IFLOW: Resource-Aware Overlays for Composing and Managing Distributed Information Flows

Vibhore Kumar, Zhongtang Cai, Brian F. Cooper, Greg Eisenhauer, Karsten Schwan
Mohamed Mansour, Balasubramanian Seshasayee, and Patrick Widener
College of Computing, Georgia Institute of Technology
801 Atlantic Drive, Atlanta GA 30332, USA
{vibhore, ztcai, cooperb, eisen, schwan, mansour, bala, pmw}@cc.gatech.edu

ABSTRACT
Information flow applications like distributed continual queries, online scientific collaboration and visualization, and the operational information systems used by large corporations involve the acquisition, processing, and dissemination of events across distributed computing platforms. Multiple distinct middleware frameworks have been developed to support such data-intensive applications, including publish-subscribe systems, messaging infrastructures, and in-house solutions customized to specific business needs. This paper presents IFLOW, an overlay messaging framework that provides the basic functionality for realizing the multiple middleware models now present in commercial and academic endeavors. IFLOW offers (1) overlay management functions, used to construct, maintain, and manage the overlay networks across which information flows travel, (2) support for resource awareness, which is the online knowledge of the underlying network and processing resources and which is used to manage information flows in order to maintain and maximize their utility to applications, (3) methods for the dynamic deployment of application logic into the overlay, and (4) concrete representations of information flow-graphs, used for control functions and consisting of descriptions of sources, sinks, flow operators, edges and utility functions. We demonstrate IFLOW's generality by using it to implement three different communication models: a high performance model for streaming scientific data, a utility-based information flow model for operational information systems, and the publish-subscribe event distribution model. The high performance of these application models as realized with IFLOW is due to multiple factors, including IFLOW's carefully crafted fast path vs. control abstractions. Further, experimental evaluations show that IFLOW facilities like utility-driven flow deployment, resource-aware reconfiguration, and update routing provide high performance for information flow applications.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – *distributed applications.* K.6.4 [**Management of Computing and Information Systems**]: System Management – *centralization/decentralization.*

Eurosys'06, April 18–21, 2006, Leuven, Belgium.

General Terms
Management, Performance, Design

Keywords
Information Flow, Continual-Queries, Scientific-Collaboration, Flow-Deployment, Flow-Adaptation, Routing, Processing, Dissemination.

1. Introduction
Distributed information-intensive applications range from emerging systems like continual queries [8, 9, 13], to remote collaboration [34, 35] and scientific visualization [18], to the operational information systems used by large corporations [23]. At present, these applications utilize diverse messaging frameworks, where each such information flow framework implements basic primitives to acquire, process, and disseminate information across the underlying distributed system. Enhancements implemented over these primitives typically address the needs of specific application domains. Examples include scalability support like dynamic event routing in publish-subscribe systems, the explicit descriptions of information flows supporting continual queries and the tunable-operators found in middleware for scientific collaboration.

Generalizing from domain-specific solutions, our research group has created the IFLOW middleware, which provides a suitable framework for the multiple messaging models now in common use. IFLOW uses a formalized notion of information flows and, based on this formalization, provides a complete set of abstractions and a small set of primitives for constructing and materializing different application-level messaging models. Specifically, information-flows are explicitly described by *Information Flow-Graphs*, consisting of descriptions of sources, sinks, flow-operators, edges and a utility-function. *Sources* represent the producers, *sinks* are the consumers and *flow-operators* transform and combine data streams and they together constitute the vertices of the flow-graph. The *edges* represent typed information flows between such vertices. Once the flow-graph has been described, deployment (i.e. the mapping and instantiation of operators on physical nodes, and edges on one or more network links), and reconfiguration (to maintain the optimality of flow-graph deployment) is driven by the *utility-function*. This function measures the utility of the deployment and thus acts as the vehicle for encoding user and system preferences.

The information flow-graph serves as the *abstraction* visible to applications, and it provides the concrete representation on top of which we construct richer application-level messaging models:

- *Information Flow-Graph Abstraction* -- IFLOW's generality is due to its simple information flow-graph abstraction, providing a basis for creating a rich set of communication models. When implementing a pub-sub model, for example, the developer would use an information flow-graph to describe the logical connectivity implied by pub-sub's 'subscriptions' [1] and the processing carried out by 'handlers' [2]. The developer would then use the basic overlay network construction, deployment and management primitives provided by IFLOW to implement efficient update routing between publishers (i.e., IFLOW sources) and subscribers (i.e., sinks). Alternatively, information flows implementing continuous queries can be realized by using IFLOW to instantiate SQL-like flow-graphs [13], where flow operators take the responsibility for 'join' or 'select' operations. Suitable runtime methods for deployment and management can be implemented with the basic support provided by IFLOW's application-independent overlay and resource management primitives.

High performance, scalability, and utility-based operation for communication models and applications constructed with IFLOW results from the following *core-functionality*:

- *Resource-Awareness* – The dynamic nature of distributed execution platforms requires applications to configure and adjust their runtime behavior based on current platform conditions. Towards this end, IFLOW combines extensible network and platform monitoring functions with cross-layer 'performance attributes' to match application requirements to the available network and processing resources. This allows IFLOW to determine the initial deployment of the flow graph that maximizes performance given the available resources. Similarly, IFLOW can adapt its deployment at runtime; example adaptations of deployed flow graphs include (1) dynamic operator deployment [7] in response to changes in available processing resources and (2) runtime adjustments of parameterized operators to match data volumes to available network bandwidths [22].
- *Utility-Driven Self-Regulation* -- End users of distributed information systems desire the timely delivery of quality information content, regardless of the dynamic behavior of the networks and computational resources used by information flows. IFLOW uses application-specific utility functions to govern both the initial deployment of information flows and their runtime regulation, thereby enabling the creation of application- or domain-specific self-regulation functionality.
- *Flexibility and High Performance through Dynamic Function Deployment* -- An important attribute of IFLOW is its ability to dynamically generate and deploy efficient native binary codes at overlay nodes. This allows IFLOW to change at runtime both what data is streamed to which overlay nodes and which operations are applied to such data. The outcome is flexibility in determining where 'best' to run certain application functions. An example for high performance applications is the runtime augmentation of server functionality to better meet current client needs [7]. Another outcome is the ability to create efficient IFLOW applications for dynamic computing and network infrastructures that range from high performance systems [19], to workstations and even portable devices [18] and from high speed LANS to wide area [21] and wireless connections.
- *Efficient Exchanges of Typed Data* -- IFLOW explicitly addresses high performance information flows, including those with large data volumes. Its efficient methods for binary structured data representation and transport can perform comparably to and better than those of well-known high performance computing infrastructures like MPICH [19], offering low send/receive overheads and using dynamic native-code generation to deal with data heterogeneity and to instantiate efficiently executable flow-operators.
- *Scalability through Hierarchical Management* -- IFLOW scales to large underlying platforms by using hierarchical management techniques, as exemplified by its automatic flow-graph partitioning algorithm presented in [13]. One way in which this algorithm attains scalability is by making locally rather than globally optimal deployment decisions, thereby limiting the amount of non-local resource information maintained by each node.

The IFLOW project leverages a multi-year effort in our group to develop middleware for high-end enterprise and scientific applications. As a result, the IFLOW prototype described in this paper uses multiple software artifacts developed in our prior work. IFLOW's resource- and network-awareness is supported by the network monitoring and data adaptation methods described in [21]. High performance and the ability to deal with large data volumes are derived from its methods for dynamic binary code generation and the careful integration of multiple software layers explained in [2], as well as by its binary methods for data representation and runtime conversion for heterogeneous systems [19]. Performance and scalability are due to the separation of overlay-level messaging from the application-level models desired by end users and as stated above, its hierarchical methods for the efficient deployment of large information flow-graphs to distributed systems [13, 14].

While leveraging previous work to construct an efficient IFLOW prototype, this project constitutes an entirely new direction in our research. First, IFLOW uses a well-defined set of information flow abstractions implemented with software that substantially differs from and improves upon that used in our past work on publish-subscribe infrastructures [2]. Second, a unique attribute of IFLOW is its ability to efficiently implement multiple application-level messaging models, including the in-flight data manipulation model (suitable for scientific collaboration and for continual query applications) and the pub-sub communication model. Third, in contrast to our previous work on pub-sub [2], scalability and high performance for IFLOW applications are attained by separating its underlay (i.e., resource awareness), from messaging (i.e., overlay primitives), from control (i.e., implementations of different messaging models). Fourth, by using utility-driven self-regulation in concert with resource-awareness, IFLOW flow-graphs deployed across dynamic distributed underlying platforms can continuously provide high performance services to end users.

Experimental results presented in Section 7 demonstrate IFLOW's capabilities. A low send/receive overhead, $14.37\mu sec$ and $5.45\mu sec$, respectively, and a throughput comparable to that of raw-sockets for message sizes over 10KB makes the case for high-performance applications, closely matching the performance attained by well-known HPC infrastructures like MPICH. Dynamic reconfiguration is enabled by low instantiation/deletion overheads for overlay components, $0.89\mu sec$ and $0.03\mu sec$ for

local (i.e., in the same address space) operations. Finally, the paper substantiates its claim of both the completeness of abstractions and the general efficiency of IFLOW features with performance evaluations of three different application and communication models implemented using IFLOW. Our performance evaluation includes a head-to-head comparison with DTMI, an industrial-strength middleware currently used by Delta Air Lines.

The remainder of this paper is organized as follows. Section 2 describes two applications - scientific collaboration and an airline's operational information system, both of which can be implemented using the abstractions provided by the IFLOW messaging framework. Section 3 discusses related work. The software architecture and the design of basic IFLOW components are discussed in Section 4, explaining its layering, followed by a description of each layer's functionality. Section 5 describes the algorithms used in the IFLOW framework; these include the InfoPath algorithm used to construct an application-level

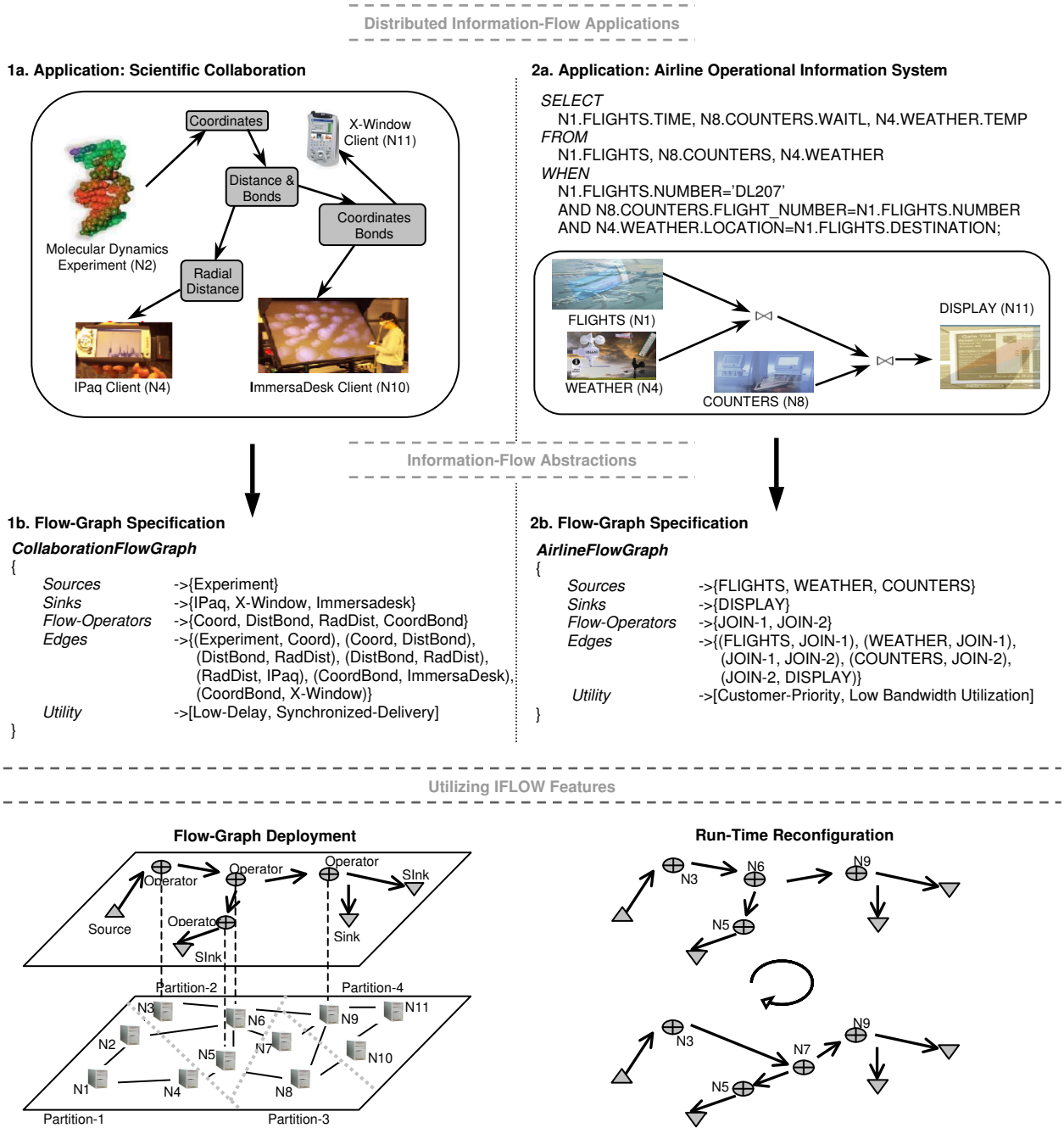

Figure 1. Implementation of different messaging models using IFLOW

multicast tree, the PathMap algorithm used to efficiently map the flow-graph onto underlay nodes, and online algorithms for run-time reconfiguration. Section 6 describes our experiences with the implementation of a scientific collaboration application, a prototype operational information system and a pub-sub middleware using the IFLOW framework. Section 7 presents an experimental evaluation of IFLOW's claims. Finally, we conclude in Section 8 with a discussion of possible future directions.

2. Examples

Figure 1 depicts two examples of applications that can benefit from the generality of the IFLOW model and infrastructure – real-time scientific collaboration and an airline's operational information system.

The left portion of Figure 1 shows a *collaborative real-time visualization* application using a many-to-many information flow, with data originating in a parallel molecular dynamics (MD) simulation, passing through operators that transform and annotate the data, and ultimately flowing to a variety of clients. This real-time data visualization requires synchronized and timely delivery of large data volumes to collaborators. For example, it is unacceptable when one client consistently lags behind the others. Similarly, there may be end-to-end control loops, as when one collaborator drives the annotation of data for other collaborators. Finally, there exist quality/performance tradeoffs, as end users may prefer consistent frame rates to high data resolution (or vice versa).

The right-hand information flow in Figure 1 represents elements of an *operational information system* (OIS) providing support for the daily operations of a company like Delta Air Lines (see [23] for a description of our earlier work with this company). Here, SQL-like operations translate into a deployed flow-graph with sources, sinks and operators. Such an OIS imposes the burden of high event rates on underlying resources, which must be efficiently utilized to deliver high utility to the enterprise. For example, a request pertaining to seat-assignment for a business-class customer may be given a higher priority than one for coach-class, because it reflects higher returns for the business. Similarly, other factors like time-to-depart, destination, etc. can drive the prioritized allocation of resources to the deployed information flows.

The remainder of this paper will describe in more detail how IFLOW supports applications like these and others. Briefly, the main benefits of the IFLOW architecture for these applications are (1) its ability to separate basic, fast-path data exchanges from the management functions needed to ensure levels of performance that meet developers' utility goals, coupled with (2) a rich set of functionality to support runtime flow management. For instance, the mapping of information flows to underlying heterogeneous, dynamic platforms is carried out by IFLOW-provided mapping methods, guided by user-specified utility functions. Runtime flow management is based on real-time resource monitoring in the underlay, exploited by adaptive methods that can adjust data contents and/or flow rates to available resources, also based on utility [14]. The OIS developer may use these facilities to embed business sense into the system, to drive the configuration and management of information flows to attain high business value. Scientific applications may use them to embed suitable quality/performance tradeoffs via dynamically tunable operators, such as image down-sampling or compression [21].

3. Related Work

Publish-Subscribe Infrastructures & Application-Level Multicast. Pub-sub middleware like IBM's Gryphon [1], ECho [2], ARMADA [3] and Hermes [4] are examples of application-level messaging middleware. These systems automatically route information to subscribers via scalable messaging infrastructures, thereby simplifying the development of distributed messaging applications. While many pub-sub middlewares make use of IP multicast [5], IFLOW (like Gryphon [1]) uses application-level overlays for efficient information dissemination. Other systems that implement application-level message multicast include SCRIBE [15] and SplitStream [16], both focused on peer-to-peer systems. IFLOW does not currently deal with common peer-to-peer issues, like dynamic peer discovery or frequent peer disconnection, but the basic multicast functionality offered by these systems is easily realized with our abstractions. Other distinguishing features of IFLOW are its support for dynamic deployment and run-time optimization techniques to adapt information flows to changes in user requirements or resource availabilities.

Distributed Data-Stream Processing. Projects like TinyDB [6], STREAM [8], Aurora [9], and Infopipes [10] have been working to formalize and implement database-style data stream processing for information flow applications. Some existing work deals with resource-aware distributed deployment and optimization of SQL-like execution trees [11, 12, 13]. IFLOW goes beyond database-style streams in order to provide a general abstraction for expressing complex information flows, including pub-sub, scientific data flows, and others in addition to SQL-like queries. In addition, IFLOW facilitates the use of application-specific operators that implement desired runtime quality/performance tradeoffs. IFLOW also provides self-regulating information flow overlays to deal with run-time resource variations, a capability not present in many existing systems.

Scientific Collaboration. High-speed networks and grid software have created new opportunities for scientific collaboration, as evidenced by past work on client-initiated service specialization [18], remote visualization [29], the use of immersive systems across the network [31], and by programs like the Terascale Supernova Initiative [34]. In all such applications, scientists and engineers working in geographically different locations collaborate, sometimes in real-time, by sharing the results of their large-scale simulations, jointly inspecting the data being generated and visualized, and sometimes even directly running simulations through computational steering [33] or by control of remote instruments [32]. Such large-scale collaborations require infrastructures that can support the efficient `in-flight' capture, aggregation, and filtering of high-volume data streams. Resource-awareness is required to enable suitable run-time quality/performance tradeoffs. IFLOW addresses these needs with built-in support for the resource-aware deployment of customized information flow-graphs and by supporting dynamic reconfiguration policies that maintain high performance levels for deployed flow-graphs.

Self-Configuring Services, Architectures, and Infrastructures. Researchers in the pervasive computing domain believe that with the computing power available everywhere, mobile and stationary devices will dynamically connect and coordinate to seamlessly help people in accomplishing their tasks [44]. Tools like one.world [45] provide an architecture for simplifying application

development in such environments. While IFLOW's information flow abstraction is sufficiently rich to deploy flows that accomplish user tasks in mobile environments, the focus of its implementation on high-end systems makes it complementary to much of the work being done in the pervasive computing domain. In contrast, it is straightforward for IFLOW to manage evolving data-sources, as done in systems like Astrolabe [17], which has the capability to self-configure, monitor and adapt a distributed hierarchy to manage evolving data-sources. An interesting generalization of IFLOW would be to introduce more complex concepts for automatic service synthesis or composition, an example of the latter being the "service recipes" in projects like Darwin [20]. Finally, infrastructures like IFLOW will strongly benefit from efforts like the XenoServer project [46], which proposes to embed servers in large-scale networks that will assist in deployment of global-scale services at a nominal cost.

Utility Driven Self-Regulation. Adaptation in response to change in environment or requirements has been a well-studied topic. The challenge of building distributed adaptive services with service-specific knowledge and composition functionalities is dealt with in [26]. Self-Adaptation in grid applications using the software-architectural model of the system is discussed in [28]. A radically different approach, similar to IFLOW, for self-adaptive network services is taken by [30] where the researchers propose a bottom-up approach, by embedding an adaptive architecture at the core of each network node. In contrast, IFLOW's self-regulation is based on resource information and user preferences, the latter expressed with flow-specific utility-functions. This utility-driven self-management is inspired by earlier work in the real-time and multimedia domains [41]. The specific notions of utility used in this paper mirror the work presented in [42] which uses utility functions for autonomic data-centers. Autonomic self-optimization according to business objectives is also studied in [43], but we differ in that we focus on the distributed and heterogeneous nature of system resources.

4. Software Architecture

Insights from our earlier work with the ECho pub-sub infrastructure have led us to structure IFLOW into the three software layers shown in Figure 2.

First, the *Control Layer* is responsible for accepting information flow composition requests, establishing the mapping of flow-graph vertices to the physical platform represented by the underlay, and handling reconfigurations. It has been separated from the messaging layer because it may need to implement different application-specific methods for flow-graph deployment and reconfiguration, each of which may be driven by a different utility-function. For instance, for pub-sub, the methods provided by this layer are used to implement publish-subscribe semantics, like those that require that all sources contributing to a logical subscription be notified when a new subscriber arrives. By providing basic methods for implementing such semantics, rather than integrating these methods into the messaging layer, IFLOW not only offers improved control performance compared to the earlier, integrated ECho pub-sub system developed in our work, but it also gives developers the freedom to implement different application-specific messaging semantics. Our current work has developed the information flow and pub-sub semantics. In ongoing work, we are creating transactional semantics and reliability guarantees like those used in industrial middleware for operational information systems [23].

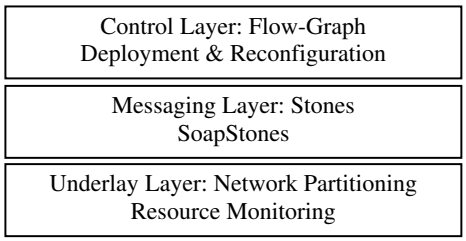

Figure 2. IFLOW Software Architecture

The second layer is the *Messaging Layer*, responsible for both data transport and the application of operators to data. It consists of an efficient messaging and operator module, termed 'Stones', and its web-service-enabled distributed extension, termed 'SoapStones'. A high performance implementation of information flows can make direct use of Stone functionality, using them to implement data and control plane functions, the latter including deploying new Stones, removing existing ones, or changing Stone behavior. This is how the ECho pub-sub system is currently implemented, essentially using additional 'Control Stones' to create the control infrastructure needed to manage the one-to-one connections it needs for high volume information exchanges. An alternative implementation of ECho now being realized with IFLOW, provides additional capabilities to our pub-sub infrastructure. The idea is to use IFLOW's overlays to replace ECho's existing one-to-one connections between providers and subscribers with overlay networks suitably mapped to underlays. Finally, the purpose of SoapStone is to provide ubiquitous access to Stone functionality, leveraging the generality of the SOAP protocol to make it easy for developers to implement new control protocols and/or realize the application-level messaging protocols they require. A concern not addressed in this paper but a subject of our future work are the high overheads of SoapStone (due to its use of the SOAP protocol). As a result, it is currently used mainly for initial flow-graph deployment and for similarly low-rate control actions.

The third layer is the *Underlay Layer*. It organizes the underlying hardware platform into hierarchical partitions that are used by the deployment infrastructure. The layer also implements scalable partition-level resource-awareness, with partition coordinators subscribing to resource information from other nodes in the partition and utilizing it to maintain the performance of deployed information flows. The underlays separation affords us with the future ability to add generic methods for dynamic resource discovery, underlay extension and contraction, and underlay migration. Our current work is focused on the underlay's efficient data transport across heterogeneous platforms [19].

Finally, the purpose of IFLOW, of course, is to provide an efficient basis for implementing a wide variety of application-specific messaging models. The following subsections describe in more detail the composition of the three layers.

4.1 Control Layer

-- Composition, Mapping, and Reconfiguration

The *Control Layer* offers the abstraction of an Information Flow-Graph to applications. It is responsible for mapping a specified flow-graph onto some known underlay. Deployment is based on the resource information supplied by the underlay layer and a function for evaluating deployment utility. The application can specify a unique utility-function local to a flow-graph, or a global

utility formulation can be inherited from the underlay layer. The control layer also handles the reconfigurations to maintain high utility for the deployed information flow. We now describe in detail the abstraction and functionality offered by this layer.

4.1.1 Information Flow-Graph

The information flow-graph is a collection of vertices, edges, and a utility-function, where vertices can be sources, sinks or flow-operators:

- A `source` vertex has a static association with a network node and has an associated data stream-rate. A source vertex can be associated with one or more outgoing edges.
- A `sink` vertex also has a static association with a network node. A sink vertex can have at most one incoming edge.
- An `operator` vertex is the most dynamic component in our abstraction because its association to any particular network node can change at runtime as the control layer reconfigures the flow-graph's deployment. Each operator is associated with a data resolution-factor (which is the ratio of the average stream output-rate to the average stream input-rate), an average execution-time, and an E-Code [2] snippet containing the actual operator code. An operator vertex can be associated with multiple incoming and outgoing edges.

The `utility` of a flow-graph is calculated using the supplied utility-function, which essentially contains a model of the system and is based on both application-level (e.g., user-priority) and system-level (e.g., delay) attributes. The function can be used to calculate the `net-utility` of a flow-graph mapping by subtracting the `cost` it imposes on the infrastructure from the utility-value. An example utility calculation model is shown in Figure 3, which depicts a system where end-to-end delay and the user-priority determine the utility of the system. The utility model 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".

The IFLOW framework also supports a `pin-down` operation, which statically associates an operator with a network node. Pin-down enables execution of critical/non-trivial operators on specialized overlay nodes and denotes the fact that certain operators cannot be moved (e.g., due to lack of migration support for their complex codes and states). Another interesting feature implemented in our framework is the support for parameterized 'tunable' operators, which enables remote modification of parameters associated with operator code. For example, a subscription operator might route updates based on a particular predicate, where the control layer supports remotely modifying the predicate at runtime.

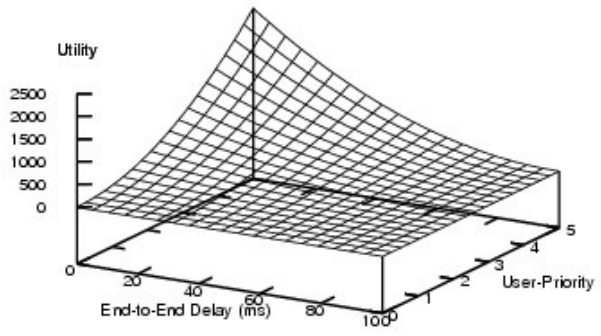

Figure 3. A Sample Utility Calculation Model

4.1.2 Flow-Graph Construction and Mapping

On close examination of the applications requiring information flow capabilities; we observe that there are two distinct classes of flows:

- *Basic information flows*
- *Semantic information flows*

Basic information flows arise in applications in which only the sources and sinks are known, and the structure of the data flow-graph is not specified. For example, in the pub-sub model, there exists no semantic requirement on the flow-graph. IFLOW can establish whichever edges and merge/split operators may be necessary, between publishers and subscribers. To accommodate this class of `basic' information flows, we have developed a novel graph construction algorithm, termed `InfoPath`. InfoPath uses the resource information available at the underlay layer to both construct an efficient graph and map the graph to suitable physical network nodes.

Semantic information flows are flows in which the data flow-graphs are completely specified, or at least have semantic requirements on the ordering and relationship between operators. For example, SQL-like continual queries over data streams specify a particular set of operations based on the rules of relational algebra. Similarly, in a remote collaboration application, application-specific operators must often be executed in a particular order to preserve the data semantics embedded in scientific workflows. Here, IFLOW takes the specified flow-graph and maps it to physical network nodes using the `PathMap` mapping algorithm. This algorithm utilizes resource information at the underlay layer to map an existing data flow-graph to the network in the most efficient way. The InfoPath and PathMap algorithms are described in detail in Section 5.

4.1.3 Reconfiguration

After the initial efficient deployment has been produced by PathMap or InfoPath, conditions may change, requiring the deployment to be reconfigured. IFLOW maintains a collection of configuration information, called the `IFGRepository` that can be used by the control layer when reconfiguration is necessary. As shown in Section 4.3, the underlay layer uses network-awareness to cluster physical nodes, and the IFGRepository is actually implemented as a set of repositories, one per underlay cluster. This allows the control layer to perform local reconfigurations using local information whenever possible. Global information is accessed only when absolutely necessary. Thus, reconfiguration is usually a low-overhead process.

The IFLOW framework provides an interface for implementing new reconfiguration policies based on the needs of the application. Our current implementation includes two reconfiguration policies: the `Delta Threshold Approach` and the `Constraint Violation Approach`. Both approaches take advantage of the IFGRepository and the resource information provided by the underlay layer to monitor the changes in the utility of a graph deployment. When the change in utility passes a certain threshold (in the `Delta` approach) or violates application-specific guarantees (in the `Constraint` approach), the control layer initiates a reconfiguration. The two reconfiguration approaches are described in detail in Section 5.4.

4.2 Messaging Layer

-- Stones, Queues, and Actions

The messaging layer of IFLOW is composed of communicating objects, called `Stones`, which are linked to create `data paths`. Stones are lightweight entities that roughly correspond to processing points in dataflow diagrams. Stones of different types perform data filtering, data transformation, multiplexing and de-multiplexing of data, and transmission of data between processes over network links. Application data enters the system via an explicit submission to a Stone, but thereafter it travels from Stone to Stone, sometimes crossing network links, until it reaches its destination. The actual communication between Stones in different processes is handled by the Connection Manager [36], a transport mechanism for heterogeneous systems, which uses a portable binary data format for communication and supports the dynamic configuration of network transports through the use of attributes. Each Stone is also associated with a set of `actions` and `queues`. Actions are application-specified handlers that operate on the messages handled by a Stone. Examples of actions include handlers that filter messages depending on their contents, and handlers that perform type conversion to facilitate message processing in Stones. Queues associated with Stones serve two purposes: synchronizing incoming events to a Stone that operates on messages coming from multiple Stones and temporarily holding messages when necessary during reconfiguration.

Stones can be created and destroyed at runtime. Stones can also be configured to offer different functionalities by assigning the respective actions at runtime. This permits Stones to be used as sources/sinks as well as for intermediate processing. Actions assigned to Stones can be both typed and untyped. When multiple typed actions are assigned to a single Stone, the type of the incoming event determines which action is applied. Some of the actions that can be assigned to Stones include:

- An `output action` causes a Stone to send messages to a target Stone across a network link.
- A `terminal action` specifies an application handler that will consume incoming data messages (as a sink).
- A `filter action` allows application-specified handlers that filter incoming data to determine if it will be passed to subsequent Stones.
- A `split action` allows the incoming messages to be sent to multiple output Stones. This is useful when the contents of a single link need to be sent along multiple data paths. The target Stones of a split action can be dynamically changed by adding/removing Stones from the split target list on the fly.
- A `transform action` specifies an action, which transforms data from one data type to another. These actions may be used to perform more complex calculations on data flows, such as sub-sampling, averaging, or compression.

Filter and transform actions are particularly noteworthy, as they allow the application to dynamically set the handler functions. The handler functions are specified in E-Code, a portable subset of the C language. Dynamic code generation [37] is then used to install and execute the handlers. Not only does this process facilitate dynamic reconfiguration, but it also permits the handlers to be run on heterogeneous platforms.

The design of Stones permits dynamic assignment of actions and presents a generic framework for messaging. It also allows dynamic configuration of message handling and transport, and hence offers a suitable base for network overlays.

4.2.1 SOAP-based Overlay Control

The control layer makes calls into the messaging layer to manage Stones. However, Stones are a general middleware component, and may be useful in other infrastructures too. In order to provide a convenient API that provides an abstraction of the messaging layer for both IFLOW and other frameworks, we have developed a web service front-end using SOAP. We call this API `SoapStone`. The overlay network created with Stones can thus be configured and managed through SOAP calls. The SOAP operations for overlay control have been merged with those used for configurations in the control layer, obviating the need for a separate SOAP server for the two layers.

The information flow-graph obtained from the control layer, along with the details of the mapping between the vertices of the flow-graph and the corresponding physical network nodes are used to send the appropriate SOAP calls to the corresponding nodes to create and manage Stones. Any reconfigurations necessitated by the higher layer during the course of execution can be transferred to the Stones through SOAP operations.

4.3 Underlay Layer

-- Network Partitioning and Resource Monitoring

The Underlay Layer is primarily 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). An example is shown in Figure 4. The advantage of organizing nodes in a hierarchy is that it simplifies maintenance of the partition structure, and provides a simplified abstraction of the underlying system, its administrative domains, and its resource characteristics to the upper layers. For example, when deploying a flow-graph, we can subdivide the data flow-graph to the individual clusters for further deployment.

We call the clusters *partitions*, although nodes in one partition can still communicate with those in another partition. Each node in a partition knows about the `costs` of paths between every pair of nodes in the partition. A node is chosen from each partition to act as the coordinator for this partition 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. Also, just as in the initial level, all coordinators at a particular level know the average minimum cost path to the other coordinator nodes that fall in the same partition at that level. In

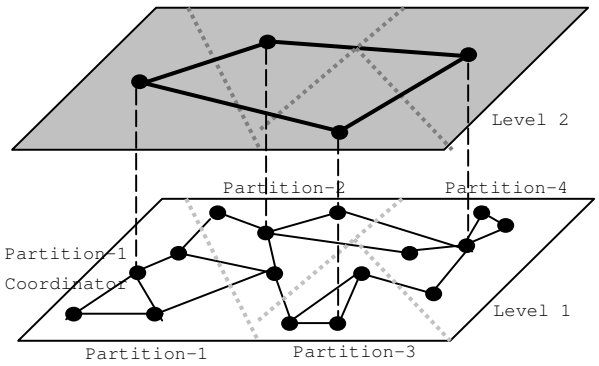

Figure 4. Hierarchical Network Partitioning

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 in each partition.

The underlay layer has several other responsibilities. First, the underlay layer is responsible for handling node `Join` and `Departure` requests. In our ongoing work, we are examining how to incorporate fault-tolerance into the underlay layer to handle node-failures. The underlay layer also implements a resource-monitoring module, using Stone-level data structures that contain per-node network and machine performance data. In particular, each coordinator maintains an `IFGRepository` of configuration and resource information for its partition. Our current implementation leverages the subscription-based monitoring capabilities from our previous work on the Proactive Directory Service [24] that supports pushing resource events to interested clients. At the control layer, the coordinator for a particular partition subscribes to resource information from the various nodes and intervening links in its partition, aggregates it, and responds to changing resource conditions by dynamically reconfiguring the information flow deployment.

5. Algorithms

-- InfoPath, PathMap, and Reconfiguration

We now describe in detail the various algorithms built into our framework. First, we present a formal model of our framework, and then discuss each algorithm.

5.1 Framework

The *information flow-graph* is represented as $G(V_g, E_g, U)$ with each vertex in V_g representing a source-node, a sink-node or an operator i.e.

$$V_g = V_g^{sources} \cup V_g^{sinks} \cup V_g^{operators}$$

The edges E_g in the information flow-graph represent the flow of information, and may span multiple intermediate edges and nodes in the underlying network. Finally, the utility-function U contains a formulation based on application-level and resource-level attributes to calculate the utility of any edge in the flow-graph, given the system conditions. The utility of the flow-graph U_G, can then be calculated as

$$U_G = \sum_{e \in E_g} U(e)$$

The underlying network is represented as a *network graph* $N(V_n, E_n)$ where vertices $v_{nj} \in V_n$ represent the actual physical nodes and the network connections between the nodes are represented by the edges $e_{nj} \in E_n$. 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.

We now formalize the *network-partitioning* scheme described in Section 4.3. Let

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

$n_{critical}^i$ = maximum number of nodes per partition at level i

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

Note that $v_{nj}^0 = v_{nj}$ 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 $n_{total}^i > n_{critical}$. 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. cost of 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:

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

and

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

In simple terms, 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.

5.2 InfoPath

-- Constructing the Information Flow-Graph

Problem: The `InfoPath` problem is to construct an appropriate path between the sources and sinks of the information flow-graph when no such path has been defined, and when all data emanating from the sources must be delivered to the sinks. The algorithm makes use of two standard operators, which merge or split the information flows. It instantiates a number of appropriate edges to maximize the net-utility (i.e., the difference of utility obtained from the deployed graph minus the cost it imposes on the system) derived by connecting sources to the sinks. More formally, we want to instantiate a number of merge/split operators $v_{gj} \in V_g^{operators}$ assigned to some $v_{ni} \in V_n$, and edges $e_{gj} \in E_g$ such that the resulting graph is a maximum net-utility graph for disseminating data from sources to the sinks.

Solution: A naive solution to routing information from sources to sinks would establish one-to-one connections. However, this approach can be very expensive. Instead, we exploit the hierarchical organization of nodes to construct a flow-graph that acts as an application-level multicast tree to efficiently disseminate updates. Towards this end, the `InfoPath` algorithm proceeds in the following manner. First, a set of sources and sinks ($V_g^{sources} \cup V_g^{sinks}$) is submitted as input to the top-level (say level t) coordinator along with a general utility formulation U. The sources and the sinks trickle down into their partitions at Level 1, the level at which the physical nodes reside, presenting each partition with a subset of initial sources and sinks. The partition-coordinator then finds the maximum *net-utility* path for connecting the subset of nodes using one-to-one connections. This is based on the observation that it is not too costly for InfoPath to establish one-to-one connections between the sources and the sinks *within* a single partition. Since the hierarchical partitioning algorithm clusters nodes according to network 'closeness', we assume that within a partition the data transmission costs are low. Then, the algorithm further condenses all the sources in a partition into a single virtual source using a merge operator, and similarly

creates a single virtual sink for all the partition's sinks using a split operator. The virtual source and sink resulting from each partition form the sources and sinks managed by the coordinator at the next level of the hierarchy. At that level, we again connect virtual sources and sinks using one-to-one connections, and this process proceeds recursively until level *t*-1. The InfoPath algorithm is used in the implementation of the Pub-Sub model described in Section 6.3.

5.3 PathMap

-- Scalable Placement of Flow-Operators

Problem: Given a flow-graph G, 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 vertices v_{gj} and v_{gk} is mapped to the network edges along the maximum net-utility path between the network nodes that v_{gj} and v_{gk} are assigned to. We define *net-utility*($M_{j\text{-}k}$) as the difference between the utility obtained from deploying the flow-graph edge $e_{gj\text{-}k}$ and the cost imposed by the deployment $M_{j\text{-}k}$ of $e_{gj\text{-}k}$ across v_{nj} and v_{nk} :

$$net\text{-}utility(M_{j-k}) = utility(e_{gj-k}) - \sum_{e_n \in M(e_{gj-k})} cost(e_n)$$

Using the above formulation, we can define the net-utility of flow-graph G, produced by a mapping M as:

$$net\text{-}utility(M) = \sum_{e_g \in E_g} (utility(e_g) - \sum_{e_n \in M(e_g)} cost(e_n))$$

For example, consider a cost function that is based on the amount of data transferred (i.e. bandwidth consumed). 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 cost of transferring data along the lowest-cost path between the vertices v_{ni} and v_{nj}. Our goal is to construct a mapping M of vertices that implies the maximum *net-utility* mapping between the edges E_g in G to edges E_n in N. We calculate net-utility as the difference between the utility achieved as a result of flow-deployment and the cost of utilizing the system's resources for such a deployment. For example, the costs of transferring data along different physical links differ. However, by paying the cost, certain guarantees (such as average delay experienced) are achievable. Rather than blindly minimizing the cost of deploying a flow-graph, we try to maximize the *net-utility* achievable.

Solution: The `PathMap` algorithm works as follows. The given information flow-graph $G(V_g, E_g, U)$ 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 hierarchical structure and limited size of partitions ensures that there are few nodes at this level, minimizing the expense of the exhaustive mapping. The net-utility for each assignment is calculated and the assignment with maximum net-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 partition 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, and operators are assigned to actual physical nodes. Because the graph is recursively deployed using the hierarchical partitioning structure, edges are kept inside a partition whenever possible, and whenever an edge must cross partitions a maximum net-utility path between the two partitions is chosen. PathMap is used in our implementation of collaborative visualization (Section 6.1) and operational information systems (Section 6.2).

5.4 Implemented Reconfiguration Policies

An overlay can be reconfigured in response to a variety of events, which are reported to the first-level cluster-coordinators by the monitoring component. These events include changes in network delays, available bandwidth, data-operator behavior, available processing capability, etc. Although most reconfigurations are localized within a partition, they can be expensive and temporarily disrupt the operation of the system. Therefore, it is impractical to respond to all such events reported by the monitoring component. The standard reconfiguration policies built into IFLOW aim to limit the number of reconfigurations while trying to maintain near-optimal performance. In addition to the two policies described below, applications can specify their own custom policies.

In the *Delta Threshold 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 in Section 7 demonstrate that it is effective in controlling reconfigurations.

The *Constraint Violation 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 performance. 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 currently, reconfigurations in IFLOW middleware are not lossless - some updates and state may be lost during the process. This is acceptable for those information flow applications that are able to tolerate some level of approximation and loss, as in case of the scientific collaborations we have been studying. In ongoing work we are examining how to model reconfiguration as a database-style transaction in order to achieve losslessness.

6. Case Studies

-- Application Implementation using IFLOW

This section describes the implementation of three messaging-applications built with IFLOW – a collaborative visualization application, an operational information system, and a pub-sub communication framework. The following subsections distill each of the above applications down to the five components - *sources, sinks, operators, edges* and a *utility-function*. Once the application model has been distilled, it is a matter of invoking the correct IFLOW primitives and implemented features to materialize the self-managing information flow application.

6.1 Collaborative Visualization

Our first application is the collaborative visualization of a molecular dynamics simulation, which is of interest to computational scientists in a wide variety of fields, from pure science (physics and chemistry) to applied engineering

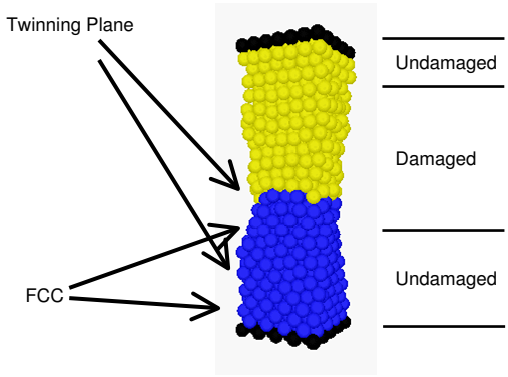

Figure 5. *Events of Interest*: Sample molecular dynamics data that shows different events of interest. For this single simulation of a block of copper being stretched, on the left we see attributes a physicist might want to highlight, while the right side shows the higher-level synthesis a mechanical engineer may want to see.

(mechanical and aerospace engineering). The visualization application is geared to deliver the events-of-interest (see Figure 5) to participating collaborators, formatted to suit the rendering capabilities at their ends. More details about this application are available in [18].

Application development for such a collaboration using IFLOW involves identifying sources – *simulation sites*, sinks – *scientists with rendering equipments*, operators – *filters for events-of-interest, formatting or downsizing of data to suit end-user capabilities*, edges – *representing the data-flow* and a utility function that encodes the requirement for *information-freshness* (low-delay), *low bandwidth-utilization* and *synchronized-delivery* (similar delays along various information flow-paths). Once the above parameters have been identified, an Information Flow-Graph can be constructed using IFLOW primitives and submitted to the underlying infrastructure for actual deployment using PathMap.

Deployment of an information flow requires existence of a bootstrapped middleware. IFLOW's underlay provides primitives that automatically initialize and organize the underlying nodes that desire to participate in the middleware. Once the participating nodes are up and running, the job of deploying the collaboration flow-graph is trivially simple – a call to the PathMap algorithm at any node in the infrastructure with the constructed flow-graph. Once the flow-graph has been mapped onto the infrastructure nodes, reconfiguration is taken care of by IFLOW in accordance with the supplied utility-function. An evaluation of the PathMap algorithm is presented in Section 7.1.3.

6.2 Operational Information System

This application is inspired by the use-case and scenarios provided to us by our long-standing collaborator, Delta Air Lines. An operational information system, as described in Section 2, provides continuous support for an organization's daily operations. We implement an information flow motivated by the requirement to feed overhead displays at airports with up-to-date information. The overhead displays periodically update the weather 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 selecting the weather information (which originates from a weather station) based on flight information, and other for combining the appropriate flight data (which originates from a central location like Delta's TPF facility) with periodic updates from airline counters that decide the waitlist order, etc. Thus, the three *sources* can be identified as – the weather information source, the flight information source, and the passenger information source. They are then combined using the operators to be delivered to the *sink* – the overhead display. A detailed discussion of the utility-driven deployment of such a flow-graph can be found in [14]. In Section 7.2 we evaluate IFLOW's realization of such a flow-graph against Delta's custom in-house middleware, DTMI.

6.3 Pub-Sub Implementation

To further substantiate the claim of IFLOW's generality and abstraction completeness, we have implemented a simple publish-subscribe messaging model using the IFLOW framework. In this implementation, messages are sent via `publishers` into a pub-sub `channel`, which may have zero or more `subscribers`. The subscribers may be on the same machine as the publisher or anywhere else in the network; however, their locations are immaterial to the publisher. Each subscriber receives only the messages sent to the channel to which it is subscribed.

The development of such a messaging model requires abstracting the publishers and subscribers as sources and sinks (respectively) in the information flow-graph; it also requires an application-dependent utility formulation that can drive deployment. The InfoPath algorithm, described earlier, provides the capability to establish an efficient information flow path between a given set of sources and sinks for message dissemination. Each pub-sub channel is assigned a globally unique identifier, essentially identifying the information flow-graph, or a part of it, stored in the IFGRepository of coordinators across the network. Once the information flow has been initially deployed on a bootstrapped infrastructure, and an identifier has been obtained for the deployment, the task of adding/removing participants can be handled as follows:

1. *Adding a publisher* requires identifying the cluster to which the publisher belongs, setting up one-to-one connections with the sinks in the cluster, and finally establishing a connection with the merge operator (virtual source for next level in the hierarchy) corresponding to this flow-graph.
2. Similarly, *adding of a subscriber* requires identifying the cluster to which the subscriber belongs, establishing intra-cluster one-to-one connections with the publishers, and subscribing to the virtual sink for out-of-cluster updates.

The task of removing participants can be implemented trivially at the cluster level and similar to additions requires no global exchange of messages. An evaluation of our pub-sub implementation is provided in Section 7.3.

7. Experiments

In this section, we report experiments that are designed to evaluate the performance of the IFLOW middleware. First, microbenchmarks examine specific features of our system, including an evaluation of the scalability achieved using the hierarchical partitioning at the underlay, the performance of the messaging layer compared to MPICH, and the performance of algorithms implemented with the control layer. We then present end-to-end performance comparisons between our emulation of

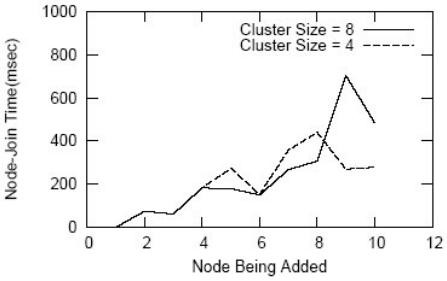

Figure 6. Time to Add N[th] Node to Underlay

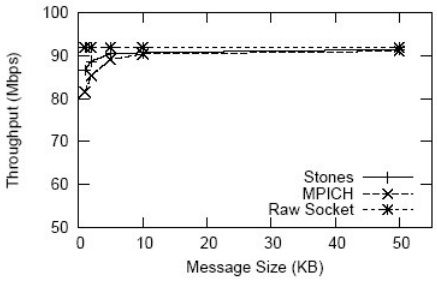

Figure 7. Throughput Using Stones

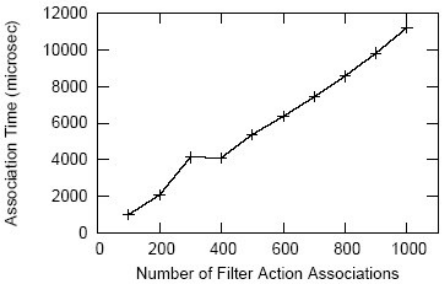

Figure 8. Time to Associate Filter Actions

Delta's middleware and DTMI, the in-house middleware currently being used by the organization. The idea is not to demonstrate superior IFLOW performance, since DTMI has richer facilities for interoperating with a large variety of applications and application API, but simply, to establish that IFLOW is competitive in terms of its performance behavior. Finally, we evaluate the performance of our implementation of the pub-sub communication model. Due to space constraints, we do not present performance results from our implementation of the collaborative visualization application. However, since this application uses the same IFLOW primitives, the performance is similar to the other applications.

Results from the performance measurements of the applications implemented using IFLOW corroborate our claim of IFLOW's generality. They show that our system offers *complete* and *general* abstractions for implementing distributed information flow applications, while offering a range of other desirable features like scalability, utility driven optimizations, etc.

7.1 Microbenchmarks

7.1.1 Underlay Layer: Scalability

The hierarchical partitioning of nodes at the underlay layer affect several aspects of the IFLOW middleware, including the addition/removal of network nodes from the underlay, the flow-deployment algorithm implemented at the control layer (evaluated in Section 7.1.3), and the add/remove facility for pub-sub participants (evaluated in Section 7.3). Figure 6 shows the time taken to add the N^{th} physical node to the middleware's underlay layer for cluster sizes 4 and 8. Addition of a node to the middleware with *N-1* nodes, using a naïve approach, should require interactions between the new node and the existing *N-1* nodes to determine resource information like inter-node delay etc. Such a naïve scheme would result in unacceptable node join times even for slightly large values of *N*. IFLOW overcomes this problem by limiting the partition sizes it maintains for its *tree-like* hierarchy. Thus, any node that joins the underlay finds its appropriate cluster, an $O(logN)$ operation, followed by a constant time operation to determine intra-cluster attributes, which in-turn will depend on the maximum cluster size. Results indicate a sub-linear increase in node-join times with increasing number of nodes – implying the much needed scalability.

7.1.2 Messaging Layer: Stone Performance

The following microbenchmarks are measured using a 2.8 GHz Xeon quad processor with 2MB cache, running Linux 2.4.20 smp as a server. The client machine used is a 2.0 GHz Xeon quad processor, running Linux 2.6.10 smp. Both the machines are connected by single-hop 100Mbps ethernet.

`Send/Receive Costs`: Stone's most significant performance feature is its use of the native data format on the sender side, coupled with dynamically generated unmarshalling code at the receiver to reduce the send/receive costs. 'send side cost' is the time between an application submitting data for transmission until the time at which the infrastructure invokes the underlying network 'send()' operation. 'receive side cost' represents the time between the end of the 'receive()'operation and the point at which the application starts to process the event. Since these costs are in the range of 0.005ms to 0.017ms (Table 1), the resulting overheads are very small compared to the typical round trip delays experienced in local area networks (about 0.1-0.3ms with a Cisco Catalyst 6500 series switch) and negligible for typical wide area round trip delays (50ms-100ms).

`Throughput Comparison against MPICH`: We also compare the throughput achieved for different message sizes using Stones to that of raw sockets and MPICH. Figure 7 shows that achieved throughput values closely follow the raw socket throughput for packet sizes exceeding 2KB, and are better than the values achieved using MPICH. This is very encouraging for IFLOW applications that target the high performance domain.

`Stones Instantiation Overheads`: The deployment of a flow-graph at the overlay layer consists of creating source, sink or filter Stones and associating suitable action routines to the Stones. The experiments reported in Table 2 show the small delays required for local Stone actions, including those via the SOAP interface (but not including the overhead of SOAP calls), making them suitable for supporting reconfigurations that require frequent Stone creation and deletions. The comparatively high cost for source Stone creation is due to format registration but this is done only once and hence is not a problem.

`Multiple Filter Association Times`: To facilitate operations to be performed on data flowing through the Stones, we would need to associate filter actions to the corresponding Stones (as mentioned in Section 4.2). Figure 8 shows the time

Table 1. Stones: Send & Receive Overheads

Message Size KB	**100**	**10**	**1**
Receiver Cost *µsec*	17.4	14.3	6.8
Sender Cost *µsec*	9.3	5.4	5.3

Table 2. Stone/SoapStone Microbenchmarks in *µsec*

Operation	**Stone**	**SoapStone**
Stone Create	0.89	1535.25
Stone Delete	0.03	1211.56
Source Create	1663.94	3005.27
Sink Create	8.31	1219.23

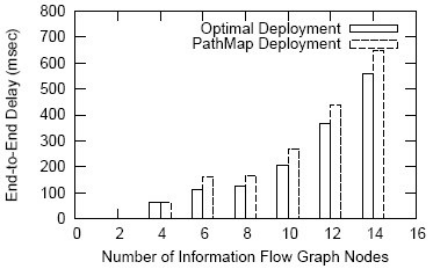

Figure 9. PathMap vs Optimal Deployment

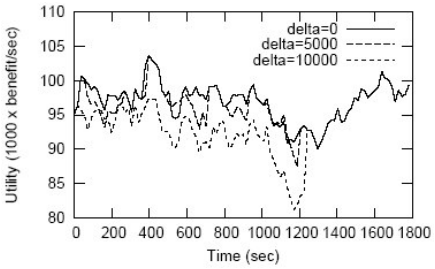

Figure 10. Utility Variation with Delta-Threshold Approach

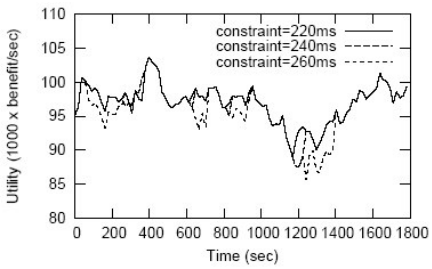

Figure 11. Utility Variation with Constraint-Violation Approach

involved in associating filter actions to a Stone. All associations are performed on the same Stone, but the destination Stones were changed. The graph shows a linear trend, indicating that filter Stones are suitable for large- as well as small-scale deployments.

7.1.3 Control Layer: Deployment & Reconfiguration

The GT-ITM internetwork topology generator [38] is used to generate a sample Internet topology for evaluating the performance of the control layer, in terms of deployment optimality and reconfiguration benefit. We use the transit-stub topology with 128 nodes for the ns-2 simulation, including 1 transit domain and 4 stub domains. 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 is composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen [39]. The ns-2 simulation is 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 to the underlay layer's resource monitoring infrastructure.

The first experiment is conducted to evaluate the effectiveness of the deployment routine, PathMap, in the control layer simulated over the above network. The PathMap algorithm tries to find a near-optimal deployment for a given flow-graph because finding an optimal deployment requires a brute-force search, which runs in exponential time. Hence, our algorithm aims to trade some optimality for better response-time. We parameterized our algorithm to use end-to-end delay as the optimization parameter; Figure 9 shows the achieved delay values for increasing number of information flow-graph nodes. We notice that the end-to-end delay using PathMap is very close to the one achieved using optimal (brute force) scheme. Thus, the PathMap algorithm hits the right balance between optimality and response-time, making it suitable for high-performance applications. The InfoPath algorithm is evaluated in Section 7.3 where it is used for establishing links between the pub-sub participants.

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 utility (with edge utility determined using the formulation $k*(c-delay)^2*bandwidth_{available}/bandwidth_{required}$, where k and c are constants) of a 10-node data flow-graph using the delta-threshold approach in the presence of network perturbations is shown in Figure 10. The rationale behind delta-threshold approach is that a reconfiguration is beneficial only when the benefits due to reconfiguration surpass the cost of reconfiguration. Hence, pursuing the optimal deployment for smaller gains in utility may not be the best approach. The delta-threshold approach aims to minimize the number of potentially lossy reconfigurations. We notice that even for a sufficiently large value of the threshold, the achieved utility closely follows the maximum achievable utility, but this is achieved with far fewer reconfigurations (1 with a threshold of 10000 as compared to 11 with a 0 threshold). Thus, an appropriate threshold value can be used to trade-off utility for a lower number of reconfigurations.

Figure 11 shows the variation of 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 trigger a reconfiguration when this bound is violated. The experiment is driven by real world requirements 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 note some resemblance in behavior between the delta-threshold approach and the constraint 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 violations is far easier than optimizing a general utility function. Self-optimization 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 graphs for achieving maximum utility.

7.2 Comparison with Delta's OIS Middleware

Delta Technology Messaging Interface (DTMI) is a messaging middleware developed by Delta Technology (DT) to support demanding enterprise-class applications. DTMI provides the foundations for building service-oriented architectures (SOA). Services (application logic exposed as an API) are deployed in server processes running on a cluster of heterogeneous machines. Each server process can host multiple services and a service can be deployed on multiple server processes. Services communicate by sending messages to each other. DTMI achieves high throughput in message passing by using low-latency System V message queues as the inter-process communication (IPC) mechanism for processes collocated on the same host while socket communication (over TCP) is used for message passing across machine boundaries.

This evaluation is conducted to substantiate our claim that IFLOW performs competitively against an industrial strength middleware. We use Emulab [40] to create a 10 node (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9) topology that was generated using GT-ITM. Links are 100Mbps and the inter-node delays are set between 1msec and 6msec. We then instantiate the flow-graph (three sources, two operators and a sink) described in Section 6.2 using both IFLOW and DTMI.

Table 3. Comparison of Processing & Propagation delays on IFLOW and DTMI

Event size (bytes)	Time taken for 10000 events (seconds)	
	IFLOW	*DTMI*
256	9.29	26.01
1024	10.43	27.63
5120	17.18	29.03
10240	27.33	34.62

Table 4. Time taken to Add/Remove a pub-sub participant

Time (*msec*)	16 Node	32 Node
Add	67.9	78.4
Remove	48.8	54.6

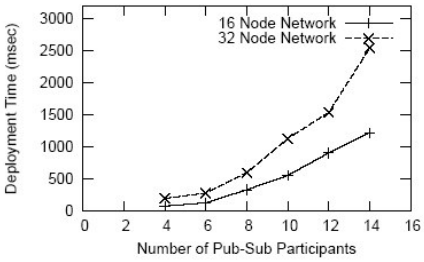

Figure 12. Pub-Sub deployment time

Experiments are then conducted to measure the time taken by the two realizations of the same flow-graph to process 10,000 events, for different event sizes. As Table 3 shows, IFLOW performs competitively with DTMI. In fact, DTMI is slower than IFLOW, but this is because DTMI provides more services, such as fault tolerance and sanity checks. Thus, while it is not fair to conclude that IFLOW is truly faster than DTMI, our results provide evidence that IFLOW performs competitively against an industrial strength middleware.

7.3 Evaluating the Pub-Sub Implementation

The next set of experiments again uses Emulab with 16 and 32 node (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9) topologies, generated with GT-ITM. Links are 100Mbps and the inter-node delays are set between 1msec and 6msec. The first experiment measures the time taken by our implementation of pub-sub (described in Section 6.3) to establish the links between the pub-sub participants. Our implementation uses the InfoPath algorithm that tries to establish a near-optimal update dissemination path between the participants using the information provided to it by the underlay. The time taken to deploy a pub-sub with varying number of participants on 16 and 32 node networks is shown in Figure 12. The low deployment times can be attributed to the bound on the partition size (8 in this case), and to the fact that link establishment between sub-groups of participants residing in different partitions can happen in parallel.

We also measure the time taken to add or remove a participant. Table 4 shows the time taken to perform add/remove operations over a 10 node pub-sub graph. For each topology, we report the average-time of removing a publisher and a subscriber. We again notice very small times for such operations as most of these operations involve changes that are limited to an underlay-partition.

8. Conclusion & Future Work

This paper presents the design, implementation, and initial experiences with IFLOW, a novel messaging framework that can be used to develop distributed applications that involve the acquisition, processing and dissemination of updates. IFLOW models such an application as an Information Flow-Graph, thus encompassing the ability to express the requirements of several messaging applications. This formalized notion proves equally useful for implementation of deployment and reconfiguration algorithms at the control layer. The Stones and queues at the messaging layer serve as the building blocks for the information overlays across the network, efficiently assisted by the underlay layer, which provides resource-aware system partitions. As part of our ongoing efforts, we are focusing on enriching IFLOW with semantics useful for an industrial middleware that is capable of supporting an Operational Information System, on improving the performance of SoapStones, and embedding fault-tolerance into the underlay layer.

References

[1] Gryphon: A robust pub-sub message broker, http://www.research.ibm.com/gryphon/

[2] G. Eisenhauer, F. Bustamante, K. Schwan, Event Services for High Performance Computing. Proceedings of HPDC-2000, Pittsburgh, USA.

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

[4] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. Proceedings of DEBS'02, Austria, July 2002.

[5] L. Opyrchal, et al. Exploiting IP Multicast in Content-Based Publish-Subscribe Systems. Proceedings of ACM/IFIP/USENIX Intl. Middleware Conference, 2000.

[6] S. Madden, M. Franklin, J. Hellerstein, W Hong. TinyDB: An Acqusitional Query Processing System for Sensor Networks. ACM Transactions on Database Systems, March 2005.

[7] M. Wolf, H. Abbasi, B. Collins, D. Spain, K. Schwan. Service Augmentation for High End Interactive Data Services. IEEE International Conference on Cluster Computing (Cluster 2005), Boston, Massachusetts, USA, September 27 - 30, 2005.

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

[9] 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. Proceedings of Very Large Databases Conference, Hong Kong, 2002.

[10] R. Koster, A. Black, J. Huang, J. Walpole, C. Pu. Infopipes for composing distributed information flows. Intl. Workshop on Multimedia Middleware, Canada, 2001.

[11] Y. Ahmad, U. Cetintemel. Network-Aware Query Processing for Distributed Stream-Based Applications.Proceedings of Very Large Databases Conference, Canada, 2004.

[12] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Proceedings of ICDE'03, India, 2003.

[13] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, K. Schwan. Resource-Aware Distributed Stream Management using Dynamic Overlays. 25th IEEE International Conference on Distributed Computing Systems (ICDCS), Columbus, Ohio, USA, 2005.

[14] V. Kumar, B. F. Cooper, K. Schwan. Distributed Stream Management using Utility-Driven Self-Adaptive Middleware. Proceedings of International Conference on Autonomic Computing (ICAC), Seattle, Washington, USA, 2005.

[15] M. Castro, P. Druschel, A-M. Kermarrec, A. Rowstron, SCRIBE: A large-scale and decentralised application-level multicast infrastructure, IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, No, 8, October 2002.

[16] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, A. Singh, SplitStream: High-bandwidth multicast in a cooperative environment, SOSP'03, New York, October, 2003.

[17] K. P. Birman, R. V. Renesse, J. Kaufman, W. Vogels. Navigating in the Storm: Using Astrolabe for Distributed Self-Configuration, Monitoring and Adaptation. Active Middleware Services 2003: 4-13.

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

[19] F. Bustamente, G. Eisenhauer, K. Schwan, P. Widener. Efficient Wire Formats for High Performance Computing. In Proceedings of Supercomputing 2000.

[20] A. Huang, P. Steenkiste. Building Self-configuring Services Using Service-specific Knowledge. 13th IEEE Symposium on High-Performance Distributed Computing (HPDC'04), IEEE, June 2004, Honolulu, Hawaii.

[21] Y. Wiseman, K. Schwan, P. Widener. Efficient End-to-End Data Exchange Using Configurable Compression. Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS), March 2004.

[22] Q. He, K. Schwan. IQ-RUDP: Coordinating Application Adaptation with Network Transport. Proceedings of HPDC-11, ACM/IEEE, July 2002.

[23] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, D. Amin. Operational Information Systems: An Example from the Airline Industry. WIESS 2000.

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

[25] G. Blair, L. Blair, V. Issarny, P. Tuma, A. Zarras. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. In Proceedings of Middleware 2000 IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing.

[26] A. Huang, P. Steenkiste. Building Self-configuring Services Using Service-specific Knowledge. 13th IEEE Symposium on High-Performance Distributed Computing (HPDC'04), IEEE, June 2004, Honolulu, Hawaii.

[27] C. D. Gill, et al. Integrated Adaptive QoS Management in Middleware: An Empirical Case Study. Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04).

[28] S. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, N. Hu. Software Architecture-Based Adaptation for Grid Computing. 11th IEEE Symposium on High-Performance Distributed Computing (HPDC'02), IEEE, July 2001, Edinburgh, Scotland.

[29] K. Ma, D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. Proceedings of the Supercomputing 2000 Conference, Dallas, Texas, 2000.

[30] N. Janssens, W. Joosen, P. Verbaeten, K.U.Leuven. Decentralized Cooperative Management: A bottom-up Approach, IADIS International Conference on Applied Computing, 2005.

[31] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke. Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment. Proc. 5th IEEE Symposium on High Performance Distributed Computing, pp. 562-571, 1997.

[32] B. Parvin, et al. DeepView: A Collaborative Framework for Distributed Microscopy. IEEE Conf. on High Performance Computing and Networking, 1998.

[33] J. E. Swan II, M. Lanzagorta, D. Maxwell, E. Kuo, J. Uhlmann, W. Anderson, H. Shyu, and W. Smith. A Computational Steering System for Studying Microwave Interactions with Space-Borne Bodies. Proceedings IEEE Visualization 2000.

[34] The Terascale Supernova Initiative is a project funded under the SciDAC program. http://www.phy.ornl.gov/tsi/

[35] LSC. http://www.ligo.org/. LIGO Scientific Collaboration.

[36] G. Eisenhauer. The Connection Manager Library. http://www.cc.gatech.edu/systems/projects/CM/cm.pdf, 2004.

[37] M. Poletto, D. Engler, and M. F. Kaashoek. tcc: A template-based compiler for c. Proceedings of the Workshop on Compiler Support for Systems Software, 1996.

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

[39] The Network Simulator – ns-2. http://www.isi.edu/nsnam/ns/

[40] Network Emulation Testbed. http://www.emulab.net/

[41] 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.

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

[43] 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.

[44] R. Grimm, T. Anderson, B. Bershad, D. Wetherall. A system architecture for pervasive computing. In Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the Pc: New Challenges For the Operating System, ACM Press, New York, NY, 177-182.

[45] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall. Programming for Pervasive Computing Environments. ACM Transactions on Computer Systems, January 2002.

[46] E. Kotsovinos, D. Spence. The XenoServer Open Platform: Deploying global-scale services for fun and profit. Poster, ACM SIGCOMM '03, August 2003.