JECho - Supporting Distributed High Performance Applications with Java Event Channels

Dong Zhou, Karsten Schwan, Greg Eisenhauer, Yuan Chen
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
{zhou, chwan, eisen, yuanchen}@cc.gatech.edu

Abstract

This paper presents JECho, a Java-based communication infrastructure for collaborative high performance applications. JECho implements a publish/subscribe *communication paradigm, permitting distributed, concurrently executing sets of components to provide interactive service to collaborating end users via* event channels*. JECho's efficient implementation enables it to move events at rates higher than other Java-based event system implementations. In addition, using JECho's* eager handler *concept, individual event subscribers can dynamically tailor event flows to adapt to runtime changes in component behaviors and needs, and to changes in platform resources.*

JECho has been used to build distributed collaborative scientific codes as well as ubiquitous applications. Its event interface and eager handler mechanism have been shown flexible and in some scenarios, critical to the successful implementations of such applications. This paper's micro-benchmarks demonstrate that, with optimizations and customizations of the runtime system and the object transport layer, TCP-based reliable group communication in Java can reach good performance levels. These benchmark results also suggest that it is viable to use JECho to build large-scale, high-performance event delivery systems. JECho's implementation is in pure Java. Its group-cast communication layer is based on Java Sockets, and it also runs in some embedded environments that currently lack standard object serialization support.

1. Introduction

End users of high performance codes increasingly desire to interact with their complex applications as they run, perhaps simply to monitor their progress, or to perform tasks like program steering[8][9], or to collaborate with fellow researchers using these applications as computational tools. For instance, in our own past research, we have constructed a distributed scientific laboratory with 3D data visualizations of atmospheric constituents, like ozone, and with parallel computations that simulate ozone distribution and chemistries in the earth's atmosphere [4][10]. While an experiment is being performed, scientists collaborating within this laboratory may jointly inspect certain outputs, may create alternative data views on shared data or create new data streams, and may steer the simulations themselves to affect the data being generated. Similarly, the Hydrology Workbench[11] created by NCSA researchers uses a Java-based visualization tool, termed VisAD[3], to permit end users to view data produced by the running model or from previous generated model files. Finally, for meta-computing environments, researchers have created and are developing the

Access Grid[1] framework and, in related work, domain-specific 'portals' for accessing and using computations that are spread across heterogeneous, distributed machines.

Our group has been developing both C/C++- and Java[5]- based middle-ware targeted at such high performance interactive applications. In this context, the need to work with Java is evident from our interactions with end users. For example, in a 'Design Workbench' application we are jointly developing with end users in Mechanical Engineering at Georgia Tech, users are expecting to use web browsers to remotely inspect ongoing experiments. They use both physical views captured by cameras watching production machines and logical views derived from data captured from running materials simulations. In addition, in discussions surrounding the Chemical Engineering Workbench being developed at MIT[1], end users would even like to interact with their running simulations from the shop floor, via Palmtops or sub-notebook devices, using wireless communication media. In this context, Java-based programs are easily deployed. Finally, with Habanero[25], VisAD and specialized visualization engines like Povray[18], there is a plethora of Java-based collaboration and visualization tools available from other researchers and from industry, of which researchers need to take advantage when constructing their HPC applications.

Current Problems and Contributions

For the interactive HPC applications described above, our group's earlier work showed substantial (order of magnitude) differences in performance for Java- vs. non-Java-based communications[12], for the transfer of scientific data of interest to HPC end users. In response to this problem, we have been developing a lightweight, Java-based communication middle-ware, called JECho.

JECho addresses three requirements of Java-based interactive HPC applications, in Grid environments and/or in ubiquitous computing/communication settings:

1. *High level support for anonymous group communication* -- to permit end users to collaborate via logical event channels[19][20] to which subscribers send, and/or from which they receive, substantial amounts of data, rather than forcing them to explicitly build such collaboration structures from lower-level communication constructs like Java sockets or raw object streams;
2. *Scalability in group communication* -- to permit large numbers of end users to collaborate with performance exceeding that of other Java-based communication paradigms, including Javaspaces[14], Jini events[15], and the lower-level mechanisms used by them, such as RMI[16]; and
3. *Heterogeneity of collaborators* -- to enable collaboration across heterogeneous platforms and communication media, thereby supporting the wide variety of scientific/engineering, office-, and home-based platforms across which end users wish to collaborate.

JECho addresses these three requirements by providing a lightweight, performance conscious, distributed implementation of event channels[1]. Besides using a simplified and optimized runtime system, performance enhancement is achieved by using an optimized object transport layer. This layer operates across both standard and embedded JVMs, and uses standard Java serialization as fallback (i.e., objects that implement only the `java.io.Serializable` or

[1] JECho also supports reliable mobility for communication end-points and the ability to inter-operate with the ECho native distributed event system described in [21], thereby permitting users to construct collaborative applications that span both the C/C++ and Java domains. These aspects of JECho, however, are not described further or evaluated in this paper.

`java.io.Externizable` interface will be sent from one J2SE JVM to another using Java's standard serialization). Scalability is addressed by offering both synchronous and asynchronous event delivery modes, and by reducing the total serialization overhead experienced in group communications. Heterogeneity in end user needs and of underlying execution platforms is addressed by JECho's concept of *eager handlers,* which may be used to dynamically customize communications for individual user or for groups of collaborators.

JECho is pure Java, thus operates across both the NT and Unix operating systems. Ports to wireless devices like laptops and palmtops are in progress. Benchmarking results show that JECho meets the requirements of high performance collaborations listed above. First, its synchronous communication mode has latencies that vary from slightly to considerably better than that of Java RMI. Second, its asynchronous delivery mode provides substantially higher communication throughput (up to 1240%) than its synchronous mode. Furthermore, with respect to group size, both delivery modes scale better than current implementations of RMI and the one-way messaging of a Java-based commercial product. Its asynchronous mode also scales well with respect to the lengths of the communication paths being constructed. This latter scalability is critical for the stream- and network-like communication structures constructed in support of many of our collaborative HPC applications[16][4].

JECho's novel concept of '*eager handler*' is the basis for mapping JECho to highly heterogeneous computing platforms. Our ongoing experimentation addresses both ubiquitous and grid applications. The idea of an eager handler is to partition a sink-side (i.e., a client-specific) event handler at runtime to move event handling code from the sink to the appropriate sources. In this fashion, a sink can specialize its event sources, typically resulting in reduced sink-to-source communication bandwidth requirements, although such specialization is subject to the availability of source-side processing resources. By using eager handlers, we were able to improve the effective communication throughput of a scientific-data visualization application by up to 85%. By changing handler partitioning at runtime, we were able to maintain such improvements even when end users radically changed their behaviors. Even radical changes like replacing the handlers employed for communications costs as little as 1.23msec for clients of one of the sample applications in our experiment environment (see section 5).

The remainder of this paper is organized as follows. Section 2 describes sample applications. Section 3 introduces JECho's abstractions, followed by a brief outline of their implementation in Section 4. Results of benchmarking tests appear in Section 5. Section 6 discusses related work, and Section 7 completes the paper with conclusions and future work.

2. Target Applications and Environments

The evaluation of JECho presented in this paper uses applications representative of some of future systems. In such applications, end users collaborate via potentially high-end computations that involve large data sets, and/or rich media objects are created and shared across highly heterogeneous hardware/software platforms.

One such application created and evaluated by our group implements the collaborations of scientists and engineers. In the application, data is not only moved between multiple application components,

but also from these components to user interfaces running on various access engines. The two types of access engines with which we experiment in this paper are (1) those used in labs/offices offering high end graphical interfaces and machines and (2) those in mobile settings using Java-based tools running on laptops or even PDAs. In such a setting, users wish to switch from one access engine to another, as they move from one lab/office to another or from lab/office to shop floors or conference rooms. Furthermore, two-way interactions occur, such as those where engineers continuously interact via simulations or computational tools (including when jointly 'steering' such computations and sharing alternative views of large-scale data sets[8][9]).

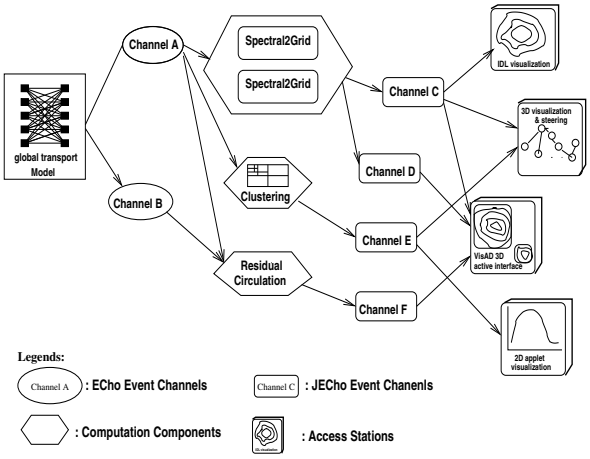

Figure 1. Using event Channels in Multi-user, Multi-view Collaborations.

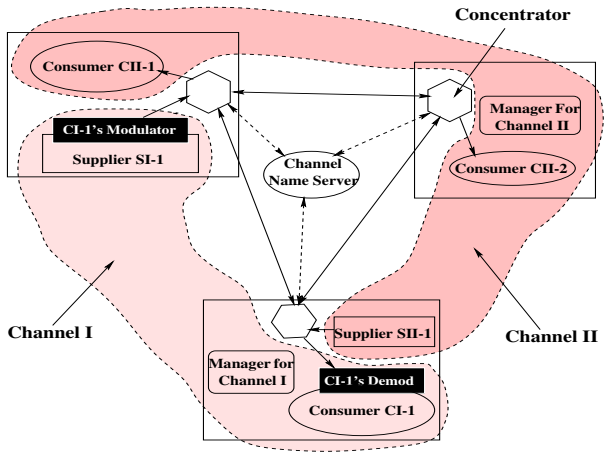

Figure 2. Components of JECho Distributed Event System.(Modulator and Demodulator are explained later)

Figure 1 depicts a simple version of a multi-user and multi-view collaboration via computational components. Three concrete instances of such collaborations have been constructed by our group, including an interactively steered simulation of the earth's atmosphere[4], an instance of the hydrology workbench originally developed at the Univ. of Wisconsin[11], and a design workbench used by mechanical engineers in materials design. The figure also shows different user interface devices and connectivity being employed, ranging from high-end immersive systems to web browsers or palmtops (being used to 'stay in touch' or to loosely cooperate with selected application components). Two important elements of JECho depicted in this figure are

(1) its ability to automatically transfer data from the Fortran or C/C++ domains (in which high end computations tend to operate) to the Java domain and vice versa, and
(2) its ability to operate across heterogeneous underlying hardware/software systems, including NT, Unix and embedded systems.

In measurements reported in more detail in [41], we evaluate experimentally the costs of translating the data typically used in our applications to Java objects (and vice versa). One interesting result is that it is often the costs of object communication in the Java domain that dominate performance, not the costs of translating structured binary data to Java objects (and vice versa).

A second application now being constructed by our group targets ubiquitous computing environments, involving wireless-connected laptops and palmtop devices. This application implements server-side

functionality that provides client-specific flexibility, in excess of which currently offered by typical web portals. The idea is to use eager handlers to permit servers generate and deliver content to clients based on dynamically changing client profiles. One example of such generated content are user-selected instant replays for sports actions being viewed, where both the replays and the concurrently ongoing continuous data deliveries must be adapted to current client connectivity and capabilities.

3. JECho Concepts

Basic Concepts

JECho supports group communication by offering the abstractions of events and event channels. An *event* is an asynchronous occurrence, such as a scientific model generating data output of interest to several visualization engines used by end users, or a control event sent by a wireless-connected sub-notebook throttling data production at some source. Events, then, may be used both to transport data and for control. In either case, an event is a Java object with some well-defined internal structure defined using XML[22] or lower-level specifications. An *event endpoint* is either a producer that raises an event, or a consumer that observes an event. An *event channel* is a logical construct that links some number of endpoints to each other. An event generated by a producer and placed onto a channel will be observed by all of the consumers attached to the channel. An *event handler* resident at a consumer is applied to each event received by the specific consumer.

Since the notion of publish/subscribe communications via events is well known, the remainder of this section focuses on an innovative software abstraction, termed *eager handler*. Its purpose is to deal with the dynamic heterogeneous systems and user behaviors targeted by JECho. Two aspects of JECho not discussed further in this paper are the inter-operability between Java and non-Java event endpoints, and end point mobility. Eager handlers, however, are critical to JECho's ability to deliver suitable performance for heterogeneous, high performance computing and communication environments. The performance implication of eager handlers, as well as JECho's other optimization and customization efforts, will be described in detail in Section 5 below.

Eager Handlers -- Distributing Event Handling Across Producers and Consumers

Consider the multi-user and multi-view depiction of data being generated by a single source. Use the multiple, distributed visualizations of the scientific data generated by a single running simulation as an example[4](see Figure 1). When some Java-based visualization engines, such as VisAD, are to visualize data received from the running model, they usually cannot display the wealth of data continuously being produced, neither does the end user want to inspect all the data at all the times. In response, most visualization applications will, not only transform data for display, but also down-sample or filter it, in order to create useful views. In other words, the data consumer (i.e., the visualization) applies a handler to the incoming data that filters or down-samples it before presenting the data to its graphical processing component. Moreover, such filtering varies over time, as end users view data in different forms, zoom into or out of specific data areas, or simply change their level of attention to their graphical displays. In all of those cases, it is clearly inappropriate to send all possible data for display to the visualization engine, only to discover that most data will be discarded.

To summarize, in order to customize data for each visualization client, it is necessary for a client to dynamically control its data sources in accordance with its current data needs and resource availability[13]. Otherwise, it would receive unneeded or undesirable data. In effect, event receivers must be able to customize event producers.

JECho handles the dynamic, receiver-initiated specialization of data producers with a novel software abstraction: *eager handler*[1]. An eager handler is an event handler that consists of two parts, with one part remaining in the consumer's space and the other part replicated and sent into each event supplier's space. We term the latter *event modulator*, while the part that stays local to the consumer is termed *event demodulator*. Events first move through the modulator, then across the wire, and then through the demodulator. The event modulator is split from the original handler, moved across the wire, and then installed in order to operate inside the producer's address space. Namely, it is 'eager' to touch the producer's events before they are sent across the wire.

The result of using an eager handler is not that all event consumers suddenly receive modulated events. Instead, by partitioning a handler, the specific client's modulator implicitly creates a new event channel 'derived' from the channel used previously, and the client automatically subscribes to this new channel. As a result, eager handler creation initially affects only the specific client that performed handler partitioning, though we also permit additional clients to subscribe to the newly created and now modulated event stream. More specifically, any consumers of a channel that use the same modulator subscribe to the same event channel `derived' from the original one. Whether or not two modulators are the same is determined by the user-defined *equals()* methods of the modulators.

A sample eager handler used in this paper is applied to an event channel that provides to a scientist data from a running atmospheric simulation. Such data is, in accordance with the atmosphere's representation, structured into vertical layers, with each layer further divided into rectangular grids overlaid onto the earth's surface. A scientist viewing this data (by subscribing to this channel) may change her subscription at any time. Examples of such changes include: (1) specifying to the partitioned handler new values for desired grid positions, and (2) changing the partitioned handler to create new ways in which data is clustered, down-sampled, or converted for interactive display. Such flexibility is important since at any one time, the scientist is typically interested only in studying specific atmospheric regions, at some desired level of detail, using certain visual tools and analysis techniques. Runtime handler partitioning helps us implement such tasks by enabling changes both at the data consumer and data provider sides of a communication, thereby reducing the bandwidth needs and the processing power requirements at the recipients.

We have already demonstrated the importance and benefits of client-controlled, dynamic data filtering for wide area systems[13]. Such filtering is even more important in the Java environment where communication costs are high. Therefore, our principal goal in creating the notion of eager handlers is to prevent networks with limited bandwidth and event consumer stations with limited computing capability from being flooded by events. In addition, eager handlers can be also be used for:

- *Consumer-specific traffic control*: Using eager handlers, event consumers can change the scheduling methods and/or priority rules used by producers, thereby enabling clients to control

[1] JECho's eager handler is similar to ECho's[21] notion of 'derived' event channel, with JECho offering more general functionality due to its use of Java facilities like object serialization, dynamic class loading, etc.

event traffic based on application-level semantics; examples include priority delivery for events tagged as 'urgent' and runtime changes in event delivery rates.

- *Quality control on event streams*: An event consumer may use an eager handler to filter out less important events, to perform lossy compression to match event rates to available network bandwidth, or to simply drop some of the events (rather than leaving it up to the supplier to determine which events are important to the consumer). Such consumer-based event stream control is particularly important when producers do not know about consumers' event usage, thereby making it unrealistic to have producers implement appropriate QoS criteria for the event stream.
- *Event transformation and filtering*: Since only consumers know about their current usage of data-carrying events, JECho gives them the ability to customize and transform events before producers send them. One example of the utility of consumer-based event transformation is a consumer providing a handler that transforms a full stock quote issued by a live feed into one only carrying only a tag and a price. Other examples include event clustering, encryption, and compression.

Our current research is exploring some of these broader opportunities realizable by runtime handler partitioning. In this paper, we demonstrate the utility of eager handlers to limit bandwidth consumption as well as the computational costs experienced by receivers.

4. JECho Implementation

A JECho system (see Figure 2) consists of *channel name servers*, *concentrators*, *channel managers*, *channels* and *event endpoints*. In this section, we first describe issues in implementing JECho's base system, then we describe the implementation of JECho's eager handlers.

Base System

The key goals of JECho are system performance and scalability. For the base system's implementation, this means that channels, endpoints, and events must be lightweight entities in terms of the event processing and transport overheads they imply.

Scalability with Respect to Numbers of Channels and Clients:
JECho's implementation uses the *concentrator* model. Each Java virtual machine (JVM) involved in the system has a concentrator that serves as a hub for all incoming/outgoing events. Since the concentrator multiplexes the potentially large number of logical event channels used by the JVM onto a smaller number of socket connections to other JVMs, JECho can easily support thousands of event channels. Furthermore, since each concentrator can rapidly dispatch local events, without involving some remote entity, event transport within a JVM has low latency. Finally, concentrators can reduce total inter-JVM event traffic by eliminating duplicated events sent across JVMs when there are multiple consumers of one channel residing within the same concentrator.

Bookkeeping is distributed, a prerequisite for building a scalable event infrastructure. Specifically, to each event channel is assigned a *channel manager* that maintains such information, thereby distributing such meta-data generation and storage across multiple managers. Sample bookkeeping data includes information about which concentrator is currently involved with the channel, the number and types of end points of the channel currently residing in that concentrator, etc. JECho can be

instantiated with any number of channel managers, where the mapping of channels to managers are maintained by the channel name servers.

A *channel name server* defines a name space for channel names. The name of an event channel is represented by a <*name server address, channel name*> pair. Name server address is the IP address (and TCP port number) of the channel name server, and the channel name is a user-defined string. This naming scheme helps avoid possible naming conflicts in a large-scale system as a system can deploy multiple independent name servers.

Optimizing/Customizing Object Serialization:
To efficiently handle and move the large data events used by collaborative applications in the HPC domain, specific attention must be paid to the marshalling and unmarshalling of such events[28][27]. In Java domain, this implies that we must reduce the overheads of its object serialization mechanism. A second issue to be dealt with for the ubiquitous computing platforms targeted by JECho is that the object serialization protocol is not currently supported on all editions of the Java virtual machine. In order to address these problems, JECho has customized its object transport layer. Specifically, JECho provides a customized object stream that serializes objects that implement the `jecho.JEChoObject` interface. This interface is similar to the `java.io.Externizable` interface, except that it uses `JEChoObjectInputStream` and `JEChoObjectOutputStream`, instead of standard `ObjectInputStream` and `ObjectOutputStream`.

The JECho object stream is a simplified version of Java's standard object stream, in that, amongst others, it does not support dynamic loading of classes from remote sites (the reason for this is that some JVMs do not support runtime class verification). However, an event producer can still send an object of type unknown to the consumer as long as both are running on top of JVMs that support standard serialization. This is because JECho's object stream embeds a standard object stream when both ends of the JECho stream are on non-embedded JVMs. But this standard stream is invoked only when necessary.

JECho also optimizes the object output stream for specific objects commonly used in the applications we address. This includes objects of types like `Integer` (`java.lang.Integer`), `Float` and `Hashtable`, all of which are specially treated to improve serialization and communication performance. This is especially useful when a vector or a hashtable, which is likely to contain such objects, is serialized and deserialized. One of our experiments in section 5 shows that such optimization can save up to `71.6%` of total time.

Another improvement in JECho is in the object output stream. In Java's standard object output stream, there are usually two layers of buffering when the stream is used for network communication: the first layer is the internal buffer in `ObjectOutputStream` for *block data mode*, the other layer is the buffer in `BufferedOutputStream`. The latter layer is necessary because otherwise, every change to the state of *block data mode* (which can be cause by a reset to the stream, which happens for RMI invocation) will cause a write operation to the call to the underlying network layer. JECho's object output stream combines these two layers into one, thereby avoiding the additional copying. The effect of this optimization is also shown in Section 5.

JECho's object transport layer also does *group serialization* for events to be sent to multiple destinations. Instead of using multiple object streams (one between the sender and each of the receivers), which will result in serializing the event for multiple times, JECho serializes the event once and sends the resulting byte array directly through sockets. Benefit of this is obvious when sending a complex object to multiple destinations.

Flexible Event Delivery:
Collaborative applications, as well as multimedia or sensor processing codes running in wireless domains, are often comprised of sequences of code modules operating on streaming data. These pipeline/graph-structured applications expect that different execution stages will run concurrently and across multiple machines. In response, JECho offers not only a synchronous model for event handling and delivery, but also permits applications to publish and consume events asynchronously. Asynchronous delivery means that a producer returns from an 'event submit' call immediately after the event has been placed into an outgoing event queue. It requires producers to employ other, application-level means for checking successful event distribution and reception when necessary (we have created application-level handlers that implement several useful 'end-to-end' delivery guarantees). Synchronous event delivery, however, offers strong semantics for event delivery. It returns successfully from an event submission only when all consumers of that event channel have received and processed the event (in other words, the invocation to the handler function at the consumer side has returned and an acknowledgment has been received by the supplier side). For both synchronous and asynchronous events, event delivery is partially ordered in that all consumers of a channel observe events in the same order in which any one producer generates them.

Asynchronous event delivery is important not only because its functionality matches the needs of JECho's target applications, but also because asynchronous event handling offers event throughput rates that exceed those of synchronous mechanisms (e.g., RMI or JECho's synchronous events). Asynchronous delivery can overlap the processing and transport of 'current' with 'previous' events, and it can also batch the delivery of events. Event batching means that multiple events sent to the same concentrator result in a single, not multiple Java socket operations (and multiple crossings from the Java domain into the native domain), generating significantly higher event throughput rate for smaller events (see section 5).

Implementation of Eager Handlers

One of the most interesting features of JECho is its notion of `eager handlers'. The idea is to permit an event consumer to specialize the content and the manner of handling and delivery of events by producers. This is achieved by `splitting' the consumer's event handler into two components, a `modulator' resident in the event supplier and a `demodulator' in the consumer. Furthermore, to each client, the multiple producers in which modulators exist are anonymous. Consequently, JECho must take care of modulator replication, of their placement into potentially multiple event producers, and of their safe execution in those contexts. Therefore, it is important for the system to (1) provide secure environments with necessary resources for the execution of modulators, (2) ensure state coherence among replicated modulators, and (3) define an interface for modulators to define their actions upon system state changes. JECho accomplishes (1)-(3) by providing the Modulator Operating Environment (MOE):

- MOE's *resource control interface* exports and controls `capabilities' based on which event users can access system- and application-level resources;

- MOE's *shared object interface* provides consistency control for replicated modulators that share state; and
- MOE's *intercept interface* defines a set of functions that are invoked at different state changing moments. For example, an *Enqueue* function is invoked when a supplier generates an event, a *Dequeue* function is invoked when the transport layer is ready to send an event across the network, and a *Period* function is invoked when a timer expires.

Figure 3 shows the architecture of MOE. . We next demonstrate the use of these interfaces.

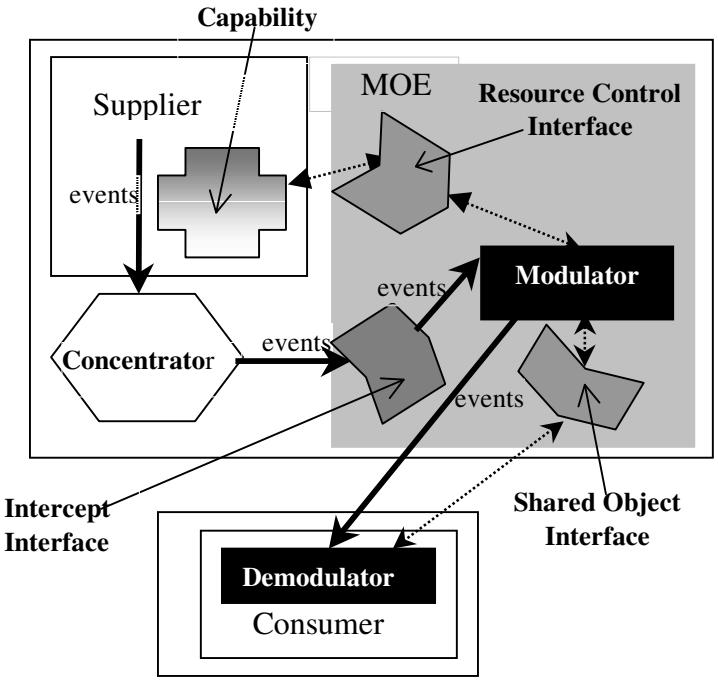

Figure 3. MOE Architecture

Resource Control Interface.
MOE's resource control interface provides a secure environment containing the resources necessary for modulator execution. A modulator can use two classes of resources: system resources and application resources. JECho depends on Java's built-in security model for access control on system resources[1].

For application resources, JECho's modulator operating environment provides a resource control interface that allows suppliers to export resource descriptors that can be granted to consumers. Specifically, a modulator can specify a list of services (implemented as Java interfaces) that it expects from the supplier's MOE in order to be able to execute correctly. In addition, when subscribing to a channel, a supplier can provide a delegate to the MOE. This delegate provides handles to services upon requests from the MOE.

[1] We are also looking at incorporating runtime resource management tools, such as Cornell's JRes[29], into MOE to more effectively control system resources.

When installing a modulator, for each service required by the modulator, if the MOE cannot provide it, then it will request the service from the supplier's delegate. If the delegate cannot provide it either, then an exception will be raised and the process of eager handler installation will fail.

Shared Object. Interface

Since a distributed event channel can have more than one supplier, a modulator of an eager handler must be replicated in all suppliers. Such replication will not cause any problem if the modulator is stateless. Otherwise, there must be a consistency control mechanism to ensure state coherency among replicated modulators. Furthermore, as a modulator comes from (is instantiated in) the consumer's space, it may reference objects defined at the consumer. In such cases, it is important to ensure that the modulator can correctly reference such objects after being installed in the supplier's space.

JECho's MOE provides a *shared-object interface* as the consistency control mechanism for both cases of state sharing. A modulator can reference a number of shared objects. Each shared object has a master copy, and from this master copy an application can create an arbitrary number of secondary copies. Both the master copy and all of the secondary copies can read and write the shared state. The master copy always has the newest version of the state; all updates performed at the secondary copies are sent to the master copy immediately. The master copy can choose from *prompt* or *lazy* update policies to decide whether updates should be propagated to secondary copies immediately or not. Secondary copies can also actively pull the newest version of the shared from the master copy.

One use of the shared object interface is demonstrated for the sample application in a code fragment appearing in Appendix 0. The purpose of that object is for the modulators and demodulators of eager handlers to share parameters. This presents to end users the appearance of a partitioned handler parameterized in terms of longitudes, latitudes, and atmospheric layers.

JECho's shared object interface is implemented in pure Java and does not require any compiler help. The feature that distinguishes it from other distributed shared data systems is that it enables a piece of code to continue working properly after the code has been migrated (and replicated) at runtime.

Intercept Interface.

A modulator can specify its response to relevant state changes occurring at the supplier by defining *intercept functions*:

- *Enqueue function*: The enqueue function of a remote handler is invoked at the time a producer pushes an event onto the channel. This function takes the event as a parameter. The function can perform any operation on the event, including discarding it, transforming it, or storing it somewhere.
- *Dequeue function*: Dequeue function is invoked at the time the transport subsystem delivers an event from a remote handler to its associated event consumer. This function returns the event to be delivered.
- *Period function*: The period function is invoked whenever the elapsed time since this function was last called exceeds some specified period. This function is useful in producers to 'push' data and at consumers to 'pull' data at well-defined rates.

Changing Modulators and/or Demodulators.

Given the MOE support of JECho, modulators can collaborate with demodulators to implement application-specific group communication protocols, and such protocols can be efficiently changed

at runtime. Changes are enacted by having an event consumer providing a new modulator-demodulator pair and then reset its event handler, thereby dynamically adapting the communication protocol it uses with its event supplier. An example of such a change can also be found in [40].

5. Evaluation

All measurements presented in this section are performed on a cluster of Sun Ultra-30 (248 MHz) workstations, each with 128MB memory, running the Solaris 7 OS and connected by 100Mbps Fast Ethernet. The roundtrip time for native sockets is about 260us. The JVM is from J2SE 1.3.0.

Recall the basic requirements of Java-based, interactive HPC applications to be supported by JECho: (1) anonymous group communication for data of substantial size, (2) scalability for groups in terms of potentially large numbers of publishers and subscribers, and (3) runtime adaptation and specialization to support highly heterogeneous distributed systems and applications. To evaluate JECho with respect to these requirements, this section presents measurements that compare JECho's performance to RMI, which is used by some the current implementations of Java-based distributed event systems including JavaSpaces and versions of Jini event systems. We also compare with Voyager's (which is an influential commercial product from ObjectSpace) messaging mechanism, albeit Voyager provides a lot more functionality other than messaging. Results show that JECho's performance exceeds that of RMI and Voyager, sometimes by substantial margins. Our results in more complicated experiment setups show that JECho, and thus Java, can potentially support large-scale applications.

Object Types	ObjectStream (JDK1.3, reset)	ObjectStream (JDk1.3, NO reset)	RMI (JDK1.3)	JECho ObjectStream	JECho Sync	JECho* Async
null	460	454	929	455	791	59
int100	968	841	1625	714	1073	177
byte400	887	766	1420	638	1011	143
Vector of Integers	2603	2553	3186	723	1097	225
Composite Object	2851	1753	3219	996	1334	318

Table 1. Round-trip Latency for Different Objects (in usec). Return objects are always 'null' objects. The difference between the 1st and 2nd columns is that the first column does a reset to the stream before sending each object. RMI also does such resets. The round-trip time for native sockets is about 260usec.

Simple Case Latency and Throughput

This experiment measures the roundtrip latency and throughput (JECho Async only) for single source, single-sink setups. Separate measurements send one of the five types of objects from source to the sink: null, an array of 100 integers, an array of 400 bytes, a `Vector` of 20 `Integers` and a `composite object`, which has a string, two arrays of primitives and a hashtable with two

* JECho Async numbers are for 'average time used per event', rather than for 'round-trip latency'.

entries. All the objects implement java.io.Externizable for better standard serialization whenever necessary. All setups except JECho Async send null objects from sink to source as acknowledgements. All timings are initiated some time after each test is started, in order to allow for dynamic optimizations to take effect.

Table 1 shows that, as one would expect, JECho Async offers much higher event throughput rates than both JECho Sync and RMI do, as it uses event batching and one-way messaging. Also, in addition to some of the simplifications made by JECho's implementation (e.g., no full class and activation support), JECho Sync has shorter latencies than RMI:

- *Less base runtime overhead*: For the 'null object' case, while the underlying streams perform at the same level, RMI has 17% more overhead than JECho Sync, partly because JECho does optimization for special cases. For instance, if a sink has only one source and message is sent synchronously, then the sink will go into 'express mode', using a single thread to read the incoming event, process the event and send back an acknowledgement.
- *Eliminating additional level of buffering*: As we described earlier, JECho uses its own output stream to combine ObjectOutputStream's internal buffering with external buffering. This is partly reflected in the 'byte400' case: standard object stream (without reset) has 20% overhead over JECho stream.

- *Special serialization for commonly used objects*: the effect of special serialization for objects like java.lang.Integer and java.util.Hashtable is reflected in the 'Vector of Integers' case, where standard stream (without reset) costs 255% more than JECho's serialization.
- *Persistent stream states:* While RMI needs to reset steam state (or create a new stream) for each invocation, JECho does not do so unless explicitly requested. In the 'Composite Object' case, this 'reset' causes about 63% of the overhead for standard stream, which is part of the reason that the current implementation of RMI is not optimal for stream-based applications.

These optimizations, combined with the fact that JECho currently does not support some of RMI's more advanced features, make JECho Sync 58.6% faster than RMI for 'composite objects'.

Multi-sink Throughput and Latency

Figure 4 shows the measurement numbers for JECho Sync, JECho Async, RMI and Voyager multicast one-way messaging under varying number of sinks.

Since current implementations of RMI do not yet support group communication, the RMI numbers in the figure are not actual measurements. Rather, they are deducted from the following formula and are used only as reference numbers:

$$T_{RMI}(n, o) = T_{RMI}(1, o) + (n - 1) * T_{OS}(1, \text{byte[sizeof (o)]}),$$

Where $T_{RMI}(n, o)$ is the latency for RMI to send object o to n sinks, $T_{OS}(n, o)$ is the roundtrip latency of the standard object stream. Note that it always takes a byte array with a length of the size of the object, rather than the object itself. In essence, this hypothetical 'multicast-RMI' (hereafter termed

RM-RMI) only serializes the object once, for the first sink, and the result byte array will be reused to be sent to remaining sinks, exactly as with the current implementation of JECho. We use these hypothetical RM-RMI numbers in order to provide a fairer comparison with JECho than that produced by our actual measurements (where RMI does repeated serialization). RM-RMI performance, therefore, is substantially better than that of our actual RMI measurements.

The reason JECho Sync still scales better than RM-RMI is that JECho Sync parallelizes its send and reply-receive tasks with respect to different subscribers, by overlapping these tasks in a way similar to that used by vector processors to achieve parallelism. As a result, an event might still be in progress of being sent to some subscriber S2 while a reply to this event is already being received from some other subscriber S1. Figure 4 shows that for each additional sink, the increased overhead of JECho Sync is about half of that of RM-RMI.

It is not surprising that JECho Async scales much better than both JECho Sync and RM-RMI. Furthermore, compared to Voyager's multicast one-way messaging, JECho Async provides much higher (50+ times better for 'null', and 18+ times better for 'composite' objects) event throughput rates. JECho Async also experienced much less overhead for each additional sink. For instance, for the 'null' objects, this overhead is about 10us for JECho Async, while it is in the range of from 200us to 700us for Voyager multicast. We suspect that this performance disparity is caused by: (1) Voyager's one-way messaging is probably built on top of synchronous unicast remote method invocation, and (2) Voyager is subject to overheads for features such as fault-tolerance support, which JECho lacks.

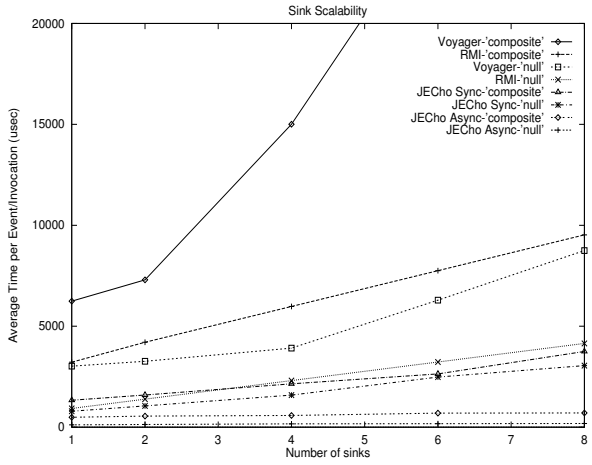

Figure 4. Average Time (in usec) for Sending an Event/Invocation for Different Number of Sinks

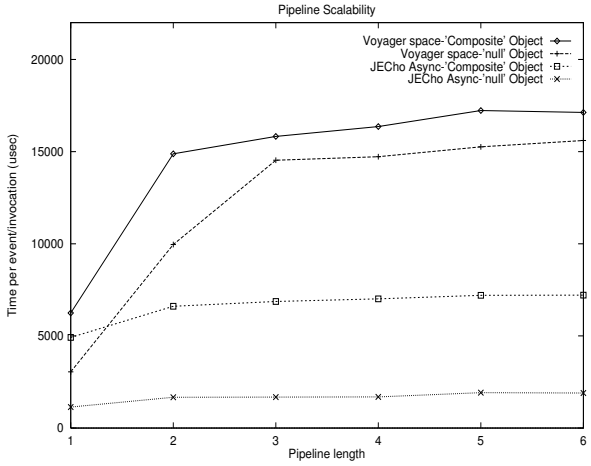

Figure 5. Average Time (in usec) for an Event/Invocation to Travel Through a Pipeline of Components, with Changing Pipeline Length(JECho numbers are timed by 10 for the ease of comparison)

Pipeline Throughput

In large-scale distributed collaborative applications and in the cluster server application described in Section 2, the communication pattern among distributed components of the application can be complex, resulting in communication paths within applications where events are sent across multiple

channels. For instance, component A might send an event to component B. In handling this event, B sends another event to component C. As a result, an event from A to B will result in the creation of a communication pipeline of length 2.

Experimental results depicted in Figure 5 clearly show that asynchronous event delivery and handling are essential for achieving scalability along the 'length' dimension of communication pipelines. Specifically, for JECho Async, the throughput rate is much less affected by any increment in pipeline length. In fact, the throughput rate is largely determined by the speed of the relayer, which is slower than both the sender and the receiver, as it has to receive as well as send events. This is shown in the figure, that JECho Async's curves are relatively flat after pipeline length of 2.

Multi-channel Throughput

Larger applications may use a large number of logical channels, reflecting the complex control and data transmission structure of these applications. Figure 6 depicts JECho Async's throughput rate under changing number of logical channels. In this experiment, the channel used for sending an event is chosen in a round-robin fashion. Results show that throughput does not vary significantly for up to 4096 channels.

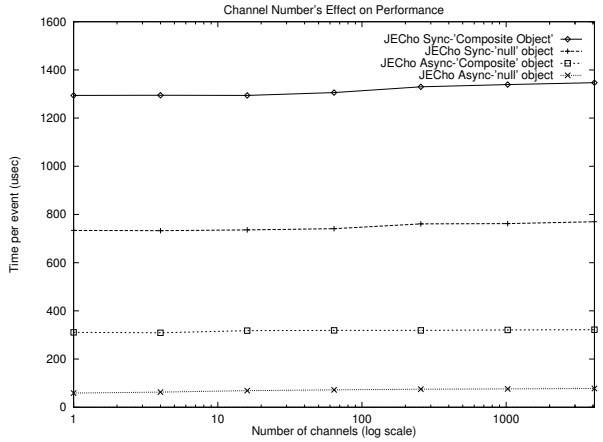

Figure 6. Average Time (in usec) for Sending an Event Using Different Numbers of Channels.

Costs/Benefits of Eager Handlers

The eager handler mechanism enables a single consumer of an event channel to specialize the events it receives, without affecting the other channel consumers. This is particularly important in an anonymous group communication environment, as a subscriber to a channel cannot have a priori knowledge of other channel subscribers, let alone about the events these other subscribers will place onto the channel.

As an example, consider the collaborative scientific application presented in Section 2. Here, it is clear that different channel subscribers may differ both with respect to their data needs and display abilities. For instance, a web-based display used by a student collaborator may view only small subsets of the data viewed by a teacher manipulating the actual application and its data outputs. The

eager handler mechanism makes it easy to implement scenarios like these, where a student simply uses a different modulator/demodulator pair than the teacher, while subscribing to what appears to be the 'same' event channel. We next first describe the costs of eager handler installation, followed by presentation of one example of its utility.

Costs of installing an eager handler.
Installing an eager handler and/or dynamically modifying it can be done in two ways:

- *Updating an existing modulator using the shared object interface:* shared objects used in a modulator can be changed at runtime. Such shared objects can be looked at as parameters of the modulator and by changing them, a consumer can change the parameters of its modulator. Since a shared object is implemented using Java sockets, the costs of changing the value of a parameter is the cost of sending the parameter object to all suppliers of the channel via object serialization. In the code fragment appearing in Appendix 1, an update to the current_view shared object has a latency of about 0.5ms (in our test environment) when there exists one supplier. The benefits of such parameterization are obvious when the view window shrinks, as it may potentially filter out large amount of events. The operation must be performed each time the view window shifts or is reduced/enlarged.
- *Changing modulator/demodulator pairs at runtime:* JECho provides an API using which a consumer may replace its modulator/demodulator pair at runtime. There are two components to the cost of doing so: one is the cost of shipping the modulator object itself from the consumer's space to the supplier's space and installing it, the other is the cost of loading the bytecode that defines that specific modulator class. The cost of class loading depends on the performance of class loader and hence is out of JECho's control. It will not be discussed further. However, to ship a modulator (again using object serialization) and to install it at a supplier, results in costs that are just slightly higher than the cost of synchronously sending an event of the same size. For example, for a modulator with state (data fields) of size similar to that of a 100-integer array, the total cost of handler shipping and installation is approximately 1.23ms under our test environment (with the supplier's classloader loading modulator code from its local file system).

Benefits of Dynamically Changing Eager Handlers.
While it is hard to quantify the benefits from features like QoS control provided by an eager handler, it is obvious that filtering and down-sampling can reduce network traffic and system load. In our sample application, depending on the dimensions of users' views and their displays' resolutions, the use of eager handlers can reduce network traffic by up to 85% via event filtering, with consequent additional savings in the processing requirements for events received by clients. Even higher savings are experienced when using event differencing.

Summary of Results

The experimental results in this section show that, JECho's synchronous event delivery is faster than RMI in single-source, single-sink cases because of JECho's optimizations and simplifications in both its object stream layer and its runtime system. JECho Sync also scales better than the reference numbers computed for an appropriate implementation of multicast-RMI.

JECho's asynchronous delivery offers much higher throughput rates than JECho Sync and than Voyager 's one-way messaging. It also scales better, both in terms of increases in the number of sinks

and increases in the lengths of communication paths. JECho's good scalability, combined with the fact that JECho channels are lightweight, and that we distribute channel name servers and channel manager, makes us believe that it is viable to build high performance, large-scale event delivery systems with JECho.

The benefits of advanced JECho features like eager handlers are apparent for a simple scenario of use demonstrated in this section. In general, such benefits arise from the reduction of network bandwidth and network-relevant processing at suppliers and from reducing the number of irrelevant events (and their processing) at receivers. Experiments with the non-Java version of JECho (the ECho event system[21]) described in [13] demonstrate the utility of eager handler. Additional experiments now in progress will show the utility of eager handler for collaborative applications in general and for the runtime quality management actions described in Section 1.

6. Related Work

There has been much work to improve Java object serialization and RMI[36][35][38]. In particular, UKA-RMI[36] also does buffering optimization to speedup serialization. However, UKA-RMI's output side buffer optimization exposes the object output stream's internal buffer but it does not eliminate the extra layer of buffering; both its output and input side buffer optimizations are orthogonal to our optimizations. There has also been a considerable work on high performance messaging in Java[30][31][33][34]. Some of these systems are native-code libraries with Java interfaces[30][33], while pure Java systems had performance limits, especially for roundtrip latencies[32][34].

Jini[15]'s distributed event specification does not rely on RMI, but most current implementations of this specification are based on unicast RMI, which, as we demonstrated, has performance limitations in distributed systems. Some commercial Java notification and messaging systems, such as JavaSpaces[14] and Voyager[26] are also based on unicast remote method invocations. While these systems provide higher-level features such as transaction and persistency support, they usually do not implement direct connections between sources and links, hence they are less likely to satisfy the performance requirements of throughput- and latency-conscious applications.

Gryphon[39] is a content based publish/subscribe system that implements the JMS distributed messaging system specification[23]. Its parallel matching algorithm enables the system to expand to very large scale in terms of the number of clients it services. However, Its matching criteria are currently limited to database query like expressions, while JECho's eager handler permits virtually arbitrary codes.

The use of code migration for performance improvement is not novel. In particular, some database systems[24] support stored procedures to allow database clients to define subroutines to be stored in the server's address space and invoked by clients. The notion of eager handler is more powerful than stored procedures, in that it permits clients to place `active' functionality into suppliers, with modulators run by their own execution threads. Furthermore, JECho permits handlers to be comprised of arbitrarily complex Java objects, and its MOE (modulator operating environment) provides a general environment in which modulators and demodulators can be dynamically installed, deleted, and changed, for multiple suppliers and consumers.

7. Conclusions and Directions of Future Work

This paper presents JECho, a high performance Java-based communication middle-ware supporting both synchronous and asynchronous group communication. It also presents eager handlers, a mechanism that enables the partitioning of event handling across event suppliers and consumers, thereby allowing applications to dynamically install and configure client-customized protocols for event processing and distribution.

Benchmarking results show that JECho provides shorter latencies and higher throughput than other pure Java-based communication paradigms, including RMI, the transport facility used in most current implementations of Jini's distributed event system. Our results also show that JECho channels are lightweight entities, thereby making it easy to create hundreds of event channels that link event producers and consumers. Furthermore, JECho offers scalability in terms of numbers of data receivers/senders and/or lengths in communication pipelines. Finally, our efficient implementation of eager handlers makes it a useful basis for creating the large-scale, heterogeneous communication infrastructures required for collaborative HPC and cluster applications.

Our future work entails (1) implementing a more secure modulator operating environment, (2) designing an efficient consistency control protocol specialized for high performance event communication systems, (3) automating the process of eager handler generation with the help of runtime program analysis, and (4) supporting standards such as JMS.

8. References

[1] Alliance Chemical Engineering Applications Technologies, http://www.ncsa.uiuc.edu/alliance/partners/ApplicationTechnologies/ChemicalEngineering.html.

[2] NCSA Alliance, Access Grid, http://www-fp.mcs.anl.gov/fl/accessgrid

[3] Space Science and Engineering Center University of Wisconsin – Madison, VisAD, http://www.ssec.wisc.edu/~billh/visad.html.

[4] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin and Jeffrey Vetter, "From Interactive Applications to Distributed Laboratories", IEEE Concurrency, Vol. 6, No. 2, 1998.

[5] James Gosling, Bill Joy and Guy L. Steele, "The Java Language Specification", Addison-Wesley, 1996.

[6] Sun Microsystems, "Java on Solaris 2.6: A White Paper", http://wwwwseast2.usec.sun.com/solaris/java/wp-java.

[7] Sandeep K. Singhal, Binh Q. Nguyen, Richard Redpath, Michael Fraenkel and Jimmy Nguyen, "Building High-Performance Applications and Servers in Java", ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications, Oct. 1997, Atlanta, Georgia.

[8] J.S. Vetter and K. Schwan, "High performance computational steering of physical simulations", Proceedings of IPPS 97, 1997.

[9] Greg Eisenhauer and Karsten Schwan, "An Object-Based Infrastructure for Program Monitoring and Steering", Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98), Aug. 1998.

[10] William Ribarsky, Yves Jean, Thomas Kindler, Weiming Gu, Gregory Eisenhauer, Karsten Schwan and Fred Alyea, "An Integrated Approach for Steering, Visualization, and Analysis of Atmospheric Simulations", Proceedings IEEE Visualization '95, 1995.

[11] National Center for Supercomputing Applications and University of Illinois at Urbana-Champaign, Hydrology Workbench, http://scrap.ssec.wisc.edu/~rob/sc98.

[12] Dong Zhou and Karsten Schwan, "Adaptation and Specialization for High Performance Mobile Agents", Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, 1999

[13] Carsten Isert and Karsten Schwan, "ACDS: Adapting Computational Data Streams for High Performance, Proceedings of IPDPS '00, 2000.

[14] Sun Microsystems, "JavaSpaces Specification", http://www.sun.com/jini/specs/js.pdf.

[15] Sun Microsystems, "Jini Distributed Event Specification", http://www.sun.com/jini/specs/index.html.
[16] Sun Microsystems", "Remote Method Invocation Specification", http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.
[17] Sun Microsystems, "Subcontract: A Flexible Base for Distributed Programming", http://www.sun.com/research/techrep/1993/abstract-13.html, 1993
[18] Persistence of Vision, Povray, http://www.povray.org.
[19] Douglas C. Schmidt and Steve Vinoski, "OMG Event Object Service", SIGS, Vol. 9, No. 2, Feb. 1997.
[20] Douglas C. Schmidt and Steve Vinoski, "Object Interconnections: Overcoming Drawbacks with the OMG Events Service", SIGS, Vol. 9, No. 6, June 1997.
[21] Greg Eisenhauer, "The ECho Event Delivery System", Technical Report, GIT-CC-99-08, College of Computing, Georgia Institute of Technology, 1999.
[22] Tim Bray, Jean Paoli and C.M. Sperberg-McQueen, "Extensible Markup Language (XML): Part I. Syntax", http://www.w3.org/XML/.
[23] Sun Microsystems, "JMS Specification", http://www.sun.com/forte/jmq/documentation/jms-101-spec.pdf.
[24] Oracle, Oracle8i, http://www.oracle.com/database/oracle8i.
[25] National Center for Supercomputing Applications} and University of Illinois at Urbana-Champaign, Habanero, http://havefun.ncsa.uiuc.edu/habanero/.
[26] ObjectSpace Inc., Voyager, http://www.objectspace.com/products/voyager/.
[27] Fabian Bustamante, Greg Eisenhauer, karsten Schwan and Patrick Widener, "Efficient Wire Formats for High Performance Computing", Proceedings of SC2000, 2000.
[28] M. Schroeder and M. Burrows, "Performance of Firefly RPC", In Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5, page 83-90. ACM, SIGOPS, Dec. 1989
[29] Grzegorz Czajkowski, "JRes: A Resource Accounting Interface for Java", In Proceedings of the 1998 ACM OOPSLA Conference, Vancouver, BC, October 1998.
[30] V. Getov, S. Flynn-Hummel, and S. Mintchev. "High-Performance parallel programming in Java: Exploiting native libraries". ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, 1998.
[31] A.J. Ferrari. "JPVM: Network parallel computing in Java". In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998, Concurrency: Practice and Experience, 1998.
[32] Narender Yalamanchilli and William Cohen, "Communication Performance of Java based Parallel Virtual machines". In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, 1998.
[33] P. Martin, L.M. Silva and J.G. Silva, "A Java Interface to MPI", Proceeding of the 5th European PVM/MPI Users' Group Meeting, Liverpool UK, September 1998.
[34] K. Dincer, E. Billur, and K. Ozbas, "jmpi: A Pure Java Implementation of MPI", in Proceedings of ISCIS XIII '98 (International Symposium on Computer and Information Systems), Antalya, TURKEY, Oct. 26-28, 1998.
[35] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol and Mustaque Ahamad, "Efficient Implementation of Java Remote Method Invocation (RMI)", Proceedings of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 1998.
[36] Christian Nester, Michael Philippsen and Bernhard Haumacher, "A more efficient RMI for Java", Proceedings of the ACM 1999 conference on Java Grande, June 1999.
[37] M. Hicks, S. Jagannathan, R. Kelsey, J. T. Moore and C. Ungureanu, "Transparent communication for distributed objects in Java", Proceedings of the ACM 1999 conference on Java Grande, June 1999.
[38] Jason Maassen, Thilo Kielmann, and Henri E. Bal, "Efficient Replicated Method Invocation in Java", Proceedings of the ACM Java Grande 2000 Conference, June 2000.
[39] Gryphon, IBM Research, "http://www.research.ibm.com/gryphon/Gryphon_ASSR/gryphon_assr.html".
[40] Dong Zhou and Karsten Schwan, "JECho—A Java-based Distributed System", Tech Report, to be available at http://www.cc.gatech.edu/tech_reports/.
[41] Y. Chen, K. Schwan and D. Zhou, "An Object-based Infrastructure for Supporting High Performance Distributed Interactive Applications", http://www.cc.gatech.edu/people/home/yuanchen/#research, to be submitted.

9. Appendix A

Handler Partitioning Sample Code

```
// This is the class that defines the shared object
public class BBox extends SharedObject {
    int start_layer, end_layer;
    int start_lat, end_lat;
    int start_long, end_long;

    ...
}

// This is the class that defines the event consumer
public class GridViewer implements PushConsumer {
    BBox current_view;
    FilterModulator mod;

    public GridViewer () {
        current_view = new BBox ();

        // Create a modulator, passing current view as para
        mod = new FilterModulator (mod);

        // Create a push consumer handle with this object as the consumer and with no capability requirement,
        // no restriction on event types Use newly created mod as modulator and use no demodulator
        PushConsumerHandle pch = new PushConsumerHandle (this, null, null, mod, null);

        // connected to the channel named ‘‘MyChannel”
        pch.connectTo (new EventChannel ("MyChannel", null));

        ...
    }

    // This is the handler for events being received
    public void push (Object event) {
        ...
    }

    // This is the method that responds to GUI actions
    public void action (...) {
        if (GUI view changed) {
            // Locally modifies the current view
            current_view.start_layer = new_value1;
            current_view.end_layer = new_value2;
            ...
            // publish modifications so that the modulator can see the change too
            current_view.publish ();
        } else {
            ...
       }
    }
}

// This is the class that defines Modulator
public class FilterModulator extends FIFOModulator {
    BBox consumer_view;
```

```
    public FilterModulator (BBox view) {
        super ();
        consumer_view =      view;
    }

    // Overides the ‘‘enqueue” intercept function
    public void enqueue (DECEvent e) {
        GridData gd = (GridData)e.getContent ();

        // discard the event if layer is not inside consumer’s view
        int layer = gd.getLayer ();
        if (layer < consumer_view.start_layer ||
            layer > consumer_view.end_layer) return;

        // discard the event if latitude is not inside consumer’s view
        int lat = gd.getLatitude ();
        if (lat < consumer_view.start_lat ||
            lat > consumer_view.end_lat) return;

        // discard the event if longitute is not inside consumer’s view
        int long = gd.getLongitute ();
        if (long < consumer_view.start_long ||
            long > consumer_view.end_long) return;

        // Inside consumer view, so enqueue it
        super.enqueue (e);
    }
}
```

This code fragment demonstrates the programming interface for implementing eager handler in the previously described example to achieve event filtering. As we can see from the code, one advantage of MOE’s shared object interface is its ease of use. A shared object need only extend the SharedObject interface and call the publish method, which is defined in the SharedObject class, whenever it wants to propagate modifications. JECho’s runtime system takes care of making copies of the object on different JVMs when necessary and propagates changes to all of the copies of the shared object. For small objects, such latency is less than 0.5ms in environments where the minimum RMI ping time is over 1.5ms.

10. Appendix B

Sample Code for Dynamically Changing Modulator/Demodulator

```
// This is the class that defines the event consumer
public class GridViewer implements PushConsumer {

    ...

    // This is the method that responds to GUI actions
    public void action (...) {
        ...

        // If we are changing from filter mode to diff mode, create a DIFFModulator and
```

```
        // synchronously reset the modulator/demodulator(null) pair.
        if (mode changed to DIFF mode) {
            pch.reset (new DIFFModulator (DIFF_THRESHOLD), null, true);
        }
    }
}
```

The above code segment shows the way to dynamically change modulator/demodulator pair in JECho. Here, the visualization application supports two different modes of operation. One mode is used when data is streamed and displayed continuously. In the other mode, data is sent and displays are updated only when significant changes occur in selected data fields, thereby having the display act as an 'alarm' for such changes. The latter mode employs a modulator that does differencing rather than filtering.