Resource-Aware Distributed Stream Management using Dynamic Overlays

Vibhore Kumar, Brian F Cooper, Zhongtang Cai, Greg Eisenhauer, Karsten Schwan
College of Computing, Georgia Institute of Technology
801 Atlantic Drive, Atlanta, GA 30332-0280
{vibhore, cooperb, ztcai, eisen, schwan}@cc.gatech.edu

Abstract

We consider distributed applications that continuously stream data across the network, where data needs to be aggregated and processed to produce a 'useful' stream of updates. Centralized approaches to performing data aggregation suffer from high communication overheads, lack of scalability, and unpredictably high processing workloads at central servers. This paper describes a scalable and efficient solution to distributed stream management based on (1) resource-awareness, which is middleware-level knowledge of underlying network and processing resources, (2) overlay-based in-network data aggregation, and (3) high-level programming constructs to describe data-flow graphs for composing useful streams. Technical contributions include a novel algorithm based on resource-aware network partitioning to support dynamic deployment of data-flow graph components across the network, where efficiency of the deployed overlay is maintained by making use of partition-level resource-awareness. Contributions also include efficient middleware-based support for component deployment, utilizing runtime code generation rather than interpretation techniques, thereby addressing both high performance and resource-constrained applications. Finally, simulation experiments and benchmarks attained with actual operational data corroborate this paper's claims.

1. Introduction

Many emerging distributed applications must cope with a critical issue: how to efficiently aggregate, use, and make sense of the enormous amounts of data that is generated by these applications. Examples include sensor systems [1], distributed scientific processes like SkyServer [2], operational information systems used by large corporations [3], and others. Middleware initiatives for such applications include publish/subscribe systems like IBM's Gryphon [4] project or related academic endeavors [5], or commercial infrastructures based on web services, based on technologies like TPF, or using in-house solutions. However, middleware that relies on centralized approaches to data aggregation suffers from high communication overheads, lack of scalability, and unpredictably high processing workloads at central servers.

Our solution is to use in-network aggregation to reduce the load problems encountered in centralized approaches. This approach exploits the fact that data in these 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 can contribute some of its cycles towards processing of the updates it is forwarding, the resulting advantage being the distribution of processing workload and a possible reduction in communication overhead involved in transmitting data updates.

In this paper, we examine how to construct a distributed system for processing and aggregating streams of data. However, in order to set up such a system with nodes ready to contribute their resources for data processing, we must address several challenges, including:

- *Ease of Deployment* – provide a simple interface for composing new streams from existing streaming data and also support run-time modifications to stream composition conditions.
- *Scalability* – there may be hundreds of streaming data sources, and the system should be incrementally scalable without significant overhead or effort.
- *Heterogeneity* – streaming data arrives at the sink after traversing a possibly heterogeneous set of nodes, which means that the system should support in-network processing of the streams at any node despite varying resource capabilities and operating environments.
- *Dynamism* – should automatically reconfigure to deal with changes in network conditions, node overloads and changes in data stream rates.

- *Performance* – since updates arrive at a very high rate, the infrastructure should not impose large overheads when aggregating and processing the updates.

We are implementing a Distributed Stream Management Infrastructure (DSMI) to compose new data streams by aggregating and processing existing streaming data originating at distributed locations. The system supports a SQL-like language to describe the data-flow graph for producing the new, transformed stream from existing data streams. The language allows users to refer to any stream originating in the system and supports attribute selection and join operations as in traditional databases. A resource-aware network-partitioning algorithm, described later, is used to assign operators from the flow graph to the underlying network nodes. The infrastructure relies on ECho [5], a pub-sub middleware developed at Georgia Tech, to deploy the data-operators for processing and forwarding the streaming updates in a heterogeneous environment. Automatic reconfiguration of stream overlays is achieved by coupling the resource information collected from participating hosts with the Proactive Directory Service [7] (PDS), which is a subscription-based monitoring tool also developed by our group.

1.1. Example: Operational Information System

An operational information system (OIS) [6] is a large-scale, distributed system that provides continuous support for a company or organization's daily operations. One example of such a system we have been studying is 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 sets 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 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 clients may also generate additional state updates, such as changes in flights, crews or passengers.

The key problems addressed by this paper are to reduce communication overheads, by selectively streaming the events; distributing the processing workload by using the computing resources spread across the organization; and implementing easy to use high-level language constructs for specifying new data-flow graphs.

1.2. Related Work

The stream management infrastructure we have implemented is very closely associated with topics of interest to the middleware community, and to those interested in large-scale distributed data management.

Stream Processing & Distributed Databases

Data-stream processing has recently been an area of tremendous activity for database researchers; several groups such as STREAM [10] at Stanford, Aurora [11] at Brown and MIT, and Infopipes [12] at Georgia Tech have been working to formalize and implement the concepts for data-stream processing. Most of these efforts have commonly assumed an on-line warehousing model where all source streams are routed to a central site where they are processed. There have also been some preliminary proposals that extend the single-site model to multi-site, distributed models and environments [13, 14]. Our work is also a step in this general direction. Of particular mention is the work by Madden et al. [15] that demonstrates the advantage of in-network data-aggregation in a wireless multi-hop sensor network.

Distributed query optimization deals with site selection for the various operators and has been explored in great detail in the context of distributed and federated databases [16, 17]. However, these systems do not deal with streaming queries over streaming data, which present new challenges, especially in dealing with resource limitations.

Pub-Sub Middleware

Pub-sub middleware like IBM's Gryphon [4], ECho [5], ARMADA [18] and more recently Hermes [19] have well established themselves as messaging middleware. Such systems address issues like 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 the middleware to provide high-level programming constructs to describe resource-aware and 'useful' data-flows.

Network Partitioning & 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 [2], enterprise management solutions [3] and grid computing efforts [29]. These applications perform task allocation to servers much in 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 an application-driven utility distinguishes our infrastructure.

Overlay networks [20, 21] 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 [27], which is similar to resource-aware reconfiguration of the stream overlay in our infrastructure. In our case reconfiguration is very closely associated with the application level data requirements.

1.3. Roadmap

The remainder of this paper is organized as follows. In Section 2, we discuss the design of the basic components of the infrastructure, explaining its layered architecture and a brief description of the layers' functionality. Section 3 describes the data-flow graph deployment problem in detail, followed by a brief description of the network hierarchy and its use in deployment and maintenance of the stream overlay. Implementation issues for the infrastructure are discussed in Section 4. Section 5 presents an experimental evaluation of the proposed deployment algorithm, including results that were obtained using real enterprise data. Finally we conclude in Section 6 with a discussion of possible future directions.

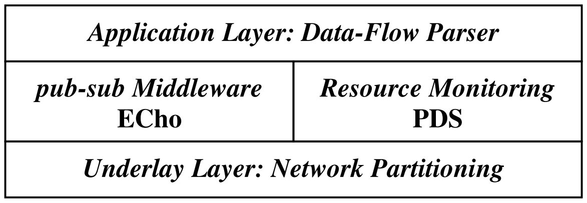
Application Layer: Data-Flow Parser
pub-sub Middleware
ECho
Resource Monitoring
PDS
Underlay Layer: Network Partitioning

Figure 1. Three layered architecture of the DSMI

2. Software Architecture

Our distributed stream management infrastructure (DSMI) is broadly composed of three layers as shown in Figure 1: (1) the Application Layer is responsible for accepting and parsing the data composition requests and constructing the data-flow graph, (2) the Middleware Layer consists of the ECho middleware and the PDS resource-monitoring infrastructure for deployment and maintenance of the stream overlay, and (3) the Underlay Layer organizes the nodes into hierarchical partitions that are used by the deployment infrastructure. The following subsections briefly describe these three layers.

2.1 Application Layer: Data-Flow Graph

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, and 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 that are specified as `operator(attribute_list [,parameter_list])`, where examples of such operations include `SUM`, `MAX`, `MIN`, `AVG`, `PRECISION`, etc. The system also provides the facility to extend this feature by adding user-defined operators. An example data-flow description is shown in Figure 3.

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 streaming between operators. This graph is deployed in the network by assigning operators to network nodes.

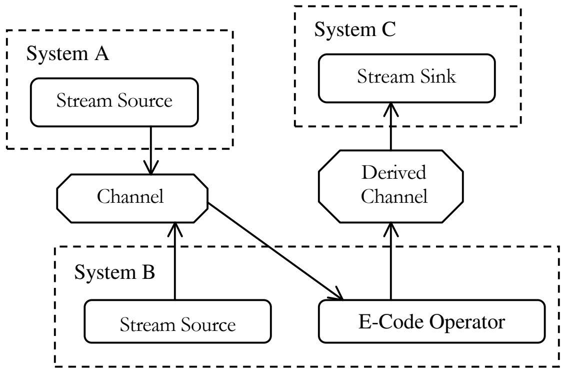

Figure 2. The ECho Framework

2.2. Middleware Layer: ECho and PDS

The Middleware Layer supports the deployment and reconfiguration of the data-flow overlay. This support is provided by two components: ECho and PDS.

The ECho framework is a publish/subscribe middleware system that uses channel-based subscription (similar to CORBA). ECho streams data over *stream 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 sinks. An example system is shown in Figure 2. 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). The 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 fill 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 submitted, 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, or may cause operators to be created or updated to stream additional relevant data if necessary.

Network-wide resource availability information is managed by the Proactive Directory Service (PDS). 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 includes a proactive, push-based access mode. 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. The infrastructure uses PDS objects to receive resource updates from the system when operating conditions change.

2.3 Underlay Layer: Network Partitioning

This layer is responsible for maintaining a hierarchy of physical nodes in order to cluster nodes that are "close" in the network sense, based on measures like end-to-end delay, bandwidth or inter-node traversal cost (a combination of bandwidth and delay). The hierarchy is

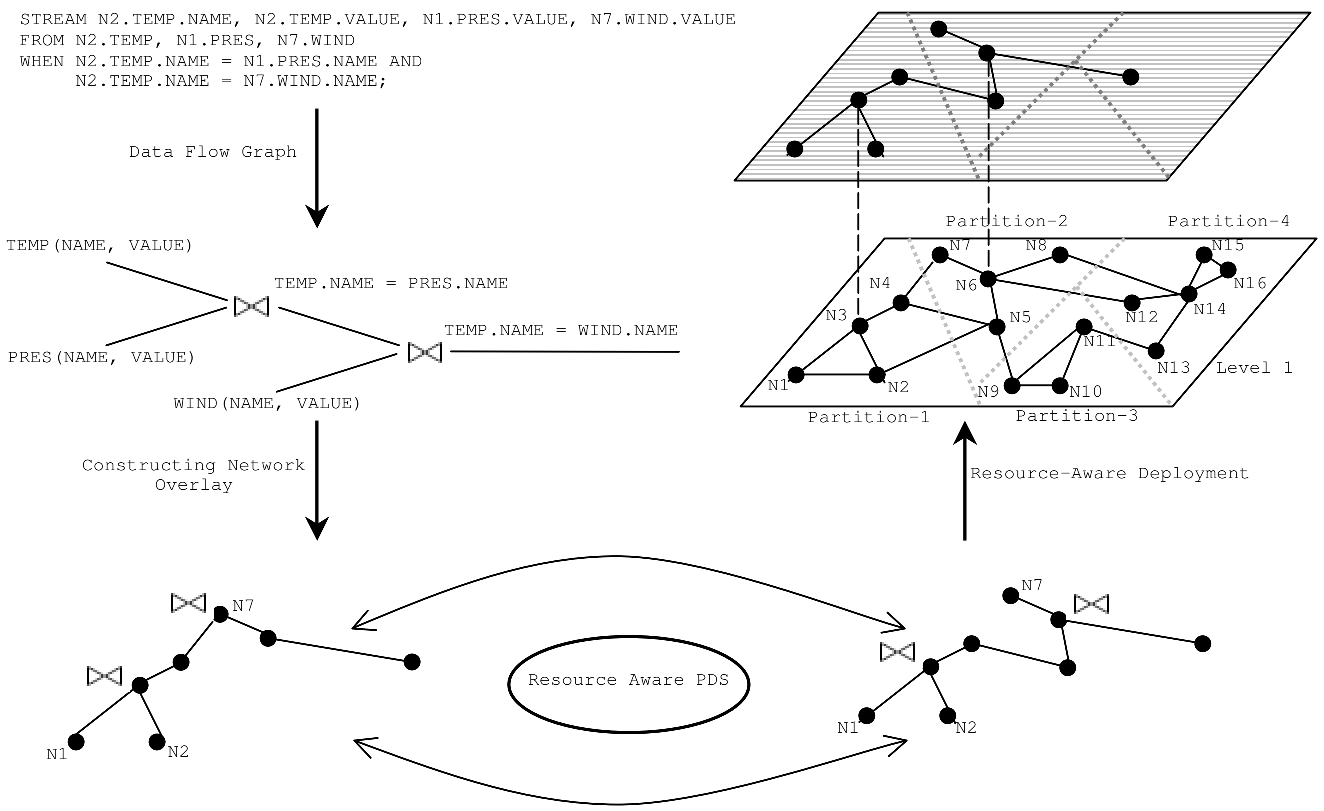

Figure 3. Steps involved in deploying the stream overlay

used for network-aware deployment of the data-flow graph. Each node in a cluster knows about the costs of paths between each pair of nodes in the 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 average min cost path to the other coordinator nodes that fall in the same partition at that level. An example is shown in Figure 4.

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.

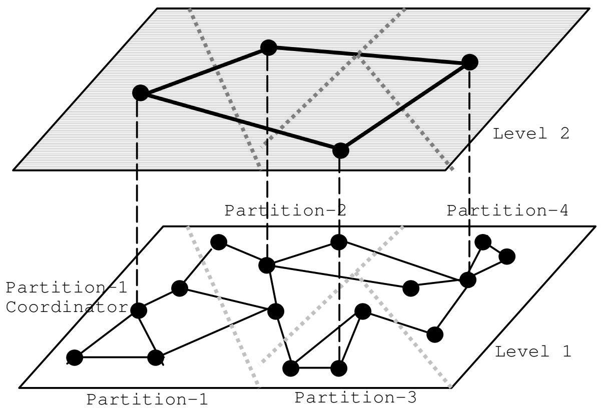

Figure 4. Hierarchical Network Partitioning

3 Deployment & Dynamic Reconfiguration

This section formally describes the data-flow graph deployment problem and then 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 resource-awareness.

3.1 Problem Statement

We consider the underlying network as a graph $N(V_n, E_n)$, where vertices V_n represent the actual physical nodes and the network connections between the nodes are represented by the edges E_n. We further associate each edge e_{ni} with a cost c_i that represents the application-oriented cost of traversing the corresponding network link. The data-flow graph derived from the SQL-like description is similarly represented as a graph $G(V_g, E_g)$ with each vertex in V_g representing a source-node, a sink-node or an operator i.e.

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

$V_{g\text{-}sources}$ 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 corresponding decrease. The edges 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 lowest cost path between the network nodes that v_{gj} and v_{gk} are assigned to. We define $cost(M)$ as the sum of the costs of the network edges mapped to the edges in the data flow graph:

$$cost(M) = \sum_{e_{ni} \in M} cost(e_{ni})$$

For example, consider a cost function that measures the end-to-end delay. 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 cost corresponding to edge e_g is the delay along the shortest path between the vertices v_{ni} and v_{nj}.

The problem is to construct the lowest cost mapping M between the edges E_g in G to edges E_n in N.

3.2 Distributed Deployment 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 obviously not scale for very large networks, and the planner can become a central point of failure. Our partitioning-based approach deploys the data-flow graph in a more decentralized way. In particular, nodes in the network self-organize into a network-aware set of clusters, such that nodes in the same cluster have low latency. Then, we can use this partitioned structure to deploy the data-flow graph in a network-aware way, without having full knowledge of the delay 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 be optimal, since our approach trades guaranteed optimality for scalable deployment. However, since the deployment is network-aware, the mapping should have low cost. 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 minimum cost 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 network values are used to partition nodes at this level. For any other level i in the hierarchy the average inter-partition cost (i.e. end-to-end delay, bandwidth, etc.) from level i-1 are used for partitioning the coordinator nodes from the level i-1. The approximate cost between any two vertices v_{nj} and v_{nk} at any level i in the hierarchy can be determined using the following equations:

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

and

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

In simple words, the cost at level i between any two vertices v_{nj} and v_{nk} of N is 0 if the vertices have the same coordinator at level i-1, otherwise it is equal to the cost 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) coordinators. 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 cost for each assignment is calculated using the algorithm shown in Figure 5 and the assignment with lowest cost 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.

3.3 Reconfiguration

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 minimum cost analysis is required. These features allow us to limit the reconfiguration to within the cluster boundaries, 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. Since it is impractical to respond to all such events reported by the PDS, we set thresholds that should be reached to trigger a reconfiguration. For example, a cluster-coordinator may recalculate the minimum cost paths and redeploy the assigned sub-graphs when more than half the links in the cluster have reported change in end-to-end delay. However, setting such thresholds depends on the application-level requirement for resource-awareness. In ongoing work we are developing a closer integration between the application-level requirements and the reconfiguration framework.

Note that reconfigurations are not lossless in terms of updates and 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 trying to model reconfiguration as a database-style transaction in order to achieve losslessness.

```
graphCost(Node n, Level l){
  if(n->left == null && n->right == null)
    return 0;
  if(n->left != null)
    cl = pathCost(n->node[l], n->left->node[l])
        + graphCost(n->left, l);
  if(n->right != null)
    cr = pathCost(n->node[l], n->right->node[l])
        + graphCost(n->right, l);
 return cl + cr;
}
```

Figure 5. Algorithm for calculating graph cost at a level

3.4 Advantages of the DSMI Approach

DSMI is an infrastructure for distributed processing of data streams. Its main contribution is in composing and transforming data streams using a data flow graph that can be defined declaratively and deployed efficiently. We have already discussed the algorithm used for resource-aware deployment of the stream overlay. Following are the advantages of our architecture for stream management:

- *Online Predicate Rewriting* – the data-operators allow the stream sink to fine-tune their behavior by updating a remotely accessible data-structure associated with each operator. This is used to support dynamic modification of composition parameters and conditions.
- *Resource-Aware Reconfiguration* – once deployed, a stream overlay reacts to changes in operating environment at two levels: to respond to changes in local conditions, an intra-partition level reconfiguration is done, while an inter-partition level reconfiguration handles substantial changes in operating conditions.
- *Operator Duplication* – the system allows for duplication of a limited type of operators to achieve parallel processing of some rapid update streams in resource-constrained environments. This technique may result in some updates being re-ordered, so it is only applicable in certain scenarios.
- *Operator Migration* – the system allows the user to initiate operator reconfigurations to complement the automatic resource-aware reconfiguration being done by the system.
- *Embedded Portability* - the E-Code Language is a portable subset of C that includes enough functionality to implement stream operators. The E-Code compiler can be extended to include other language features and can be easily ported to new platforms. Therefore, the operators are easily portable to heterogeneous nodes.
- *Minimal Impact on Performance* - given that the data-operators are invoked for each update and are deployed on a remote machine, the performance of the system depends heavily on the performance of these operators. Dynamic native compilation of the data-operators reduces the overheads of update processing.

```
{
  // an example of customized computation
  int i, j;
  for(i = 0; i<256; i= i+2) {
    for(j = 0; j<256; j=j+2) {
      output.image[i/2][j/2] =
(input.image[i][j] + input.image[i+1][j+1] +
input.image[i][j+1] + input.image[i+1][j]) / 4 ;
    }
  }
  return 1;
}
```

Figure 6. Resolution Reduction Operator as E-Code

4 Implementation

DSMI has been implemented using C++ and is closely integrated with the ECho and the PDS modules developed by our group. The system is brought-up by specifying a set of initial nodes, and an instance of DSMI is started at a well-known port on each such node. A node can join or leave the infrastructure at a later point of time. The system maintains a distributed hash table to map the user specified unique node names to the IP-address for each node. We utilize user-specified names to facilitate the task of graph composition by users. The hierarchical node-partitioning module runs an iterative variant of the well-known k-means [28] clustering algorithm to partition nodes based on inter-node end-to-end delay values.

A user can create a stream schema at any node by using the `CREATE STREAM` command at the infrastructure prompt. Registering a schema causes an ECho typed-channel to be created that has the capability to carry any update, which conforms to the specified schema. The node can then start streaming data on this channel and other node can refer to this stream as `node_name.stream_name`.

When a stream composition request is submitted at any node in the infrastructure, it is parsed to create a data-flow graph. Each edge in the graph is mapped to an ECho channel, which is instantiated with appropriate data carrying capability. Each operator in the flow graph is either a pre-compiled routine (operators which are hard to express as E-Code) or an appropriate E-Code snippet. An example image resolution reduction snippet written in E-Code is shown in Figure 6. The flow-graph operator consists of one or two incoming channels, an outgoing channel and an operator routine. Operator information is specified in XML and sent to the node where the operator has to be instantiated. Since we are using a pub-sub middleware, instantiating an operator consists of becoming a subscriber to appropriate incoming channel(s), a publisher to the outgoing channel and starting the associated operator routine. The ease of operator deployment helps us to reduce the overhead during reconfigurations as no new channels are created; only the channel publisher and the subscriber change to reconfigure the overlay.

5 Experiments

We ran a set of experiments to evaluate the performance of our architecture. First, we ran microbenchmarks to examine specific features of our system. Then, we created an end-to-end setup for an application case study using real data from Delta Airlines' OIS. Our results show that our system is effective at deploying and reconfiguring data-flow graphs for distributed processing of streaming data.

5.1 Experimental setup

The GT-ITM internetwork topology generator [8] was used to generate a sample Internet topology for evaluating our deployment 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

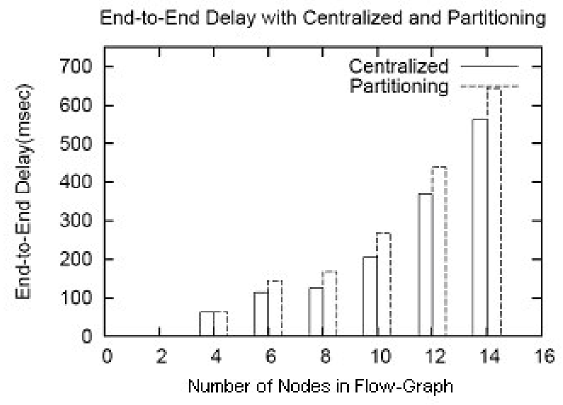

Figure 7. Comparison of end-to-end delay for centralized and partitioning approach

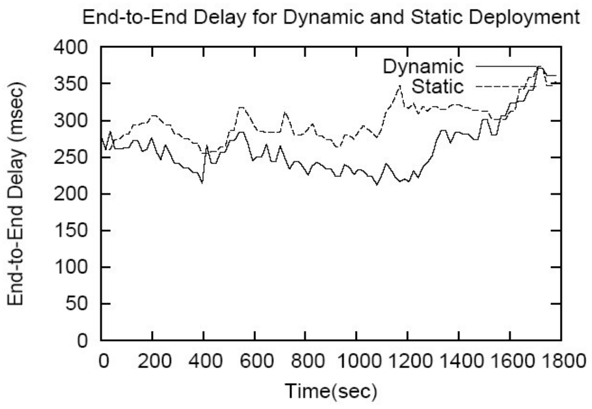

Figure 8. Variation of end-to-end delay with and without dynamic reconfiguration

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 [9]. 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.

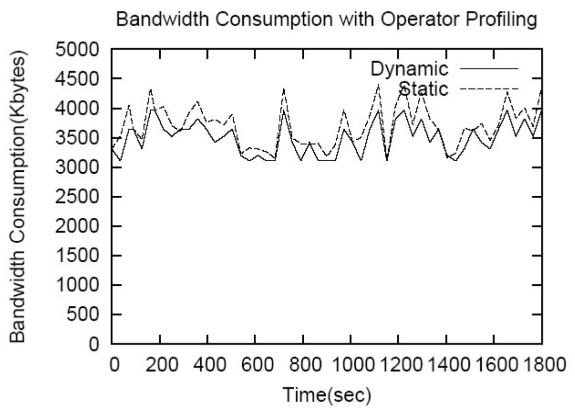

Figure 9. Variation in bandwidth consumption with dynamic reconfiguration using Operator Profiling

5.2 Microbenchmarks

The first experiment focused on comparing the cost of a deployed data-flow graph using the centralized model as opposed to the partitioning based approach used in our infrastructure. Since in centralized approach we assume that a single node knows about minimum cost paths to all other nodes, the centralized approach gives the optimal deployment solution. However, the deployment time taken by centralized approach increases exponentially with the number of nodes in the network. Figure 7 shows that although the partitioned-based approach is not optimal, the cost 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 was conducted to examine the effectiveness of dynamic reconfiguration in providing an efficient deployment. Figure 8 shows the variation of end-to-end delay for a 10-node data-flow graph with changing network conditions, as simulated by introducing cross-traffic. The performance with dynamic reconfiguration is clearly better than with static deployment. It may be noted that at some points, cost of the dynamically reconfigured flow-graph becomes more than that of the static deployment. This happens because the cost calculation algorithm used in our approach calculates the graph cost that is an approximation of the actual deployment cost. 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 cost of the dynamic deployment is not much worse than the static deployment. Moreover, for most of the time dynamic reconfiguration produces a lower cost deployment.

We also conducted experiments to compare the bandwidth consumption with and without dynamic reconfiguration. Each source was assumed to have a certain update rate of the form bytes/sec and each link was associated with a cost incurred per byte of data transferred using the link. Thus, at any point of time a deployed data graph has a cost, which is dependent on the links being used by the flow. We simulated a change in resolution factor (the ratio of the amount of data flowing out versus flowing in) for each operator in the flow graph and measured the corresponding bandwidth utilization with dynamic reconfiguration and static deployment. We notice that although dynamic reconfiguration helps in keeping the bandwidth consumption low, it does not offer very substantial gains; this is because when reconfiguration is driven by operator resolution it offers only a limited space for re-deployment.

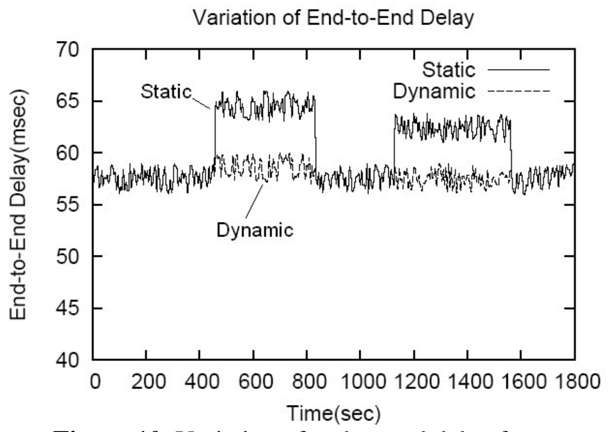

Figure 10. Variation of end-to-end delay for network perturbation and processor overload

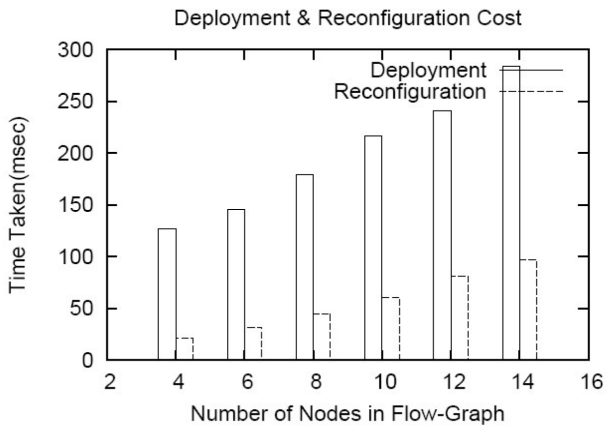

Figure 11. Comparison of deployment and reconfiguration cost on Emulab nodes

5.3 Application case study

The next set of experiment was conducted on Emulab [23] with real data from the Delta OIS combined with simulated streams for Weather and News. The experiment was designed to emulate 4 different airport locations. The inter-location delays were set to ~50ms while delays within an airport location were set to ~2ms. The emulation was conducted with 13 nodes (Pentium-III, 850Mhz, 512MB RAM, RedHat Linux 7.1) and each location had only limited nodes connected to external locations. The experiment was 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 'destination' location and switch over to seating information for the aircraft at boarding gate. Other information displayed on such monitors includes names of wait-listed passengers, and current status of flight, etc. We deployed 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 (Delta's TPF facility in this case).

The first experiment conducted on Emulab studied the behavior of system in case of network perturbation and then studied its response to processor overload. Once the data flow graph for providing an overhead display feed was deployed, we used iperf [24] to introduce traffic in some of the links used by the flow-graph. This is represented by the first spike in Figure 10. With dynamic reconfiguration 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 dynamic reconfiguration we end with a better delay than the static deployment. Even with dynamic reconfiguration 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 was conducted to compare the time for initial deployment and reconfiguration. Figure 11 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 1. 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

Middleware Microbenchmarks on Emulab

Table 1 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 [5].

6 Conclusions & Future Work

In this paper we presented DSMI, a highly scalable and resource-aware approach to distributed stream management. The approach makes use of in-network data aggregation to distribute the processing and reduce the communication overhead involved in large-scale distributed data management. One of the important features of our infrastructure is its ability to efficiently and scalably deploy data-flows across the network. The run-time reconfiguration of the deployed flow graph in response to change in operating conditions and support for high-level language constructs to describe data-flows are other distinguishing features of the infrastructure. As a part of ongoing work we are examining how to avoid loss of updates and state in case of reconfiguration. We are also examining how to represent reconfiguration as a database-style transaction, motivated by similar work done by our group [25]. Another aspect of the infrastructure that is of particular interest to us is the closer integration of reconfiguration policy with the application level requirements. Overall, our architecture is a flexible, scalable platform for distributed processing of stream data.

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 [26] for providing data and a very useful use-case for the experiments.

References

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

[2] 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.

[3] 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.

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

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

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

[13] Y. Ahmad, U. Çetintemel: Network-Aware Query Processing for Distributed Stream-Based Applications. Proceedings of the Very Large Databases Conference, VLDB 2004, Toronto, Canada.

[14] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Proceedings of the 19th International Conference on Data Engineering, ICDE 2003,.

[15] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '02), Massachusetts, Dec. 2002.

[16] M. J. Franklin, B. T. J´onsson, and D. Kossmann. Performance tradeoffs for client-server query processing. SIGMOD Record, 25(2):149–160, June 1996.

[17] D. Kossmann. The state of the art in distributed query processing. ACM Computing Surveys, 32(4):422–469, 2000.

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

[19] 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.

[20] 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.

[21] 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.

[22] Z. Cai, G. Eisenhauer, C. Poellabauer, K. Schwan, M. Wolf, IQ-Services: Resource-Aware Middleware for Heterogeneous Applications. 13th Heterogeneous Computing Workshop (HCW 2004), Santa Fe, NM, April 2004.

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

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

[25] C. Isert, K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. International Parallel and Distributed Processing Symposium (IPDPS), May 2000.

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

[27] 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.

[28] R. O. Duda and P. E. Hart. Pattern Classication and Scene Analysis. John Wiley & Sons, 1973.

[29] 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.