Autonomic Information Flows

Karsten Schwan, Brian F. Cooper, Greg Eisenhauer, Ada Gavrilovska,
Matt Wolf, Hasan Abbasi, Sandip Agarwala, Zhongtang Cai,
Vibhore Kumar, Jay Lofstead, Mohamed Mansour,
Balasubramanian Seshasayee, and Patrick Widener

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{schwan, cooperb, eisen, ada, mwolf, habbasi, sandip, ztcai, vibhore,
lofstead, mansour, bala, pmw}@cc.gatech.edu

Abstract

Today's enterprise systems and applications implement functionality that is critical to the ability of society to function. These complex distributed applications, therefore, must meet dynamic criticality objectives even when running on shared heterogeneous and dynamic computational and communication infrastructures. Focusing on the broad class of applications structured as distributed information flows, the premise of our research is that it is difficult, if not impossible, to meet their dynamic service requirements unless these applications exhibit autonomic or self-adjusting behaviors that are 'vertically' integrated with underlying distributed systems and hardware. Namely, their autonomic functionality should extend beyond the dynamic load balancing or request routing explored in current web-based software infrastructures to (1) exploit the ability of middleware or systems to be *aware of* underlying resource availabilities, (2) dynamically and jointly *adjust* the behaviors of interacting elements of the software stack being used, and even (3) dynamically *extend* distributed platforms with enterprise functionality (e.g., network-level business rules for data routing and distribution),. The resulting vertically integrated systems can meet stringent criticality or performance requirements, reduce potentially conflicting behaviors across applications, middleware, systems, and resources, and prevent breaches of the 'performance firewalls' that isolate critical from non-critical applications.

This paper uses representative information flow applications to argue the importance of vertical integration for meeting criticality requirements. This is followed by a description of the AutoFlow middleware, which offers methods that drive the control of application services with runtime knowledge of current resource behavior. Finally, we demonstrate the opportunities derived from the additional ability of AutoFlow to enhance such methods by also dynamically extending and controlling the underlying software stack, first to better understand its behavior and second, to dynamically customize it to better meet current criticality requirements.

1 Introduction

Distributed information-intensive applications range from emerging systems like continual queries[BW01, CCC+02, KCC+05b], to remote collaboration[DT, LSC03] and scientific visualization[WCHS02], to the operational information systems used by large corporations[OEP+00]. A key attribute of these applications is their use in settings in which their continued delivery of services is critical to the ability of society to function. A case in point is the operational information system running the 24/7 operations of the airline partner with whom our research center has been cooperating[OEP+00]. Another example is the use of telepresence for remote medicine or diagnosis. Other well-known critical settings include the information flows in financial

applications, the online data collection, processing, and distribution in automotive traffic management, and more generally, the rich set of information capture, integration, and delivery services on which end users are increasingly reliant for making even routine daily decisions.

The objective of the AutoFlow project is to better meet the critical performance requirements of distributed information flow applications. In this context, multiple technology developments provide us with new ways of meeting these requirements. One is the ubiquitous use of middleware to extend application functionality across the distributed, heterogeneous communication and computational infrastructures across which they must run. Second, while it is not easy or desirable to rewrite applications to make better use of system and network infrastructures, middleware provides a basis on which it becomes possible to customize and extend underlying systems and networks to better meet the needs of the many applications written with these widely used software infrastructures. In other words, middleware can exploit the increasingly open nature of underlying systems and networks to migrate selected services 'into' underlying infrastructure. Third, the increasing prevalence of virtualization technologies, on computing platforms and in the network, is providing us with the technical means to better control or isolate extended from non-extended elements of the vertical software stacks used by applications and middleware, and to create 'performance firewalls' between critical vs. non-critical codes. An outcome of these developments is that vertical extension and the associated control across the entire extended software stack are possible without compromising a system's ability to simultaneously deliver services, critical and non-critical ones, to many applications and application components.

The AutoFlow project exploits these facts to create middleware and open systems infrastructure that jointly implement the following functionality to meet future applications' criticality and high performance needs:

- *'Vertical' and 'Horizontal' Agility* – The AutoFlow middleware presented in this paper uses information flow graphs as a precise description of the overlay networks used by critical applications' distributed information flows. Based on these descriptions, applications can dynamically create new services, which are then deployed by middleware as native binary codes to the machines where they are needed. The outcome for these high performance codes is 'horizontal agility', which is the ability to change at runtime both what data is streamed to which overlay nodes and where operations are applied to such data. A simple example for real-time scientific collaboration is the runtime augmentation of server functionality to better meet current client needs[WAC+05]. Additional benefits from using horizontal agility are described in Section 4.2. Furthermore, middleware can also migrate certain services 'vertically', that is, for suitable services, middleware can use dynamic methods for system and network extension to realize more appropriate service implementations than those available at application level. Experimental results demonstrating the benefits derived from such vertical service migration appear in Section 4.6. The importance of both horizontal and vertical agility is demonstrated with critical information flows for high performance and enterprise applications in Sections 4.3 and 4.6.

  An additional benefit derived from the general nature of *Information Flow Graphs* is their role as a uniform basis for creating the diverse communication models sought by applications, including publish-subscribe[SBC+98], information flows implementing continuous queries[BW01], and domain-specific messaging models like those used by our airline partner[Del].

- *Resource-Aware Operation* – The dynamic nature of distributed execution platforms requires applications to adjust their runtime behavior to current platform conditions. Toward this end, AutoFlow uses cross-layer 'performance attributes' for access to platform monitoring information. Such attributes are used by application- or model-specific methods that dynamically adapt information flows. Sample adaptation methods implemented for real-time exchanges of scientific data with AutoFlow's publish-subscribe communication model include (1) dynamic operator deployment[WAC+05] in response to changes in available processing resources and (2) runtime adjustments of parameterized operators to match data volumes to available network bandwidths[HS02]. For distributed query graphs, techniques for dynamically tuning operator behavior to maximize the utility of information flows with available

network resources are described in [KCC+05b]. Section 4.3 presents results documenting the benefits derived from the network-aware operation of AutoFlow applications.

- *Utility-Driven Autonomic Behavior* – End users of distributed information systems desire the timely delivery of quality information content, regardless of the dynamic resource behavior of the networks and computational resources used by information flows. To meet application needs, AutoFlow 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 autonomic functionality. A utility metric used in this paper uses application-stated importance values for sending different elements of a high volume, scientific information flow to best use the limited network bandwidth available across international network links.

- *Scalability through Hierarchical Management* – AutoFlow scales to large underlying platforms by using hierarchical techniques for autonomic management, as exemplified by its automatic flow-graph partitioning algorithm presented in [KCC+05b]. 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. Scalability results presented in this paper justify hierarchical management with microbenchmarks evaluating the deployment time for a publish-subscribe implementation using a hierarchical approach

The AutoFlow project leverages a multi-year effort in our group to develop middleware for high-end enterprise and scientific applications. As a result, our AutoFlow prototype uses multiple software artifacts developed in prior work. AutoFlow's resource- and network-awareness is supported by the monitoring methods described in [JPS+02].

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 [EBS00], as well as by its binary methods for data representation and runtime conversion for heterogeneous systems[BESW00]. Performance and scalability are due to a 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[KCC+05b, KCS05].

While leveraging previous work, the AutoFlow project makes several novel research contributions. First, the AutoFlow middleware 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 autonomic messaging models. Specifically, information flows are described by *Information Flow-Graphs*, consisting of descriptions of sources, sinks, flow-operators, edges and utility-functions. *Sources*, *sinks* and *flow-operators*, which transform and combine data streams, constitute the vertices of the flow-graph, while the *edges* represent typed information flows between vertices. Once a flow-graph has been described, its deployment (i.e., the mapping of operators to physical nodes and edges to network links) creates an overlay across the underlying physical distributed system. Automated methods for deployment and runtime reconfiguration are based on resource awareness functionality and on *utility-functions* that act as a vehicle for encoding user and application requirements. The information flow-graph serves as the abstraction visible to applications, and it also provides the concrete representation on top of which to construct domain-specific messaging models like publish-subscribe, for example[EBS00]. Second, in contrast to our previous work on pub-sub[EBS01], scalability and high performance for AutoFlow applications are attained by separating resource awareness functionality placed into AutoFlow *underlays*[KCC+05b] from the information flow abstractions in the *overlay,* and from the *control plane* used to realize different messaging models. Third, by using utility-driven self-regulation, AutoFlow flows deployed across heterogeneous, dynamic underlying distributed platforms can continuously provide high performance services to end user applications.

Experimental results presented in Section 4 demonstrate AutoFlow's basic capabilities. A low send/receive overhead, $5.45\mu sec$ and $14.37\mu sec$ for 10KB messages, respectively, 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, $0.89\mu sec$

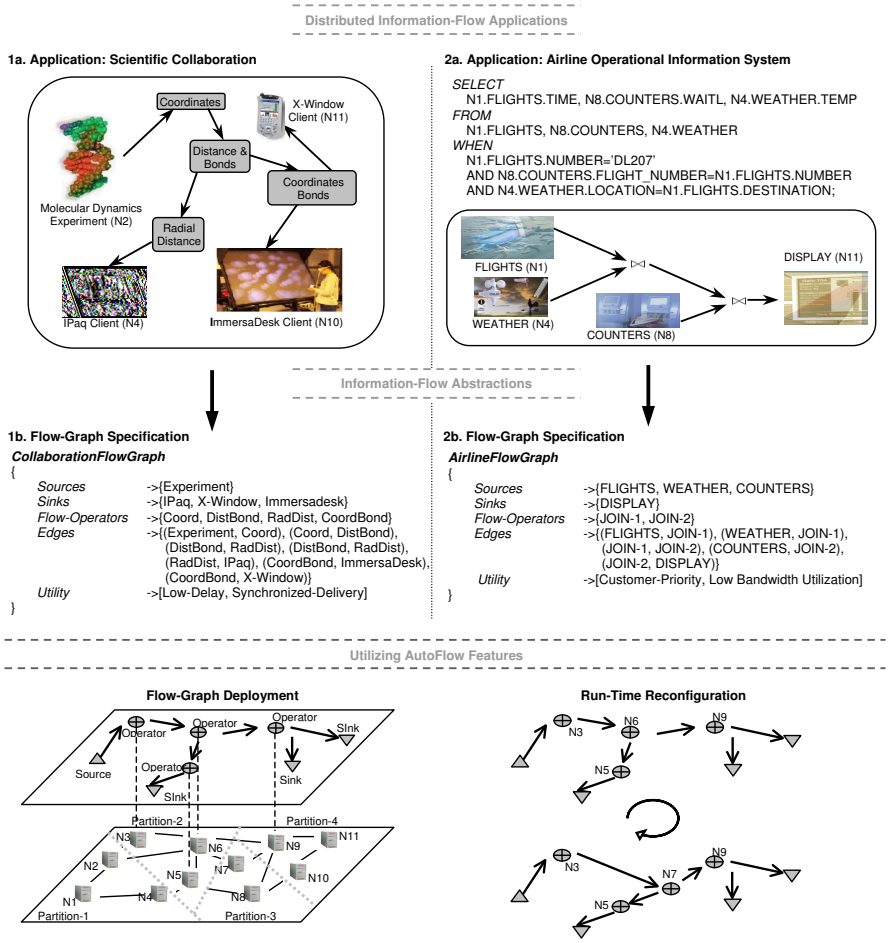


Figure 1: Implementation of different applications with AutoFlow

and $0.03\mu sec$ for local (i.e., in the same address space) operations. Experimental results in Section 4.3 demonstrate the importance of horizontal agility, using information about the structure of elements in data flows for real-time scientific collaboration to filter flows so as to best meet individual client needs. Section 4.6 uses enterprise data to illustrate the performance advantages gained from using a specialized 'information appliance' (a network processor able to interpret and process enterprise data) to manipulate application data. Finally, hierarchical deployment and reconfiguration is evaluated in Section 4.2.

The remainder of this paper is organized as follows. Section 2 briefly describes the two applications used to evaluate the AutoFlow approach – scientific collaboration and an airline's operational information system. The software architecture and the design of basic AutoFlow components are discussed in Section 3, explaining its layering and followed by a description of each layer's functionality. Section 4 presents an experimental evaluation of AutoFlow's claims. Section 5 presents related work. We conclude in Section 6 with a discussion of possible future directions.

2 Target Application Classes

In order to motivate this paper, we will first describe some sample applications in which autonomic information flows will be used. Figure 1 depicts two classes of information flow applications – real-time scientific collaboration and an airline's operational information system.

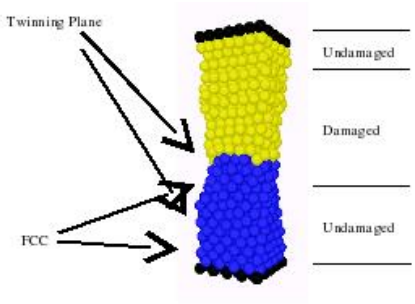


Figure 2: *Events of interest:* Sample molecular dynamics data that shows different events of interest. For this single simulation of a block of copper being stretched, on the left we see attributes a physicist might want to highlight, while the right side shows the higher-level synthesis a mechanical engineer may want to see.

2.1 Collaborative Visualization

Our first example application is the collaborative visualization of a molecular dynamics (MD) simulation. We note that MD simulations are of interest to computational scientists in a wide variety of fields, from pure science (physics and chemistry) to applied engineering (mechanical and aerospace engineering). Our particular visualization application is geared to deliver the events-of-interest (see Figure 2) to participating collaborators, formatted to suit the rendering capabilities at their ends. More details about this application are available in [WCHS02].

Collaborative real-time visualization uses 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, predictable delivery latency is important since there may be end-to-end control loops, as when one collaborator drives the annotation of data for other collaborators. Finally, autonomic methods can exploit a number of quality/performance tradeoffs, as end users may have priority needs for certain data elements, prefer consistent frame rates to high data resolution (or vice versa), etc. Thus, this online visualization application has rich needs for autonomic behaviors.

2.2 Operational Information System

The other 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 [OEP+00] for a description of our earlier work with this company). An operational information system 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 the weather station) based on flight information, and the 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.

Here, SQL-like operators 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

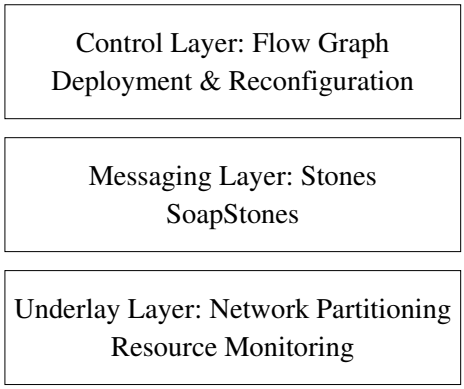


Figure 3: AutoFlow Software Architecture

deliver high utility to the enterprise. Utility-driven autonomic methods for managing event flows may take into account, for example, that a request pertaining to seat-assignment for a business-class customer may be given a higher priority 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. A detailed discussion of the utility-driven deployment of such a flow-graph can be found in [KCS05],

The remainder of this paper will describe in more detail how autonomic methods in AutoFlow support the dynamic behaviors of applications like these. Briefly, the main contributions of the AutoFlow architecture 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. 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[KCS05]. 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 monetary benefits. Scientific applications may use them to embed suitable quality/performance tradeoffs via dynamically tunable operators, such as image down-sampling or compression[WS04].

3 Software Architecture

Insights from our earlier work with the ECho publish-subscribe infrastructure have led us to structure AutoFlow into the three software layers shown in Figure 3.

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 must be able implement different application-specific methods for flow-graph deployment and reconfiguration, each of which may be driven by a different utility-function. By providing basic methods for implementing such semantics, rather than integrating these methods into the messaging layer, AutoFlow 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 alternative messaging semantics. In ongoing work, for example, we are creating transactional semantics and reliability guarantees like those used in industrial middleware for operational information systems.

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, remove existing ones, or changing Stone behavior. This is how the ECho pub-sub system is currently implemented, essentially using additional message exchanges 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 AutoFlow provides additional capabilities to our pub-sub infrastructure. The idea is to use AutoFlow'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. Not addressed in this paper but subject of our future work are the relatively 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, and higher rate control actions are implemented directly with the Stone infrastructure.

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. Its separation affords us with the important ability to add generic methods for dynamic resource discovery, underlay growth and contraction, underlay migration, and the vertical extension of the underlay 'into' the underlying communication and computation platforms.

The focus of this paper, of course, is the autonomic functionality in AutoFlow. Toward this end, we next describe the control layer functions used for runtime reconfiguration of AutoFlow applications.

3.1 Control Layer– *Composition, Mapping, and Reconfiguration*

The *Control Layer* implements the abstraction of an Information Flow-Graph, and 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 reconfigurations to maintain high utility for a deployed information flow.

3.1.1 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 [EBS00] 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. A sample utility calculation model is shown in Figure 4, which depicts a system where end-to-end delay and the user-priority determine the utility of the

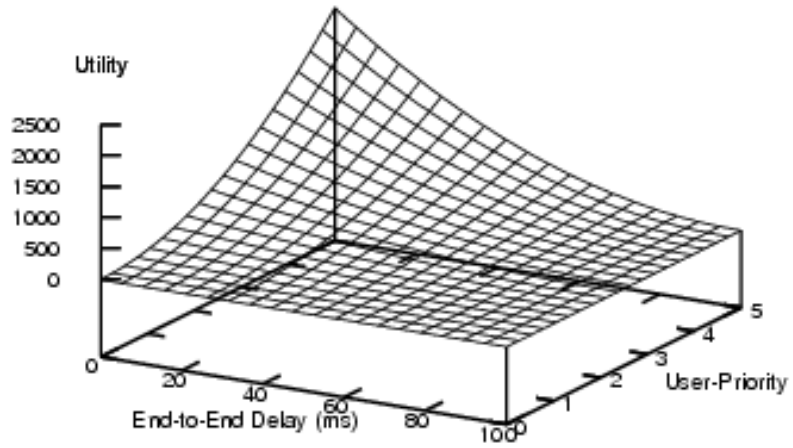


Figure 4: A Sample Utility Calculation Model.

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 AutoFlow framework also supports a `pin-down` operation, which statically associates an operator with the network node. Pin-down enables execution of critical/non-trivial operators on specialized overlay nodes and/or it simply 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 into the 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.

3.1.2 Flow-Graph Construction and Mapping

On close examination of the applications requiring information flow capabilities, we observe that there are two distinct class 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. AutoFlow can establish whichever edges and merge/split operators may be necessary, between publishers and subscribers. This class of 'basic' information flows is accommodated by 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 relationships between operators. For example, SQL-like continual queries over data streams specify a particular set of operations based on the laws 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 work flows. Here, AutoFlow 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 [KCC+05b, KCC+05a].

3.1.3 Reconfiguration

After the initial efficient deployment has been produced by PathMap or InfoPath, conditions may change, requiring the deployment to be reconfigured. AutoFlow maintains a collection of configuration information, called the `IFGRepository`, that can be used by the control layer when reconfiguration is necessary. The underlay 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 AutoFlow 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 [KCC+05a].

A rich set of methods for runtime adaptation controls information flows without reconfiguring their links or nodes. These methods manage the actual data flowing across overlay links and being processed by overlay nodes. Two specific examples are presented and evaluated in Section 4 below: (1) the adaptation of the data produced by a scientific data visualization, to meet utility requirements that capture both the quality and the delay of the data received by each visualization client, and (2) improvements in end-to-end throughput for high rate enterprise data flows by 'early' filtering of less important data at an AutoFlow-extended network interface attached to an AutoFlow host.

3.2 Messaging Layer– *Stones, Queues, and Actions*

The messaging layer of AutoFlow 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 [Eis04], 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 the 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 then be configured to offer different functionalities by assigning the respective actions. 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).

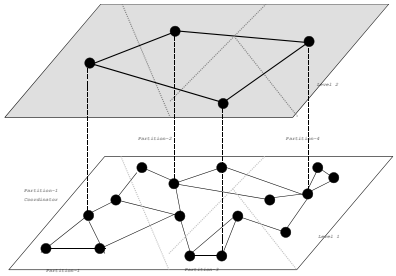


Figure 5: Hierarchical Network Partitioning.

- A `filter action` allows application-specified handlers that filter incoming data to determine whether it should 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 must 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.
- A `transform action` converts data from one data type to another. These actions may be used to perform more complex calculations on a data flow, such as sub-sampling, averaging, or compression.

Filter and transform actions are particularly noteworthy, as they allow the application to dynamically set handler functions. Handler functions are specified in E-Code, a portable subset of the C language. Dynamic code generation [MRF97] 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. Alternative static implementations of filters and transformers and less portable dynamic methods using DLLs or similar mechanisms are also available, to accommodate complex stone processing.

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

3.2.1 SOAP-based Overlay Control

The control layer must make calls into the messaging layer to manage stones. However, stones are a general middleware component, and may be useful in other infrastructures, an example being the overlay-based implementation of GridFTP described in [CEH+05]. For a convenient API that provides an abstraction of the messaging layer for both AutoFlow 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 enacted upon the affected stones through SOAP operations.

3.3 Underlay Layer – *Network Partitioning and Resource Monitoring*

The Underlay Layer maintains 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 combina-

tion of bandwidth and delay). An example is shown in Figure 5. The organization of nodes in a hierarchy simplifies maintenance of the partition structure, and it provides an 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 other partitions. 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 into the same partition at that level. 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 in each partition.

An important set of underlay responsibilities for autonomic computing is its support for online resource monitoring. To deal with node failures or the addition and deletion of new machines, the underlay layer has functions that handle node `Join` and `Departure` requests. The 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 [BWS02], which supports pushing relevant resource events to interested clients. In addition, performance attributes are used to describe information about the underlying platforms captured by instrumented communication protocols, by active network bandwidth measurements, and by the system-level monitoring techniques described in [JPS+02]. At the control layer, the coordinator for a particular partition subscribes to resource information from various nodes and intervening links in its partition, aggregates it, and responds to changing resource conditions by dynamically reconfiguring the information-flow deployment.

Stones and actions directly support 'horizontal' agility, The monitoring support described in the previous paragraph provides system-level resource information to middleware layers. However, none of these abstractions permit the 'vertical' agility needed for critical applications. A key attribute of underlays, therefore, is that they can extend 'into' the underlying communication and computational platforms. One model for this extension is described in [KGSS05], where the `C-Core` runtime captures the resources available across both general purpose host processors and the specialized communication cores on a host/communication co-processor pair or more generally, on future heterogeneous multicore platforms. For platforms like these, the underlay extended with `C-Core` (1) permits the dynamic creation, deployment, and configuration of services onto those cores that are best suited for service execution (e.g., hosts vs. network processors), and (2) it also monitors and exports the necessary resource utilization and configuration state needed for making appropriate dynamic deployment decisions. In the current design, `C-Core` provides to the EVPath middleware constructs termed *adaptation triggers*. The application and middleware use these constructs to specify services suitable for mapping to the underlying network. The `C-Core` infrastructure is responsible for determining, at runtime, the best processing contexts for running such services. By explicitly specifying the application-level services that may be suitable for running 'in' the network, the SPLITS compilation support associated with `C-Core` can statically generate the appropriate representations of such services for network-level execution (i.e., use a different set of code generators) [Gav04].

Specific examples that demonstrate the utility of such 'vertical' agility are evaluated in Section 4. The first example is drawn from the Delta OIS application. It demonstrates the benefits of deploying filtering handlers 'closer' to the network, at the network processor level. The filters extract from the data stream only those Delta events containing information for flights out of the Atlanta airport. Those events are further translated into appropriate formats which can be exchanged with external caterers. Both the filtering handler and the data translation handler can be implemented for network-processor or host-resident execution. Experiments demonstrates that the network near execution of such filtering actions can result in close to 30% performance improvement compared to filter executions on host machines.

The second example further demonstrates the utility of dynamic vertical reconfiguration. Consider the

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 1: Stones: Send and Receive Overheads.

imaging server described in the SmartPointer application, which customizes the image data to match the end-users interests. Our earlier results [Gav04] have demonstrated that networking platforms such as the IXP network processors are capable of performing manipulations of OpenGL image data at gigabit rates. However, depending on current loads on the general purpose computational node (i.e., the host), and the concrete ratio of original image size vs. derived image size, the ability of the host or the communications processor to execute the image manipulation service vary significantly. As a result, it is necessary to monitor platform runtime conditions such as processing loads, and application parameters, such as image sizes or cropping coordinates. Based on such monitoring, one should then dynamically reconfigure the deployment of the imaging service from a version where all image manipulation is performed at the computational host, to another one where the original image is passed to the communications hardware when then performs all additional image cropping and transmission operations.

4 Experimental Evaluation

Experiments are designed to evaluate the autonomic concepts used in the AutoFlow middleware: agility, resource-aware operation, and utility-driven behavior. First, microbenchmarks examine some specific features of the AutoFlow middleware, including the performance of the messaging layer compared to the well-known MPICH high performance software and the performance of algorithms implemented with the AutoFlow control layer. We also evaluate the performance of our implementation of the pub-sub communication model. Second, we examine the use of utility functions in evaluating the need to reconfigure an overlay network in the face of changing network resource availability, a form of horizontal agility. Next, we examine several forms of resource-aware operation, including the ability to adjust an application's own bandwidth requirements in response to network resource changes, further adapting application transmission in the context of lower-level network information (such as reported round-trip times (RTTs). Lastly, the importance of vertical agility is demonstrated with measurements attained on a host/attached IXP platform, emulating future heterogeneous multi-core systems.

4.1 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 machines are connected via single-hop 100Mbps ethernet.

`Send/Receive Costs`: AutoFlow'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 cost. '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 (see Table 1), the resulting overheads are quite 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 6 shows that achieved throughput values

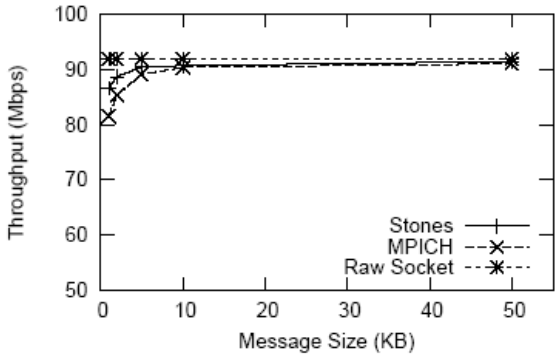


Figure 6: Throughput using Stones

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

Table 2: Stones/SoapStone Microbenchmarks in *microsec*.

closely follow the raw socket throughput for packet sizes exceeding 2KB, and are almost equal to the values achieved using MPICH. This is very encouraging for AutoFlow applications that target the high performance domain.

`Stones Instantiation Overheads`: The deployment of a flow graph at the overlay layer consists of creating sources, 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 overheads 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 an artifact of the use of PBIO. In this case, PBIO registers the source data format information with the third-party format server. This is a non-local operation, but it occurs only once per process per data format.

`Multiple Filter Association Times`: To facilitate operations to be performed on data flowing through stones, filter actions must be associated with corresponding stones (as mentioned in Section 3.2). Figure 7 shows the times involved in associating filter actions to a stone. All associations are performed on the same stone, but the destination stones are changed. The graph shows a linear trend, indicating that filter stones are suitable for large-scale as well as small-scale deployments.

4.2 Utility-based Reconfiguration

The GT-ITM internetwork topology generator[ZaB96] 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 [Pro]. The simulation is carried out for 1800 seconds and snapshots capturing end-to-end delay between directly connected nodes were taken every

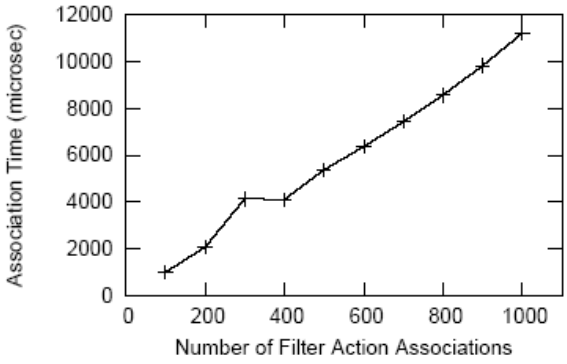
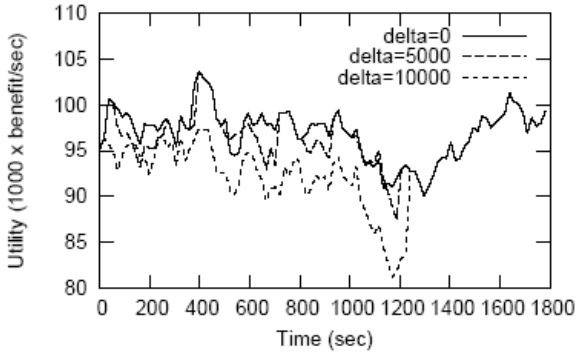
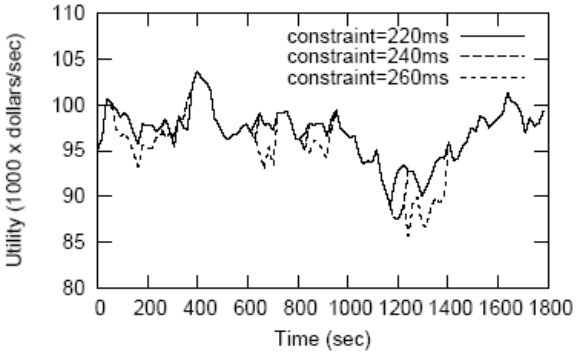


Figure 7: Time to Associate Filter Actions



(a) Utility Variation with Delta-Threshold Approach



(b) Utility Variation with Constraint-Violation Approach

Figure 8: Utility Variation with different algorithms.

5 seconds. These are then used as inputs to the underlay layer's resource monitoring infrastructure.

Concerning the use of utility for controlling the configuration of distributed overlay networks, it is interesting to compare the performance of two self-optimization approaches: the delta threshold approach, and the constraint violation approach. The change in utility (where edge utility is determined using the formulation $k \star (c - delay)^2 \star bandwidth_{available} \star bandwidth_{required}$) of a 10-node data flow graph using the delta-threshold approach in the presence of network perturbations is shown in Figure 8a. The rationale behind the delta-threshold approach is that a reconfiguration is beneficial only when the benefits accrued over time 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 note that even for a sufficiently large value of 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 8b 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 require-

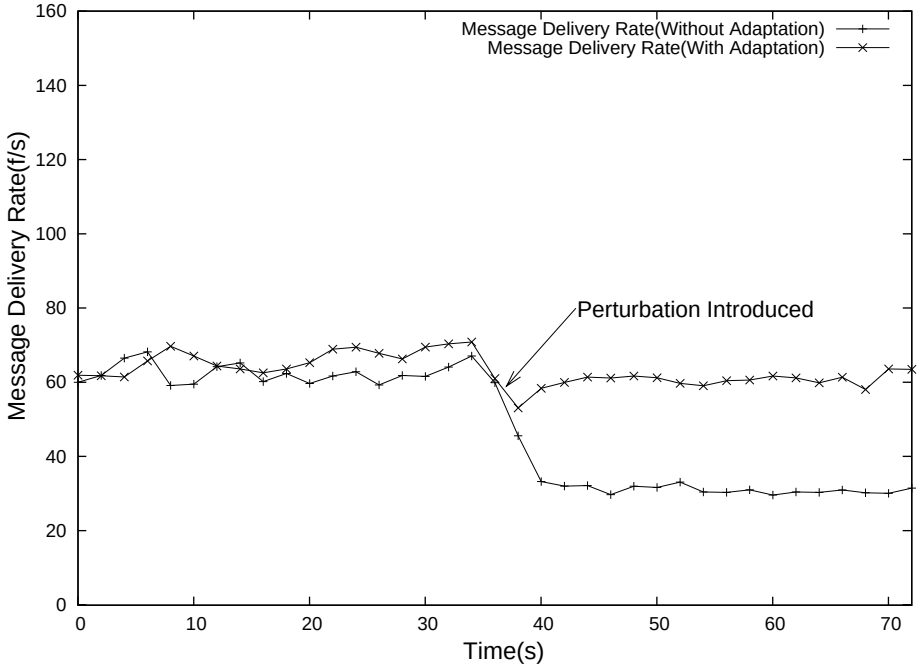


Figure 9: Adaptive Downsampling (ORNL link)

ments 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.

4.3 Adaptive Downsampling in Congested Networks

In real-time collaboration, one cannot unduly reduce the rates at which data is provided to end users, since that may violate timeliness guarantee or may cause different end users to get 'out of sync' with respect to the data they are jointly viewing. A solution is to deploy application-level data filters to downsample the actual data being sent prior to submitting it to the network transport. These filters can regulate the traffic imposed on the underlying network by 'pacing' application-level messages to effectively reduce congestion and maintain better message delivery rates. In contrast to multimedia systems that use statically defined filters specialized for that domain [FGC+97], the AutoFlow middleware's API permits clients to dynamically define and then deploy exactly the filters they wish, when and if they need them (also see [EBS00] for more detail). Furthermore, using performance attributes, filters can be controlled to deliver the best quality data permitted by current network conditions.

The experimental results in Figure 9 demonstrate the necessity and effectiveness of middleware-level adaptation through dynamic data downsampling. Here, large cross traffic (250Mbps) is injected as a controlled perturbation into the link from the machine named **isleroyale** at Georgia Tech to the machine named **cruise** at ORNL. The network bottleneck is at Georgia Tech's edge router. Permitting the client to characterize the subset of data most important to it, the client installs a data filter at the server side when congestion occurs,

and henceforth receives only 'essential' (i.e., as defined by the deployed filter) data at satisfactory speeds. The specific data downsampler used in these experiments removes data relating to visual objects that are not in the user's immediate field of view. That is, the client transfers the current position and viewpoint of the user to the filter (i.e., using attributes), at the server side these values are used to determine what data set the user is currently watching, and that information is then used to transfer appropriately downsampled data to the client. The result is a consequent reduction in the network bandwidth used for data transmission, thereby speeding up data transmission.

This experiment demonstrates the utility of network-initiated data downsampling for maintaining high data rates for limited network bandwidths. By giving end users the ability to define their own filters for implementing data downsampling, similar methods can be applied to other applications, as shown by past work on real-time applications [RS99, SLA02], and with services that implement general rather than application-specific compression methods [WS04].

4.4 Adaptive Downsampling with Knowledge of Congestion Window Size

An interesting aspect of filter/protocol interaction present in the previous experiment is that service-level downsampling actions in response to bandwidth reductions always 'lag behind' introduced cross traffic (i.e., when cross traffic is first introduced and when cross traffic is released). The cause is the delay experienced by the bandwidth measurement method used in those experiments [MJ04]. To cope with issues like these, the AutoFlow architecture is designed to support multiple methods of bandwidth measurement and/or to combine the use of such methods with monitoring information extracted directly from the communication protocols being used (i.e., using performance attributes).

The next experiments demonstrate the importance of supporting multiple means for autonomic systems to assess underlying network behavior, by performing middleware-level downsampling using information about the current congestion window size and RTT exported by the IQ-RUDP protocol. Congestion window size is exported via the CM_CWND performance attribute, and RTT is exported via the CM_RTT attribute. Traffic behavior is adapted as follows: when the congestion window size reaches half of its previous value, that fact is interpreted as congestion, resulting in data being sent at half of its previous speed (i.e., $bw = cwnd * MSS/RTT$, where bw is the TCP bandwidth, $cwnd$ is the congestion window size, and MSS is the maximum segment size). The result is significantly improved reaction time compared to using more rigorous bandwidth measurement methods, as shown in Figures 10 and 11. In this experiment, 25Mbps cross traffic is injected as a controlled perturbation. During the time when cross traffic is injected/released, the maximum normalized deviation of frame rate and targeted frame rate (Max{|frame rate - targeted frame rate|/targeted frame rate}) is 19.4 percent with pure bandwidth-based adaptive downsampling (see Figure 10). In comparison, it is improved to 6.03 percent when adaptive downsampling is performed with knowledge of congestion window size (see Figure 11).

4.5 Coordinated Adaptation in Slow Start

A well-known issue with the TCP-based transmission of large data volumes is the potential loss of available network bandwidth due to TCP slow start [KHR02]. The next experiment demonstrates how coordinating the actions of middleware-level filters with those of the network protocol can reduce slow start effects. We again export congestion window size from the transport layer.

Consider the incidence of slow start behavior, where congestion window size continues to increase until some timeout value is reached (as in TCP Tahoe) or until three duplicate ACKs are received (as in TCP Reno). It is well-known that this can result in overly large congestion windows and therefore, reduced network throughput. Consider the Netlab-based experiment shown in Figure 12. In this experiment, performance attributes are used to export IQ-RUDP's congestion window size and loss rate. The filter being used interprets a continuing increase in window size coupled with a zero packet loss rate as an indication of slow start. During such a phase, the service and DAL first measure effective network bandwidth. If bandwidth measurements become stable, then the interpretation is that the measured bandwidth is eventually attained

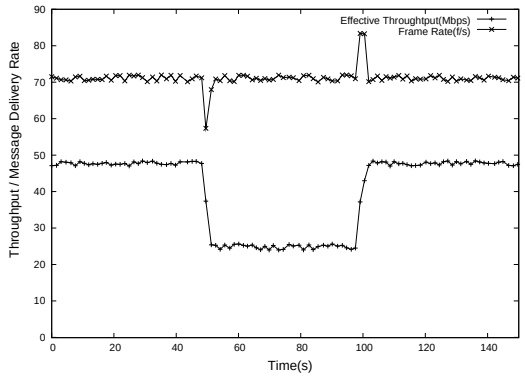


Figure 10: Downsampling Adaptation

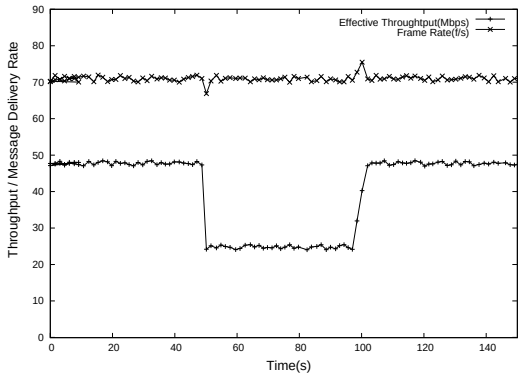


Figure 11: Downsampling Adaptation with knowledge of the congestion window size

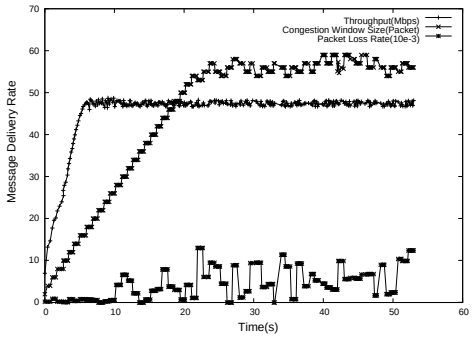


Figure 12: Un-coordinated Slow Start

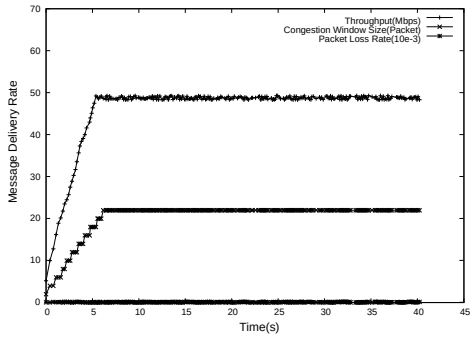


Figure 13: Coordinated Slow Start

by TCP[RMD04], which means that the middleware can safely set the protocol's congestion window to a value corresponding to this bandwidth, thereby effectively stopping its slow start phase. In the AutoFlow architecture, this task is performed with the CM_CWND_MAX attribute. The protocol also interprets this window size value as a threshold, which means that it will not increase its congestion window beyond threshold size even if loss rates remain small (i.e., are less than 0.005). To leverage any additional bandwidth available from the network, therefore, middleware continues to measure available bandwidth, using probing packets. If bandwidth increases are detected, then a new window size (and threshold) is provided to the protocol. At the same time, the protocol independently deals with decreases in bandwidth it observes, thereby retaining its property of TCP friendliness.

The utility of middleware-level bandwidth management as described above is demonstrated in the experiments shown in Figures 12 and 13. Here, the average loss rate between times 10s and 50s is 0.0 percent with coordinated slow start (see Figure 13), as compared 0.4 percent with normal slow start (see Figure 12). The average throughput during this period is improved from 47.3Mbps to 48.6Mbps. Normalized standard deviation of throughput is decreased from 0.0095 to 0.0063.

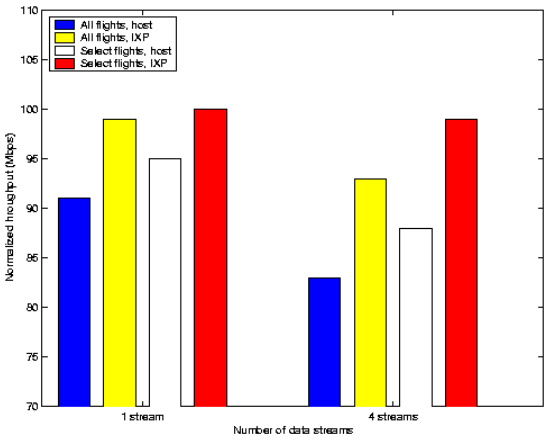


Figure 14: Data Translation/Extraction Performance on network coprocessor (IXP) or host.

4.6 Vertical Agility

This set of experiments illustrates the importance of enabling vertical system agility, that is, the ability of AutoFlow applications to adapt at runtime not only the nodes on which certain actions run, but also 'where' on such nodes action execution is carried out. Experiments are conducted using a cluster of eight Dell 530s with dual 1.7GHz Xeon processors running Linux 2.4.18, outfitted with Radisys ENP2611 IXP2400-based programmable communication co-processors (i.e., network processors, NPs), and interconnected via 1Gbps and 100Mbps links. We have evaluated the viability of executing various application-level services on the IXP NP as compared to host nodes only (i.e., going directly to the host's ethernet link, without an IXP in the path), as well as the performance levels that can be achieved. These evaluations are carried out by implementing different services with handlers that execute jointly on hosts and on their attached IXP2400s.

The results in Figure 14 illustrate the importance of offloading certain data translation services from application components executed on standard host onto attached programmable network interconnect cards. The data streams used in this experiment are generated with sequences of demo-replays of representative business data (i.e., data collected for the airline OIS). The application components executed on the cluster nodes use our version of the RUDP protocol on top of raw sockets [HS02]. The same protocol is used on the IXP microengines. We use the IXP2400 NP attached to the host via its PCI interface to emulate such programmable communication cores. The results represent the performance levels attainable for host- vs. IXP-based execution of rule chains that translate data into the appropriate format (bars marked 'all flights'), or translate the data and extract certain information currently needed by the application (bars marked 'select flights'). Results demonstrate that the in-network execution of these services results in improved performance, primarily due to CPU offloading and because load is removed from the host's I/O and memory subsystems. In addition, some unnecessary data copying and protocol stack traversals are avoided. Additional results on the utility of execution of application-level services on communications platforms such as the IXP network processor and vertical runtime adaptations appear in [GSK04, GS05].

While it may be intuitive that executing filtering functionality 'as early as possible', at the network interface, can result in improved service quality, the 'vertical' reconfiguration of other services is more sensitive to current workload or platform resources. The results in Figure 15 compares the host's vs. the IXP's ability to crop 1MB OpenGL-produced images against application-provided bounding boxes.

The first conclusion from these experiments is that even non communications-related service components can be considered for vertical deployment 'into' the platform, with this example showing performance gains reaching up to 40%. This gap increases with the amount of additional load on the host's processing contexts. Hence, performance improvements can be observed if services are redeployed on the platform

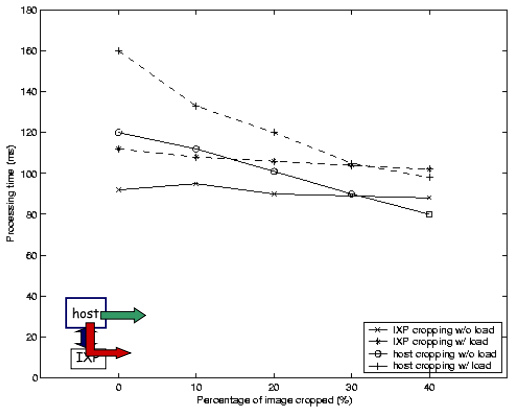


Figure 15: Host vs. IXP performance in cropping incoming images.

overlay, particularly since our related work already demonstrates that such reconfigurations have negligible overheads [GKSS05].

Our second conclusion is that middleware must consider carefully certain application-specific service parameters when determining the deployment context for such data cropping. Specifically, when cropping is implemented on the host, the CPU crops the image and sends a (possibly much) smaller image to the network device (via the PCI interface), thereby reducing image transfer time. In the IXP implementation, the host transfers the entire image to the NIC, which limits the IXP's cropping performance to essentially the throughput of the PCI interface between host and IXP. Hence, as the cropping window size decreases, the performance of the host implementation starts increasing, whereas the performance of the IXP implementation is dictated by the data rates delivered from the PCI interface and does not change significantly. We note, however, that as with graphics processors associated with host CPUs [IBM05], future heterogeneous multi-core platforms are not likely to experience PCI-based performance limitations.

5 Related Work

Publish-Subscribe Infrastructures & Application-Level Multicast. Pub-sub middleware like IBM's Gryphon [SBC+98], ECho[EBS00], ARMADA[ABD+99] and Hermes [PB02] 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[OAA+00], AutoFlow (like Gryphon[SBC+98]) uses application-level overlays for efficient information dissemination. Other systems that implement application-level message multicast include SCRIBE[CRKD02] and SplitStream [CDK+03], both focused on peer-to-peer systems. AutoFlow 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 AutoFlow 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[MFHH05], STREAM [BW01], Aurora[CCC+02], and Infopipes[KBH+01] 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[AC04, SHCF03, KCC+05b]. AutoFlow 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, AutoFlow facilitates the use of application-specific operators that implement desired runtime quality/performance tradeoffs. AutoFlow 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[WCHS02], remote visualization [MC00], the use of immersive systems across the network[FGN+97], and by programs like the Terascale Supernova Initiative[DT]. 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, running additional analyses, and sometimes even directly running simulations through computational steering[ILM+00] or by control of remote instruments[PTC98]. 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. AutoFlow 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[GABW00]. Tools like one.world[GDL+02] provide an architecture for simplifying application development in such environments. While AutoFlow'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 AutoFlow to manage evolving data-sources, as done in systems like Astrolabe[BvRKV03], which has the capability to self-configure, monitor and adapt a distributed hierarchy to manage evolving data-sources. An interesting generalization of AutoFlow 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[HS04]. Finally, infrastructures like AutoFlow will strongly benefit from efforts like the XenoServer project[KS03], which proposes to embed servers in large-scale networks that will assist in deployment of global-scale services at a nominal cost. Accord[LP05] is a framework for autonomic applications that focusses on object-based *autonomic elements* that are managed via sets of interaction rules. A composition manager assists in automatically composing sets of objects when then manage themselves via rule interactions. Accord's more declarative approach to composition fills the same role as the InfoPath's hierarchical configuration/reconfiguration algorithms (Section 4.2). Its rule-based behaviors are a more formal representation of the filter-based AutoFlow behavior. Autonomia[DHX+03] is Java-based effort to create a general framework consisting of self-managed objects, including XML-based control and management policy specifications, a knowledge repository and an Autonomic Middleware Service that handles the autonomic run-time behaviour.

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 [HS04]. Self-Adaptation in grid applications using the software-architectural model of the system is discussed in [CGS+02]. A radically different approach, similar to AutoFlow, for self-adaptive network services is taken by [JJVL05] where the researchers propose a bottom-up approach, by embedding an adaptable architecture at the core of each network node. In contrast, AutoFlow'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[KCS98], and the specific notions of utility used in this paper mirror the work presented in [WTKD04] which uses utility functions for autonomic data-centers. Autonomic self-optimization according to business objectives is also studied in [AGL+04], but we differ in that we focus on the distributed and heterogeneous nature of distributed system resources.

Vertical Agility in Exploiting Network Coprocessors

The utility of executing compositions of various protocol- vs. application-level actions in different pro-

cessing context is already widely acknowledged. Examples include splitting the TCP/IP protocol stack across general purpose processors and dedicated network devices, such as network processors, FPGA-based line cards, or dedicated processors in SMP systems [BLW02, RMM+03], or splitting the application stack, as with content-based load balancing for an http server [AAP+05] or for efficient implementation of media services [RAW03]. Similarly, in modern interconnection technologies, network interfaces represent separate processing context with capabilities for protocol off-load, direct data placement, and OS-bypass [ZBF05, SWP02]. In addition to focusing on multi-core platforms, our work differs from these efforts by enabling and evaluating the joint execution of networking and application-level operations on communications hardware, thereby delivering additional benefits to distributed applications.

6 Conclusions and Future Work

The AutoFlow project leverages a multi-year effort in our group to develop middleware for high-end enterprise and scientific applications and multiple software artifacts developed in prior work, including Dproc[JPS+02], PBIO[EBS00], ECho[EBS01], SplitStream[Gav04], IQ-RUDP[HS02], IFlow[KCC+05b] and SmartPointer[WCHS02]. This paper has examined the autonomic abilities and and application benefits resulting from the combined techniques represented by the AutoFlow middleware, demonstrating that it has sufficient base efficiency to be used in high-performance environments and that applications can use it to demonstrate both horizontal and vertical agility to improve or maintain performance in the context of changing resource availability. We demonstrated the benefits of resource-aware adaptation of application behavior including the ability to adjust an application's own bandwidth requirements in response to network resource changes, further adapting application transmission in the context of lower-level network information.

In ongoing work, we are further enriching the AutoFlow middleware to include support for lossless reconfiguration, fault tolerance at the underlay layer, improving the the performance of SoapStones, developing extended heuristics for mapping and remapping of overlay networks to available computation and communication resources, managing the evolution of distributed systems over time, and further exploiting both network coprocessors and potentially in-network processing resources.

References

[AAP+05] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha. Design, Implementation and Performance of a Content-Based Switch. In *Proc. of INFOCOM 2000*, 2005.

[ABD+99] T. Abdelzaher, M. Bjorklund, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, P. Marron, A. Mehra, and T. Mitton et al. ARMADA Middleware and Communication Services. *Real-Time Systems Journal*, 16:127–153, 1999.

[AC04] Y. Ahmad and U. Cetintemel. Network-aware query processing for distributed stream-based applications. In *Proceedings of Very Large Databases Conference*, 2004.

[AGL+04] S. Aiber, D. Gilat, A Landau, N Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In *Proceedings of the International Conference on Autonomic Computing, ICAC-2004*, 2004.

[BESW00] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient Wire Formats for High Performance Computing. In *Proc. of Supercomputing 2000*, Dallas, TX, November 2000.

[BLW02] Florian Braun, John Lockwood, and Marcel Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, January/February 2002.

[BvRKV03] Kenneth P. Birman, Robbert van Renesse, James Kaufman, and Werner Vogels. Navigating in the storm: Using astrolabe for distributed self-configuration, monitoring and adaptation. In *Proceedings of the Workshop on Active Middleware Services*, 2003.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[BWS02] Fabian E. Bustamante, Patrick Widener, and Karsten Schwan. Scalable directory services using proactivity. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, 2002.

[CCC+02] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams - a new class of data management applications. In *Proceedings of the Conference on Very Large Databases*, 2002.

[CDK+03] Miguel Castro, Peter Druschel, Ann-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *Proc. of 18th Symposium of Operating Systems Principles (SOSP-18)*, Bolton Landing, NY, 2003.

[CEH+05] Zhongtang Cai, Greg Eisenhauer, Qi He, Vibhore Kumar, Karsten Schwan, and Matthew Wolf. Iq-services: Network-aware middleware for interactive large-data applications. *Concurrency & Computation. Practice and Exprience Journal*, 2005.

[CGS+02] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Joao Sousa, Bridget Spitznagel, and Peter Steenkiste. Software architecture-based adaptation for grid computing. In *Proceedings of High Performance Distributed Computing (HPDC-11)*, July 2002.

[CRKD02] Miguel Castro, Antony Rowstron, Anne-Marie Kermarrec, and Peter Druschel. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, 20(8), 2002.

[Del] Delta Technologies Delta Airlines. Delta technologies messaging interface (dtmi). private communication.

[DHX+03] Xiangdong Dong, Salim Hariri, Lizhi Xue, Huoping Chen, Ming Zhang, Sathija Pavuuluri, and Soujana Rao. Autonomia: An autonomic computing environment. In *Proceedings of the 2003 IEEE International Performance, Computing and Communications Conference*, April 2003.

[DT] DOE-TSI. Terascale supernova initiative. `http://www.phy.ornl.gov/tsi`.

[EBS00] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. Event services for high performance computing. In *Proceedings of High Performance Distributed Computing (HPDC-2000)*, 2000.

[EBS01] Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. *ACM SIGOPS*, 35(2):7–20, April 2001.

[Eis04] Greg Eisenhauer. The connection manager library. `http://www.cc.gatech.edu/systems/projects/CM/cm.pdf`, 2004.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of SOSP*, 1997.

[FGN+97] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the i-way high performance distributed computing experiment. In *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, pages 562–571, 1997.

[GABW00] Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A system architecture for pervasive computing. In *In Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, 2000.

[Gav04] Ada Gavrilovska. *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors*. PhD thesis, Georgia Institute of Technology, 2004.

[GDL+02] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. *ACM Transactions on Computer Systems*, January 2002.

[GKSS05] Ada Gavrilovska, Sanjay Kumar, Srikanth Sundaragopalan, and Karsten Schwan. Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications. In *15th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05)*, Skamania, WA, 2005.

[GS05] Ada Gavrilovska and Karsten Schwan. Addressing Data Compatibility on Programmable Networking Platforms. In *Proc. of Symposium on Architectures for Networking and Communications Systems (ANCS'05)*, Priceton, NJ, 2005.

[GSK04] Ada Gavrilovska, Karsten Schwan, and Sanjay Kumar. The Execution of Event-Action Rules on Programmable Network Processors. In *Proc. of Workshop on Operating Systems and Architectural Support for the On-Demand IT Infrastructure (OASIS'04), in conjunction with ASPLOS-XI*, Boston, MA, 2004.

[HS02] Qi He and Karsten Schwan. IQ-RUDP: Coordinating Application Adaptation with Network Transport. In *Proceedings of High Performance Distributed Computing*, July 2002.

[HS04] An-Cheng Huang and Peter Steenkiste. Building self-configuring services using service-specific knowledge. In *Proceedings of the 13th IEEE Symposium on High-Performance Distributed Computing (HPDC'04)*, July 2004.

[IBM05] IBM. STI cell processor: New disclosures to jumpstart creation of cell-based applications beyond gaming. http://www.ibm.com/chips/power/splash/cell, Aug 2005.

[ILM+00] 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 spaceborne bodies. In *Proceedings of IEEE Visualization 2000*, 2000.

[JJVL05] Nico Janssens, Wouter Joosen, Pierre Verbaeten, and K. U. Leuven. Decentralized cooperative management: A bottom-up approach. In *Proceedings of the IADIS International Conference on Applied Computing*, 2005.

[JPS+02] J. Jancic, C. Poellabauer, K. Schwan, M. Wolf, and N. Bright. dproc - Extensible Run-Time Resource Monitoring for Cluster Applications. In *Proceedings of International Conference on Computational Science*, 2002.

[KBH+01] Ranier Koster, Andrew Black, Jie Huang, Jonathon Walpole, and Calton Pu. Infopipes for composing distributed information flows. In *Proceedings of the ACM Multimedia Workshop on Multimedia Middleware*, October 2001. `http://www.cc.gatech.edu/projects/infosphere/papers/acmmm_koster.pdf`.

[KCC+05a] Vibhore Kumar, Zhongtang Cai, Brian F. Cooper, Greg Eisenhauer, Karsten Schwan, Mohamed Mansour, Balasubramanian Seshasayee, and Patrick Widener. Iflow: Resource-aware overlays for composing and managing distributed information flows. Submitted to Eurosys-2006, Leuven, Belgium, 2005.

[KCC+05b] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2005.

[KCS98] R. Kravets, K. Calvert, and K. Schwan. Payoff Adaptation of Communication for Distributed Interactive Applications. *Journal of High Speed Networks*, Jul 1998.

[KCS05] Vibhore Kumar, Brian F. Cooper, and Karsten Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of the International Conference on Autonomic Computing*, 2005.

[KGSS05] Sanjay Kumar, Ada Gavrilovska, Karsten Schwan, and Srikanth Sundaragopalan. C-Core: Using Communication Cores for High Performance Network Services. In *Proc. of 4th Int'l Conf. on Network Computing and Applications (IEEE NCA05)*, Cambridge, MA, 2005.

[KHR02] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proceedings of ACM SIGCOMM*, Aug. 2002.

[KS03] Evangelos Kotsovinos and David Spence. The xenoserver open platform: Deploying global-scale services for fun and profit. Poster, ACM SIGCOMM '03, August 2003.

[LP05] Hua Liu and Manish Parashar. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics*, 2005. Special Issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty, IEEE Press.

[LSC03] LSC. `http://www.ligo.org/`. LIGO Scientific Collaboration., 2003.

[MC00] K. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network status. In *Proceedings of Supercomputing 2000*, 2000.

[MFHH05] S. Madden, M. Franklin, J. Hellerstein, and W Hong. Tinydb: An acqusitional query processing system for sensor networks. *ACM Transactions on Database Systems*, March 2005.

[MJ04] C. Dovrolis M. Jain. Ten Fallacies and Pitfalls in End-to-End Available Bandwidth Estimation. In *Proceedings of ACM Internet Measurements Conference*, 2004.

[MRF97] Poletto M, Engler D R, and Kaashoek M F. tcc: A system for fast, flexible, and high-level dynamic code generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI97)*, pages 109–121, June 1997.

[OAA+00] Lukasz Opyrchal, Mark Astley, Joshua S. Auerbach, Guruduth Banavar, Robert E. Strom, and Daniel C. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Proceedings of the Middleware Conference*, pages 185–207, 2000.

[OEP+00] Van Oleson, Greg Eisenhauer, Calton Pu, Karsten Schwan, Beth Plale, and Dick Amin. Operational information systems - an example from the airline industry. In *First Workshop on Industrial Experiences with System Software*, pages 1–10, San Diego, CA, October 2000.

[PB02] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *Proc. of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002.

[Pro] VINT Project. The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[PTC98] B. Parvin, J. Taylor, and G. Cong. Deepview: A collaborative framework for distributed microscopy. In *Proceedings of the IEEE Conf. on High Performance Computing and Networking*, 1998.

[RAW03] S. Roy, J. Ankcorn, and Susie Wee. An Architecture for Componentized, Network-Based Media Services. In *Proc. of IEEE International Conference on Multimedia and Expo*, July 2003.

[RMD04] R.Prasad, M.Jain, and C. Dovrolis. Socket Buffer Auto-Sizing for High-Performance Data Transfers. *Journal of Grid Computing*, 2004.

[RMM+03] Greg Regnier, Dave Minturn, Gary McAlpine, Vikram Saletore, and Annie Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. In *Proc. of HotI 11*, 2003.

[RS99] D.I. Rosu and K. Schwan. FARACost: An Adaptation Cost Model Aware of Pending Constraints. In *Proceedings of IEEE RTSS*, Dec. 1999.

[SBC+98] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. In *International Symposium on Software Reliability Engineering '98 Fast Abstract*, 1998.

[SHCF03] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of ICDE'03*, 2003.

[SLA02] Lui Sha, Xue Liu, and Tarek Abdelzaher. Queuing Model Based Network Server Performance Control. In *Proceedings of Real-Time Systems Symposium*, Dec. 2002.

[SWP02] P. Shivam, P. Wyckoff, and D.K. Panda. Can User Level Protocols Take Advantage of Multi-CPU NICs? In *Int'l Parallel and Distributed Processing Symposium*, 2002.

[WAC+05] Matthew Wolf, Hasan Abbasi, Ben Collins, David Spain, and Karsten Schwan. Service augmentation for high end interactive data services. In *IEEE International Conference on Cluster Computing (Cluster 2005)*, September 2005.

[WCHS02] Matt Wolf, Zhongtang Cai, Weiyun Huang, and Karsten Schwan. Smart Pointers: Personalized Scientific Data Portals in Your Hand. In *Proc. of Supercommputing 2002*, November 2002.

[WS04] Yair Wiseman and Karsten Schwan. Efficient End to End Data Exchange Using Configurable Compression. In *Proceedings of International Conference on Distributed Computer Systems*, Mar. 2004.

[WTKD04] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing, ICAC-2004*, 2004.

[ZaB96] Ellen W. Zegura and Ken Calvert andS. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, March 1996.

[ZBF05] Xiao Zhang, Laxmi N. Bhuyan, and Wu-Chun Feng. Anatomy of UDP and M-VIA for Cluster Communications. *Journal on Parallel and Distributed Computing*, 2005.