maintaining optimality of the deployed flow-graph. The goal of deployment is to maximize business value, based on operating conditions and business policies (e.g., a Flow-Graph being accessed by a manager has higher business utility than the one being accessed by a regular employee).

1.2. Related Work

The self-adaptive distributed stream management middleware we have implemented is aligned with topics of interest to the autonomic community, to overlay management, and to those interested in middleware for large-scale distributed data management.

Autonomic Computing & Utility Functions

The tremendous increase in complexity of computing machinery across the globe and the resultant inability of administrators to deal with it, has initiated activities in academia and industry to make systems self-managing. A vision of autonomic computing as described in [1] is to design computing systems that can manage themselves given high-level objectives from administrators. Of the four aspects of self-management defined in the vision, we are focusing on self-configuration and self-optimization in stream management middleware. Our architectural approach is similar to earlier work in adaptive systems, captured for the autonomic domain in [2]. While the utility-driven self-optimization in our system is inspired by earlier work in the real-time and multimedia domains [29], the specific notions of utility used in this paper mirror the work presented in [3] which uses utility functions for autonomic data-centers. Utility functions have been extensively used in the fields of economics [5] and artificial intelligence [6]. Autonomic self-optimization according to business objectives has been studied in [4] although the distributed and heterogeneous nature of resources makes our infrastructure more general. Other work that aims to better align academic-optima with the business-optima include the Peakstone eAssurance product [11], which deals with non-violation of the SLAs and [12], which attempts to maximize the income of an eCommerce site.

Middleware & Stream Management

Pub-sub middleware like IBM's Gryphon [13], ECho [14], ARMADA [15], Hermes [16] are well established as messaging middleware. Such systems address issues that include determining who needs what data, building scalable messaging systems, and simplifying the development of messaging applications. We believe that our work is the necessary next step that utilizes middleware to provide high-level programming constructs to describe self-adaptive and 'useful' data-flows [31, 32].

Data-stream processing has recently been an important area for database researchers; groups like STREAM [17] at Stanford, Aurora [18] at Brown and MIT, and Infopipes [19] at Georgia Tech have been working to formalize and implement higher-level concepts for data-stream processing.

Task Allocation & Overlay Networks

Distribution and allocation of tasks has been a long studied topic in distributed environments. Architectural initiatives tailored for large-scale applications include SkyServer [8], enterprise management solutions [9] and grid computing efforts [20]. These applications perform task allocation to servers in much the same way as we recursively map operators to nodes. However, a high-level construct for describing the data-flow and run-time re-assignment of operators based on a business-driven utility distinguishes our infrastructure.

Overlay networks [21, 22] focus on addressing scalability and fault tolerance issues that arise in large-scale content dissemination. The intermediate routers in overlay network perform certain operations that can be viewed as in-network data-aggregation but are severely restricted in their functionality. The advantages of using our infrastructure are two-fold: first its ability to deploy operators at any node in the network, and second is the ease with which these operators can be expressed. There has also been some work on resource-aware overlays [23, 30], which is similar to resource-aware reconfiguration of the stream overlay in our infrastructure. In our case, reconfiguration is closely associated with business-level data requirements.

1.3. Roadmap

The remainder of this paper is organized as follows. In Section 2, we discuss the design and architecture of the distributed stream management middleware. Section 3 contains a brief description of the utility functions and how they facilitate the integration of business-policies with the system. Section 4 describes the data-flow graph

deployment and optimization problem in detail, followed by a brief description of the network hierarchy and its use in autonomic deployment and maintenance of the stream overlay. Section 5 presents an experimental evaluation of the proposed deployment and optimization algorithm, including results that were obtained using real enterprise data. Finally we conclude in Section 6 with a discussion of possible future directions.

2. System Overview

Our distributed stream management middleware is broadly composed of three components:

- The *Application Layer* is responsible for accepting and parsing the data composition requests and constructing the data-flow graph
- The *Autonomic Layer* adds self-management capabilities to the data-flow graph using the Pub-Sub ECho Middleware, the PDS resource-monitoring infrastructure and the specified business policies.
- The *Underlay Layer* organizes the nodes into hierarchical partitions that are used by the deployment and the optimization framework

The following subsections briefly describe these three layers.

2.1. Application Layer: Data Flow Parser

Data flows are specified with our data-flow specification language. It closely follows the semantics and syntax of the SQL database language. The general syntax of our language is specified as follows –

```
STREAM <attribute1> [<,attribute2> [<,attribute3> …]]
FROM <stream1> [<,stream2> [<,stream3> …]]
[WHEN <condition1> [<conjuction> <condition2>[…]]];
```

In the data-flow specification language, the attribute list mentioned after the STREAM clause describes which components of each update are to be selected. The stream list following the FROM clause identifies the data stream sources. Finally, predicates are specified using the WHEN clause. Each stream in the infrastructure is addressable using the syntax `source_name.stream_name`. Likewise, an attribute in the stream is addressable using `source_name.stream_name.attribute`. Our language supports in-line operations on the attributes, including `SUM`, `MAX`, `MIN`, `AVG` and `PRECISION`. Such operations are specified as `operator(attribute_list [,parameter_list])`. The system also provides the facility to add user-defined operators. The data-flow description is compiled to produce a data-flow graph. This graph consists of a set of operators to perform data transformations, as well as edges representing data streams between operators. The graph is then deployed in the network by assigning operators to network nodes.

2.2. Autonomic Layer: Utility & Optimization

The Autonomic Layer supports self-management in terms of self-configuration and self-optimization of the data-flow overlay. This support is provided by making use of ECho, which enables pub-sub event streaming; PDS, which enables monitoring of resources; and a Business Utility Calculation module, which acts as a decision making sub-system.

2.2.1. ECho: Publish-Subscribe Event Channels

The ECho [14] framework is a publish/subscribe middleware system developed by our group that uses channel-based subscription (similar to CORBA). ECho streams data over *channels*, which implement the edges between operators in the data-flow graph. The stream channels in our framework are not centralized; instead, they are lightweight distributed virtual entities managing data transmitted by middleware components at stream sources and sink. The traffic for individual channels is multiplexed over shared communication links by aggregating the traffic of multiple streams into a single stream linking the two communicating addresses.

We follow the semantics of a publish-subscribe system in order to ensure that multiple sinks can subscribe/unsubscribe to a stream channel depending on their requirements, and that the channels survive even when there are no subscribers (although in that case no actual data is streamed). A publish-subscribe system also proves useful when many sinks have similar data filtering needs; in such a scenario, a single channel derived using a data transformation operator can fulfill the needs of all the sinks.

The data-operator in our infrastructure is typically a snippet of code written in a portable subset of C called "E-Code." This snippet is transported as a string to the node where it has to be deployed. At the target-node, the code snippet is parsed, and native code is generated. The implicit context in which the code is executed is a function declaration of the form:

```
int operator( <input type> in, <output type> out)
```

A return value of 1 causes the update to be streamed, while a return value of 0 causes the update to be discarded. The function body may also modify the input before copying it to the output. New flow-graphs may use the streams from existing operators, may cause existing operators to be updated to stream additional relevant data, or may cause new operators to be created.

2.2.2. PDS: Proactive Directory Services

Network-wide resource availability information is managed by the Proactive Directory Service (PDS) [33]. This information allows us to dynamically reconfigure the data-flow deployment in response to changing resource

conditions. PDS is an efficient and scalable information repository with an interface that supports pushing resource events to interested clients. Through this interface, PDS clients can learn about objects (or types of objects) inserted in/removed from their environment and about changes to pre-existing objects. Our middleware uses PDS objects to receive resource updates from the system when operating conditions change.

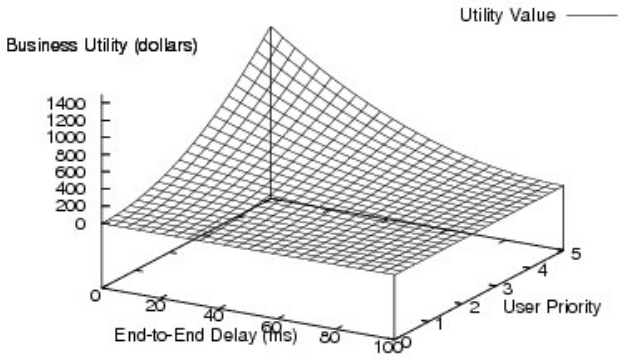

Figure 2. An example utility calculation model

2.2.3. Business-Utility Calculation Module

The business-utility calculation module contains a model of the system that can be used to calculate the net utility from individual attributes. In a distributed stream management system we identify end-to-end delay, bandwidth-usage, user-type, time-of-day etc. as attributes to calculate the business utility. An example utility calculation model is shown in Figure 2, which depicts a system where end-to-end delay and the user-priority determine the utility from the system. The high-level business policy in this scenario can be stated as "High Priority users are more important for the business" and "Less end-to-end delay is better for the business".

2.3. Underlay Layer: Network Partitioning

The underlay layer maintains a hierarchy of physical nodes, clustered according to system attributes such as end-to-end delay, bandwidth, etc. This clustering allows us to reduce the problem of calculating and maximizing utility across the whole network to the problem of managing utility for small, individual clusters. As a result, the system can only approximate the optimal overall business utility. However, it is too expensive to calculate, much less optimize, overall business utility for a complex, large-scale distributed enterprise information system. Our approach makes it feasible for the system to manage itself. In Section 5, we present experimental results, which show that our system is able to achieve close to the maximum business utility.

Each node in a cluster tracks the inter-node attributes (such as bandwidth) for the other nodes in the cluster. As a result, each node is able to calculate the business utility for changes involving other nodes in its cluster. A node is chosen from each cluster to act as the coordinator for this cluster in the next level of the hierarchy. Like the physical nodes in the first level of hierarchy, the coordinator nodes can also be clustered to add another level in the hierarchy. Similar to the initial level all the coordinators at a particular level know about probable high utility path to the other coordinator nodes that fall in the same partition at that level.

The advantage of organizing nodes in a hierarchy is that it simplifies maintenance of the clustering structure, and provides a simplified abstraction of the underlying network to the upper layers. Then, we can subdivide the data-flow graph to the individual clusters for further deployment. In order to scalably cluster nodes, we bound the amount of non-local information maintained by nodes by limiting the number of nodes that are allowed per cluster.

3. Incorporating Business Objectives

An implementation of business objective-driven optimization requires a model of the enterprise. This model is composed of the enterprise's environment, (which includes factors such as response-time, customer-priority, time-of-day, etc.) and a utility-function. The utility-function for an enterprise is a mathematical formulation that returns a measure of business-value given an environment state. As resource allocations and other environmental factors change during operation, the corresponding change in business worthiness of the system state can be calculated using this utility-function.

Our distributed stream management middleware is designed to support the operational information systems used in enterprises. We represent processing and aggregations of streaming information in an enterprise as a data-flow graph, and such flow graphs have an associated business-value for the enterprise. The business-value for a data-flow graph depends on factors like the priority of the user accessing the information, the time of day the information is being accessed, and other contributions that determine how critical the information is to the enterprise. The basic utility-function in our middleware can be defined as a combination of some of these factors, and data-flow graph specific factors such as the priority of the customer accessing the data can be added at run-time. Our aim is to deploy and manage several such data-flow graphs in order to maximize the net utility for the enterprise.

As an example, consider an information flow graph used to update the overhead displays. This flow-graph aggregates data from different data sources, which include weather stations, real-time flight information and the

check-in counters. A particular information flow gains priority over the other depending on the time remaining for flight departure, severe weather at destination, etc. Similarly, seating information pertinent to business-class passengers may arrive with high-priority as it reflects a higher business-gain than information about coach passengers.

4. Self-Configuration & Optimization

This section formally describes the data-flow graph deployment problem and presents a highly scalable distributed algorithm that can be used to obtain an efficient solution to this problem. Then, we extend the deployment algorithm by incorporating self-management.

4.1. Problem Statement

We consider the underlying network as a graph $N(V_n, E_n)$, where vertices $v_{ni} \in V_n$ represent the actual physical nodes and the network connections between the nodes are represented by the edges $e_{ni} \in E_n$. We further associate each edge e_{ni} with a delay d_{ni} and available-bandwidth b_{ni} that are derived from the corresponding network link. The data-flow graph derived from the SQL-like description is similarly represented as a graph $G(V_g, E_g)$. A data-flow graph is associated with a priority p that is a measure of its importance to the business. Each vertex V_g in G represents a source-node, a sink-node or an operator:

$$V_g = V_{g\text{-}sources} \cup V_{g\text{-}sink} \cup V_{g\text{-}operators}$$

$V_{g\text{-}sources}$ is the set of stream sources for a particular data-flow graph and each source has a static association with a vertex in graph N. Source vertices have an associated update-rate. $V_{g\text{-}sink}$ is the sink for the resulting stream of updates and it also has a static association with a vertex in graph N. $V_{g\text{-}operators}$ is the set of operators that can be dynamically associated with any vertex in graph N. Each operator vertex is characterized by a resolution factor, which represents the increase or decrease in the data flow caused by the operator. In general, join operators, which combine multiple streams, increase the amount of data-flow, while select operations, which filter data from a stream, result in a decrease. The edges in the data-flow graph are associated with a dynamic bandwidth requirement b_{gi}, which is calculated based on source stream rates and operator resolution factors. Each edge in the data-flow graph may span multiple intermediate edges and nodes in the underlying network graph.

We want to produce a mapping M, which assigns each $v_{gj} \in V_{g\text{-}operators}$ to a $v_{ni} \in V_n$. Thus, M implies a corresponding mapping of edges in G to edges in N, such that each edge $e_{gj\text{-}k}$ between operators v_{gj} and v_{gk} is mapped to the network edges along the highest utility path between the network nodes that v_{gj} and v_{gk} are assigned to. We define $utility(e_{gj\text{-}k})$ as the business-utility of the edge $e_{gj\text{-}k}$ and it is a function of the delay d_{ni}, the available-bandwidth b_{ni} of the intervening network edges e_{ni} and the required bandwidth $b_{gj\text{-}k}$ of the edge $e_{gj\text{-}k}$:

$$utility(e_{gj-k}) = f(\sum d_{ni}, \min(b_{ni}), b_{gj-k})$$
$$where \ \ i \mid e_{ni} \in M(e_{gj-k})$$

The net business-utility of data-flow graph G due to deployment M is calculated as:

$$utility(G, M) = p \times \sum_{e_{gj-k} \in E_g} utility(e_{gj-k})$$

For example, consider an edge in a system that uses utility function of the form:

$$utility \ \alpha \ (c - delay)^2 \times bandwidth_{available} \,/\, bandwidth_{required}\ .$$

(This formulation of utility is used for the experiments in Section 5.) If e_{gk} is determined by vertices v_{gi} and v_{gj}, which in turn are assigned to vertices v_{ni} and v_{nj} of the network graph N, then the utility corresponding to edge e_{gk} is calculated using required-bandwidth for edge e_{gk}; and the delay, available-bandwidth along the maximum utility path between the vertices v_{ni} and v_{nj}.

Since a stream management system usually handles multiple data-flow graphs therefore the problem is to construct a set of mappings, one per data flow graph, that maximizes the total system utility. This requires ordering the graphs for deployment such that graphs which promise to provide high utility are the first ones to be deployed; because these different graphs tradeoff resources, adding some resources to one graph means taking some away from another. At the level of an individual graph the problem is to construct the highest utility mapping M between the edges E_g in G to edges E_n in N.

4.2. Distributed Self-Configuration Algorithm

Now, we present a distributed algorithm for deploying the dataflow graph in the network. In a trivial scenario, we could have a central planner assign operators to network nodes, but this approach will not scale for large networks. Our partitioning-based approach deploys the data-flow graph in a more decentralized way. In particular, nodes in the network self-organize into a utility-aware set of clusters. This is achieved in middleware by having high inter-node utility for nodes in the same cluster (this may correspond to low latency and high available-bandwidth between the network nodes). Then, we can use this partitioned structure to deploy the data-flow graph in a utility-aware way, without having full knowledge of the network conditions and utility between all pairs of network nodes.

The result is that an efficient mapping M is constructed recursively, using the hierarchical structure of the underlay-layer. This mapping may not produce the optimal business-utility, since our approach trades

guaranteed optimality for scalable deployment. However, since deployment is utility-aware, the mapping should have high net-utility. Experiments presented in Section 5 demonstrate that our algorithm produces efficient deployments.

We now formalize the partitioning scheme described in Section 2.3. Let

n^i_{total} = total nodes at level i of the hierarchy

$n_{critical}$ = maximum number of nodes per partition

v^i_{nj} = coordinator node for node v_{nj} at level i

Note that $v^0_{nj} = v_{nj}$ and that all the participants of a partition know about high utility path to all other nodes in the same partition. We bound the amount of path information that each node has to keep by limiting the size of the cluster using $n_{critical}$. A certain level i in the hierarchy is partitioned when $n^i_{total} > n_{critical}$. We consider the physical nodes to be located at level 1 of the partition hierarchy and actual edge-utility values are used to partition nodes at this level. For any other level i in the hierarchy the approximate inter-partition utility values (derived using end-to-end delay, bandwidth, etc.) from level i-1 are used for partitioning the coordinator nodes from the level i-1. The approximate utility between any two vertices v_{nj} and v_{nk} at any level i in the hierarchy can be determined using the following equations:

$$utility^i(v_{nj}, v_{nk}) = \begin{Bmatrix} utility(v^{i-1}_{nj}, v^{i-1}_{nk}) & for\ v^{i-1}_{nj} \neq v^{i-1}_{nk} \\ 0 & for\ v^{i-1}_{nj} = v^{i-1}_{nk} \end{Bmatrix}$$

and

$$utility(v_{nj}, v_{nk}) = utility^l(v_{nj}, v_{nk}) \mid v^{l-1}_{nj} \neq v^{l-1}_{nk} \wedge v^l_{nj} = v^l_{nk}$$

In simple words, the utility at level i between any two vertices v_{nj} and v_{nk} of N is treated as 0 if the vertices have the same coordinator at level i-1, otherwise it is treated as equal to the utility at some level l where the vertices have the same coordinator and do not share the same coordinator at level l-1.

The distributed deployment algorithm works as follows. The given data-flow graph $G(V_g, E_g)$ is submitted as input to the top-level (say level t) coordinator. We construct a set of possible node assignments at level t by exhaustively mapping all the vertices $V_{g\text{-}operators}$ in V_g to the nodes at this level. The utility for each assignment is calculated using a simple graph traversal algorithm and the assignment with highest utility is chosen. This partitions the graph G into a number of sub-graphs each allocated to a node at level t and therefore to a corresponding cluster at level t-1. The sub-graphs are then again deployed in a similar manner at level t-1. This process continues till we reach level 1, which is the level at which all the physical nodes reside.

4.3. Self-Optimization

The initial deployment will have high business utility. However, when conditions change, utility can drop, and it may become necessary to reconfigure the deployment. The overlay reconfiguration process takes advantage of two important features of our infrastructure; (1) that the nodes reside in clusters and (2) that only intra-cluster maximum utility analysis is required. These features allow us to limit the reconfiguration to within the cluster boundaries for local fluctuations, which in turn makes reconfiguration a low-overhead process. An overlay can be reconfigured in response to a variety of events, which are reported to the first-level cluster-coordinators by the PDS. These events include change in network delays, change in available bandwidth, change in data-operator behavior (we call this operator profiling), available processing capability, etc. There are also some business specific events, which may trigger reconfiguration. One example is time of day, if the system has different observed traffic behaviors during different times of day. Consider region specific websites that provide local-news, weather, etc. These sites tend to have high traffic during morning, and the system goal is to maintain high-availability during this duration. Another example of a business even is the arrival of high-priority customers.

Reconfigurations can be expensive and temporarily disrupt the operation of the system. Therefore, it is impractical to respond to all such events reported by the PDS and the business. Instead, our system limits the number of reconfigurations it performs. We examine two approaches to limiting reconfigurations. One approach is to set certain thresholds that should be reached before a reconfiguration occurs. For example, a cluster-coordinator may recalculate the maximum utility paths and redeploy the assigned sub-graphs when more than half the edges in the cluster have reported change in utility value. Another example is to trigger a reconfiguration when the utility from a deployed graph falls by more than a threshold when compared with the best achievable utility. Another approach to limiting reconfigurations is to specify that a reconfiguration should only occur when certain constraints have been violated. For example, we may specify an upper bound on end-to-end delay, and trigger a reconfiguration when the total end-to-end delay violates this upper bound. The two approaches are examined in the following subsections.

4.3.1. Threshold Approach

In this approach, when we deploy a graph we specify the maximum tolerable negative deviation from the best achievable utility. Since in our system the graph is partitioned into sub-graphs, we can either aggregate deviations from the sub-coordinators to manage threshold at a central planner, or we can distribute smaller thresholds to sub-coordinators such that the sum of

smaller thresholds is equal to the overall threshold and each sub-coordinator manages the assigned threshold. We chose to implement the latter approach, and experimental results for this approach are presented in Section 5.

4.3.2. Constraint Violation Approach

This approach triggers reconfiguration when a constraint on a system parameter such as end-to-end delay or available-bandwidth is violated. The rationale behind this approach is that it may be easier to check for the violation of a constraint than to maintain a threshold against the optimal business utility. We again divide and distribute the constraint value to sub-coordinators, such that the constraint is not violated as long as all the sub-coordinators manage to satisfy their local constraint.

Note that reconfigurations in our middleware are not lossless: some updates and state maybe lost during the process. This is acceptable for most of the streaming applications, which are able to tolerate some level of approximation and loss. However, as part of the ongoing work we are examining how to model reconfiguration as a database-style transaction in order to achieve losslessness.

5. Experiments

The purpose of experiments is to evaluate the performance of our architecture. Microbenchmarks examine specific system features. An end-to-end setup for an application case study uses data emulating some elements of the Delta Air Lines' OIS. Results show that our system is effective at self-configuration and self-optimization of data-flow graphs for distributed processing of streaming data.

5.1. Experimental Setup

The GT-ITM internetwork topology generator [34] was used to generate a sample Internet topology for evaluating our self-configuration algorithm. This topology represents a distributed OIS scattered across several locations. Specifically, we use the transit-stub topology for the ns-2 simulation by including one transit domain that resembles the backbone Internet and four stub domains that connect to transit nodes via gateway nodes in the sub domains. Each stub domain has 32 nodes and the number of total transit nodes is 128. Links inside a stub domain are 100Mbps. Links connecting stub and transit domains, and links inside a transit domain are 622Mbps, resembling OC-12 lines. The traffic inside the topology was composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen [35]. The simulation was carried out for 1800 seconds and snapshots capturing end-to-end delay between directly connected nodes were taken every 5 seconds. These are then used as inputs for our distributed deployment algorithm.

5.2. Microbenchmarks

Our first experiment focused on comparing the business-utility of a deployed data-flow graph using the centralized model as opposed to the partitioning based approach used in our middleware. Since the centralized approach assumes that a single node tracks utility-determining system parameters for all other nodes, it produces the optimal deployment solution. However, the deployment time taken by the centralized approach increases exponentially with the number of nodes in the network. Figure 3 shows that although the partitioned-based approach is not optimal, the business-utility of the deployed flow graph is not much worse than the deployment in the centralized approach, and is thus suitable for most scenarios.

The next experiment is conducted to examine the effectiveness of dynamic reconfiguration in providing an efficient deployment. Figure 4 shows the variation of business-utility for a 10-node data-flow graph with changing network conditions, as simulated by introducing cross-traffic. The performance with reconfiguration is clearly better than without reconfiguration. Note that at

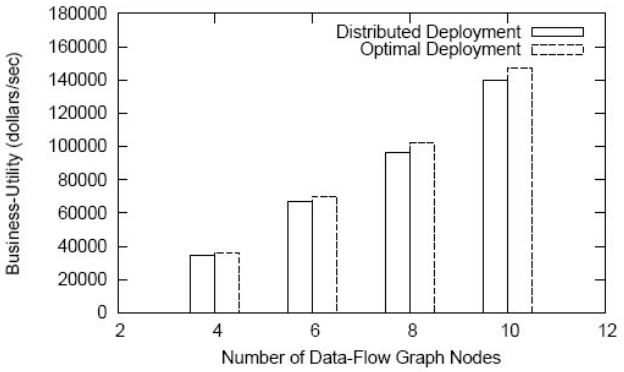

Figure 3. Comparison of utility achieved using Centralized versus Partitioning Approach

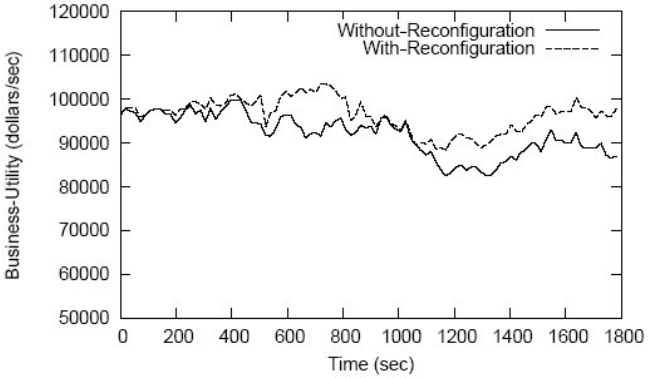

Figure 4. Utility variation for a data-flow graph with and without self-optimization

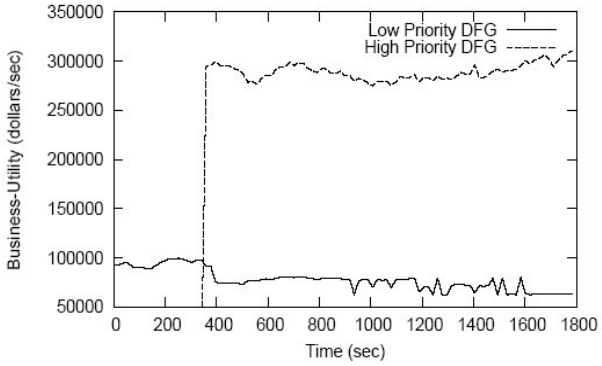

Figure 5. Effect of injecting high-priority flow graph

some points, the utility of the flow-graph with reconfiguration becomes less than that of flow-graph without reconfiguration. This happens because the hierarchical utility calculation algorithm used in our approach calculates the graph cost that is an approximation of the actual business utility. In some cases the approximation is inaccurate, causing the reconfiguration to make a poor choice. However, these instances are rare, and when they do occur, the utility of the flow-graph with reconfiguration is not much worse than the one without reconfiguration. Moreover, for most of the time reconfiguration produces a higher utility deployment.

5.3. Injecting Data-Flows & Self-Optimization

Next, we present a set of experiments that study the behavior of our stream management middleware in the presence of more than one data-flow graph. First, we conduct an experiment to study the effect of deploying a high priority data-flow graph in a system with an existing deployed low priority data-flow graph. A 10-node data-flow graph with priority=1 is introduced at time t=0.0 sec and then at t=360.0 sec another 10-node data-flow graph with priority=3 is injected into the system. Figure 5 shows the resulting business utility over time. We see a noticeable decrease in the utility for the low-priority graph when the new flow-graph is injected. This is because the high priority graph is preferentially assigned system resources. Thus, the bandwidth and the minimum delay paths previously available to the low-priority graph may get assigned to the high-priority graph thereby reducing its net business utility.

We also conduct experiments to evaluate the performance of our two self-optimization approaches: the delta threshold approach, and the constraint violation approach. The change in business-utility of a 10-node data flow graph using the delta-threshold approach in the presence of network perturbations (the traffic was composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen) is shown in Figure 6. We notice that even for sufficiently large value of threshold the achieved utility closely follows the maximum achievable utility. We also calculate the corresponding capital loss incurred due to sub-optimal deployment of the flow-graph using different thresholds. The results are shown in Table 1. The loss is calculated as the integral over time of the difference between the maximum achievable utility and the current utility for the deployed flow graph. The loss incurred increases exponentially as the threshold is increased. However, it is sufficiently low for a large number of values, and thus an appropriate threshold value can be used to trade-off utility for a lower number of reconfigurations.

Figure 7 shows the variation of business utility when the constraint-violation approach is used for self-optimization. In this experiment, we place an upper bound on the total end-to-end delay for the deployed data-flow graph, and triggered a reconfiguration when this bound was violated. This experiment is driven by real world requirement for delaying reconfiguration until a constraint is violated; because in some scenarios it might be more important to maintain the configuration and satisfy minimal constraints rather than optimize for maximum utility. We can notice some resemblance in behavior between the delta-threshold approach and the constraint

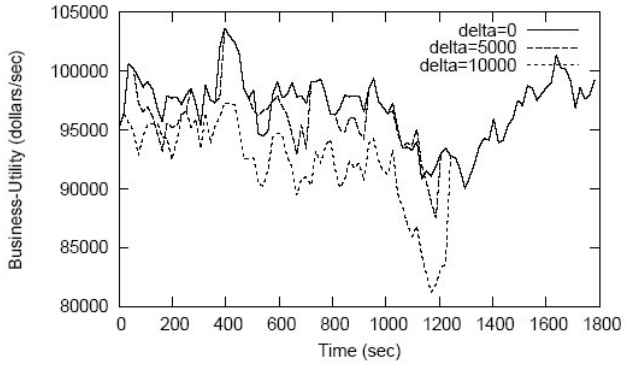

Figure 6. Utility variation using delta-threshold approach in presence of network perturbation

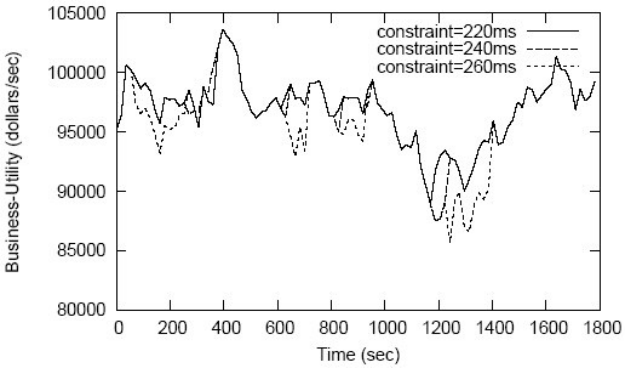

Figure 7. Utility variation using constraint violation approach in presence of network perturbation

Table 1. Loss & Reconfigurations using Delta-Threshold Approach

Delta-Threshold (dollars/sec)	Reconfigurations	Capital-Loss (dollars)
0	11	0
2500	7	9881
5000	6	250326
7500	4	876471
10000	1	5728104

Table 2. Loss & Reconfigurations due to Constraint Violation Approach

Delay-Constraint (msec)	Reconfigurations	Capital-Loss (dollars)
220	10	0
230	7	112987
240	6	292509
250	6	976588
260	6	1860597

violation approach. This is because utility is a function of end-to-end delay for the deployed flow graph. However, managing the system by monitoring constraint violation is far easier than optimizing a general utility function. Self-optimization that is driven by change in utility value is more difficult than the one driven by constraint violation because calculating maximum achievable utility requires knowledge of several system parameters and the deployment ordering amongst various deployed graphs for achieving maximum system utility. The corresponding capital loss due to different constraint values and the number of reconfigurations are shown in Table 2.

5.4. Policy driven Self-Optimization

Next, we conduct an experiment to study the responsiveness of our middleware to changes in the business policy. The priority of a data-flow graph in our system is determined by the maximum user-priority amongst all users accessing that data-flow. We initiate 3 different 10-node data-flow graphs and simulate a number of users with 5 different priority levels in the presence of network perturbation. The priority of the data-flow graphs is driven by the user accessing the data and changed as the users arrived and departed. The change in business-utility corresponding to fluctuating priority is shown in Figure 8. There is an almost instantaneous change in utility value when the priority of a particular graph changes. Another interesting result shown in the figure is the corresponding change in end-to-end delay for the graph as the priority of flow graphs change. We notice a decrease in end-to-end delay for data-flows when there is a corresponding increase in graph priority. This result is interesting because the system autonomically assigns lower end-to-end delay paths to flow-graphs with higher priority by making use of the utility function.

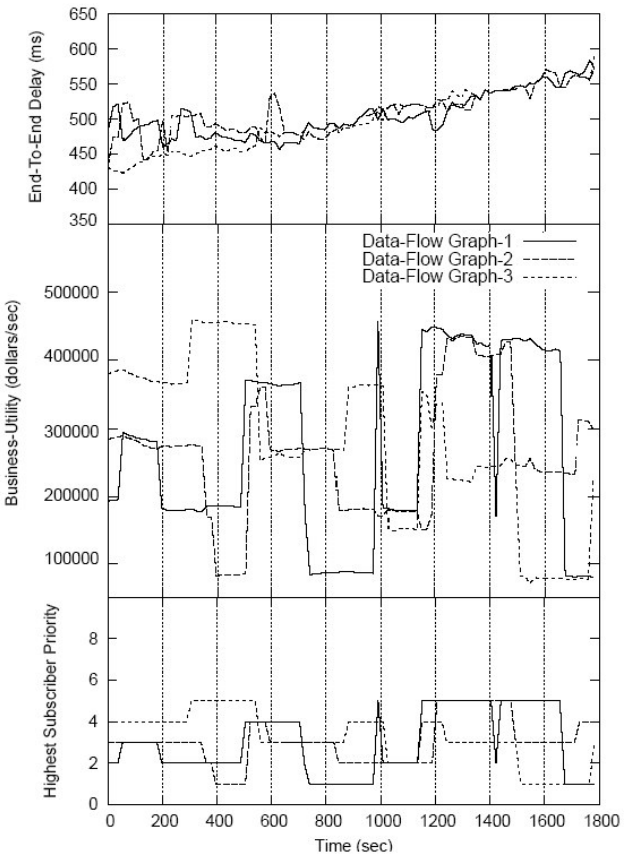

Figure 8. Effect of policy driven self-optimization

5.5. Application Case Study

The next set of experiments is conducted on Emulab [26] use data emulating an airline OIS combined with simulated streams for Weather and News. The experiment is designed to emulate 4 different airport locations. The inter-location delays are set to ~10ms while delays within an airport location were set to ~2ms. The emulation is conducted with 13 nodes (Pentium-III, 850Mhz, 512MB RAM, RedHat Linux 7.1) and each location has only limited nodes connected to external locations. These experiments are motivated by the requirement to feed overhead displays at airports with up-to-date information. The overhead displays periodically update the weather and news at the 'destination' location and switch over to seating information for the aircraft at the boarding gate. Other information displayed on such monitors includes the names of wait-listed passengers, the current status of flights, etc. We deploy a flow graph with two operators, one for combining the weather and news information at destination and the other for selecting the appropriate flight data, which originates from a central location (e.g., Delta's TPF facility).

The first experiment conducted on Emulab studies the self-optimizing behavior of our system in the presence of

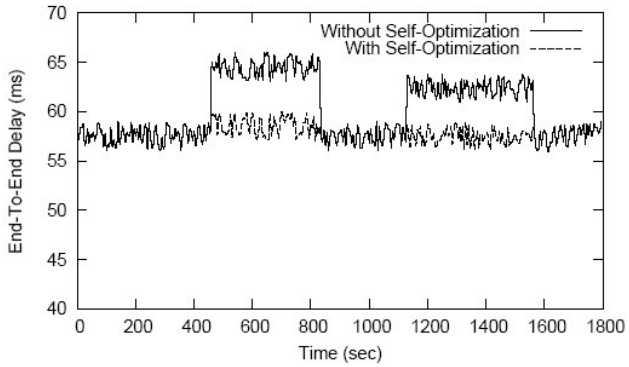

Figure 9. End-to-End delay variation with and without self-optimization for traffic spike and processor-overload

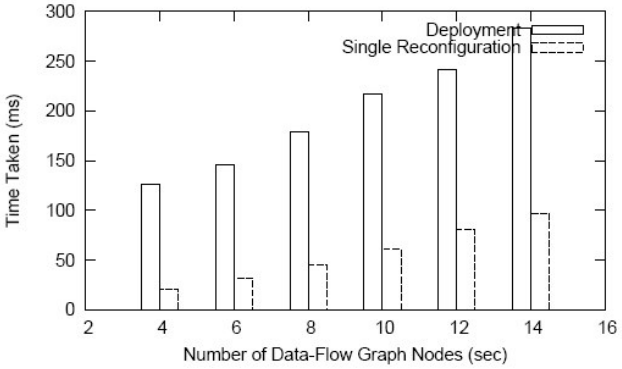

Figure 10. Comparison of time-taken for deployment and reconfiguration on the emulab nodes

network perturbation, and we also study the system's response to processor overload. A simple utility function of the form $utility = (c - delay)^2 \times CPU_{available} / CPU_{required}$ is used to obtain these results; thus, the system tries to minimize end-to-end delay when optimizing. Once the data flow graph for providing an overhead display feed is deployed, we use iperf [27] to introduce traffic in some of the links used by the flow-graph. This is represented by the first spike in Figure 9. With self-optimization the flow-graph responds well to the spike in traffic; in contrast, the statically deployed graph experiences an increased delay. The next spike is a result of an increased processing load at both the operator nodes. Again with self-optimization we end with a better delay than the static deployment. Even with self-optimization the end-to-end delay spikes, but the time before the deployment adjusts is so short (milliseconds) that the spike is effectively unnoticeable.

The next experiment examines the times for initial deployment and reconfiguration. Figure 10 shows that the times are quite small; only a few hundred milliseconds in the worst case. The figure illustrates the advantage of using a pub-sub middleware for deploying the flow graph. The pub-sub channels have to be created only at the time of deployment; reconfiguration just involves a change in publisher and subscriber to this channel and is therefore even faster. It may also be noted that once the channels for the data-flow graph have been created, deployment is essentially a distributed process, which starts once the corresponding nodes receive the operator deployment messages. This makes deployment time to increase almost linearly with the number of nodes.

Table 3. Middleware: Send & Receive Costs

Msg Size (bytes)	Send Cost (ms)	Receive Cost (ms)
125	0.084	0.154
1250	0.090	0.194
12500	0.124	0.327

5.5.1. Middleware Microbenchmarks on Emulab

Table 3 gives a measure of the low send and receive overheads imposed by the middleware layer at the intermediate nodes using the above setup. Send-side cost is the time between a source submitting data for transmission to the time at which the infrastructure invokes the underlying network 'send()' operation. Receive side costs represent the time between the end of the 'receive()' operation and the point at which the intermediate operator or the sink receives the data. Additional performance measurements comparing middleware performance to that of other high performance communication infrastructures appear in [14].

6. Conclusions & Future Work

This paper presents a scalable approach to the deployment of large-scale, distributed applications to underlying networked computing platforms, and to changing deployments when platform conditions or business needs change. The approach is novel in its explicit consideration of business value during deployment and when configuration changes occur. It is scalable in its use of a distributed algorithm for deployment and configuration management. It is dynamic in its use of runtime monitoring information about current platform conditions and its ability to deal with changes in the current business value attained from the distributed application in question. The approach is designed to be integrated into the middleware used by large-scale distributed applications, thereby adding self-management abilities to such middleware. This paper's implementation uses publish/subscribe middleware focused on stream-based business applications, an example being the operational information systems in wide use in industry. Examples are drawn from an OIS used in the airline industry.

Our future work will include experiments with wide-area infrastructures and platforms, focused on better understanding how to maintain business utility in

complex execution and application environments. Topics of interest include the development of new techniques for isolating important data streams and stream processing actions from less important ones and of preferentially protecting important actions from congestion in the underlying platform.

Acknowledgements

The authors would like to thank the Emulab community for providing the infrastructure to conduct the experiments reported in this paper. The authors would also like to thank Delta Technologies [28] for inspiring suitable use-cases for experiments.

References

[1] J O Kephart, D M Chess. The Vision of Autonomic Computing, IEEE Computer 36(1): 41-50, 2003.

[2] S R White, J E Hanson, I Whalley, D M Chess, J O Kephart. An architectural approach to autonomic computing. Proceedings of the International Conference on Autonomic Computing, ICAC-2004, New-York, USA.

[3] W E Walsh, G Tesauro, J O Kephart, R Das. Uitlity Functions in Autonomic Systems. Proceedings of the International Conference on Autonomic Computing, ICAC-2004, New-York, USA.

[4] S Aiber, D Gilat, A Landau, N Razinkov, A Sela, S Wasserkrug. Autonomic Self-Optimization according to Business Objectives. Proceedings of the International Conference on Autonomic Computing, ICAC-2004, New-York, USA.

[5] A Mas-Colell, M D Whinston, J R Green. Microeconomic Theory. Oxford University Press, 1995.

[6] S Russel, P Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, second edition, 2003.

[7] Samuel R. Madden and Michael J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. ICDE Conference, February, 2002, San Jose.

[8] Alexander S. Szalay and Jim Gray. Virtual Observatory: The World Wide Telescope (MS-TR-2001-77). General audience piece for Science Magazine, V.293 pp. 2037-2038. Sept 2001.

[9] A. Gavrilovska, K. Schwan, and V. Oleson. A Practical Approach for `Zero' Downtime in an Operational Information System. International Conference on Distributed Computing Systems (ICDCS-2002), July 2002, Austria.

[10] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu and D. Amin. Operation Information System – An Example from the Airline Industry. First Workshop on Industrial Experiences with Systems Software WEISS 2000, October, 2000.

[11] http://www.peakstone.com

[12] D Manasce, J Almeida, R Fonseca, M Mendes. Business-Oriented Resource Management Policies for eCommerce servers. Performance Evaluation, 2000.

[13] http://www.research.ibm.com/gryphon/

[14] G. Eisenhauer, F. Bustamente and K. Schwan. Event Services for High Performance Computing. Proceedings of High Performance Distributed Computing (HPDC-2000).

[15] T. Abdelzaher, et al. ARMADA Middleware and Communication Services, Real-Time Systems Journal, vol. 16, pp. 127-53, May 99.

[16] Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02), pages 611-618, Vienna, Austria, July 2002.

[17] S Babu, J Widom (2001) Continuous Queries over Data Streams. SIGMOD Record 30(3):109-120

[18] D Carney, U Cetintemel, M Cherniack, C Convey, S Lee, G Seidman, M Stonebraker, N Tatbul, S Zdonik. Monitoring Streams: A new class of data management applications. In proceesings of the twenty seventh International Conference on Very Large Databases, Hong Kong, August 2002.

[19] R. Koster, A. Black, J. Huang, J. Walpole, C. Pu. Infopipes for composing distributed information flows. Proceedings of the 2001 International Workshop on Multimedia Middleware. Ontario, Canada, 2001.

[20] W Allcock, J Bester, J Bresnahan, A Chervenak, I Foster, C Kesselman, S Meder, V Nefedova, D Quesnel, S Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. Parallel Computing, 28 (5). 749-771. 2002.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the ACM SIGCOMM ’01 Conference. ACM Press, 2001.

[22] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. IEEE J-SAC, Jan 2004.

[23] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, John Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. Proceedings of ACM NOSSDAV 2001.

[24] Wal-Mart backs RFID Technology. http://www.computerworld.com/softwaretopics/erp/story/0,10801,82155,00.html

[25] http://www.traderbot.com

[26] http://www.emulab.net

[27] http://dast.nlanr.net/Projects/Iperf/

[28] http://www.deltadt.com

[29] R. Kravets, K. L. Calvert, and K. Schwan. Payoff Adaptation of Communication for Distributed Interactive Applications. Journal on High Speed Networking: Special Issue on Multimedia Communications, 1998.

[30] Y. Chen, K. Schwan and D. Zhou. Opportunistic Channels: Mobility-aware Event Delivery. Proceedings of the 4th ACM/USENIX International Middleware Conference(Middleware 2003), June 2003.

[31] M. Wolf, Z. Cai, W. Huang, K. Schwan. Smart Pointers: Personalized Scientific Data Portals in Your Hand. Supercomputing 2002, ACM, November 2002.

[32] Z. Cai, G. Eisenhauer, C. Poellabauer, K. Schwan, M. Wolf. IQ-Services: Resource-Aware Middleware for Heterogeneous Applications". IPDPS/HCW 2004, Santa Fe, NM, April 2004.

[33] F. Bustamante, P. Widener, K. Schwan. Scalable Directory Services Using Proactivity. Proceedings of Supercomputing 2002, Baltimore, Maryland.

[34] E. Zegura, K. Calvert and S. Bhattacharjee. How to Model an Internetwork. Proceedings of IEEE Infocom '96, San Francisco, CA.

[35] http://www.isi.edu/nsnam/ns/