JECho - Interactive High Performance Computing with Java Event Channels

Dong Zhou, Karsten Schwan, Greg Eisenhauer and Yuan Chen
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
{zhou, schwan, 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 concurrent sets of components to provide interactive service to collaborating end users via event channels. JECho's* eager handler *concept allows individual event subscribers to dynamically tailor event flows to adapt to runtime changes in component behaviors and needs, and to changes in platform resources. Benchmark results suggest that JECho may be used for building large-scale, high-performance event delivery systems, which can efficiently adapt to changes in user needs or the environment using eager handlers.*

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[5][6], 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[3][7]. 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[8] created by NCSA researchers uses a Java-based visualization tool, termed VisAD[2], 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.

Java is attractive to such applications due to its ability to inter-operate across machines with different architectures and abilities, ranging from workstations to handheld devices. In addition, users can take advantage of the many available Java-based collaboration and visualization tools[2][14][19]. Unfortunately, a disadvantage of using Java in interactive HPC applications is substantially different performance for Java- vs. non-Java-based communications, as demonstrated in our previous work[9]. In response to this problem, we have been developing a lightweight, efficient, adaptable Java-based communication middleware, called JECho.

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

- *High level support for anonymous group communication* -- to permit end users to collaborate via logical event channels[15][16] to which subscribers send, and/or from which they receive data, rather than forcing them to explicitly build such collaboration structures from lower-level constructs like Java sockets or raw object streams;
- *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[11], Jini events[12], and the lower-level mechanisms used by them, such as RMI[13]; and
- *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.

2. Target Applications and Environments

The evaluation of JECho uses applications in which end users collaborate via potentially high-end computations, involving large data sets, and/or rich media objects, created

and shared across highly heterogeneous hardware/software platforms. One class of applications created and evaluated by our group implements collaborations of scientists and engineers, where 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[5][6]. Three concrete instances of such collaborations have been constructed by our group, including an interactively steered simulation of the earth's atmosphere[3], an instance of the hydrology workbench originally developed at the Univ. of Wisconsin[8], and a design workbench used by mechanical engineers in materials design.

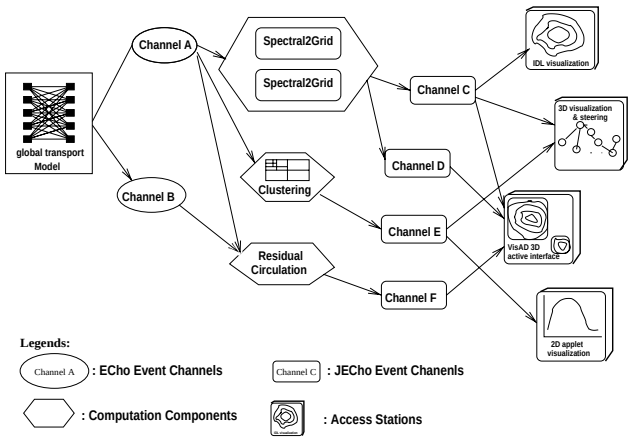

FIGURE 1. Using event Channels in Multi-user, Multi-view Collaborations.

Figure 1 depicts a simple version of a multi-user and multi-view collaboration via computational components. The figure also shows different user interface devices and their respectively different connectivities, ranging from high-end immersive systems accessed via gigabit links to web browsers on wireless-connected palmtops being used to 'stay in touch' or to loosely cooperate with selected application components.

A second class of applications we are now developing targets ubiquitous computing environments, involving wireless-connected laptops and palmtop devices, and it implements server-side functionality that provides client-specific flexibility in excess of what is currently offered by typical web portals. The idea is to have 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 ongoing sports actions, where both the replays and live feed data delivery must be adapted to current client connectivity and needs.

3. JECho Concepts

3.1 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. 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 that 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, for dealing with the dynamic heterogeneous systems and user behaviors targeted by JECho.

3.2 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, exemplified by the distributed visualizations of data generated by scientific simulation[3](see Figure 1). When using Java-based visualization engines, such as VisAD, to visualize data, it is typically impossible to continuously display the wealth of data being produced, nor does the end user want to inspect all such data all of the time. In order to create useful views, visualizations must not only transform data for display, but they must also down-sample or filter it. Specifically, 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 levels of attention to graphical output. Clearly, it is inappropriate to send all data for display to a visualization engine, only to discard much of it. It is preferable for each visualization client to dynami-

cally control which data they transform and display, by controlling what is being sent to them by data sources[10]. Thus, 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*s. 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, conceptually, an eager handler's creation affects only the specific client that performed handler partitioning. For efficiency, our implementation, however, permits all consumers of a channel that use the same modulator to share a single copy of this modulator. 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 new values for desired grid positions, and (2) changing the 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 provider sides of a communication, thereby reducing 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[10]. 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 capabilities from being flooded by events. In addition to transforming and filtering events at a source, 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 event traffic based on application-level semantics.
- *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).

This paper demonstrates the utility of eager handlers to limit bandwidth consumption and to reduce the computational costs experienced by receivers.

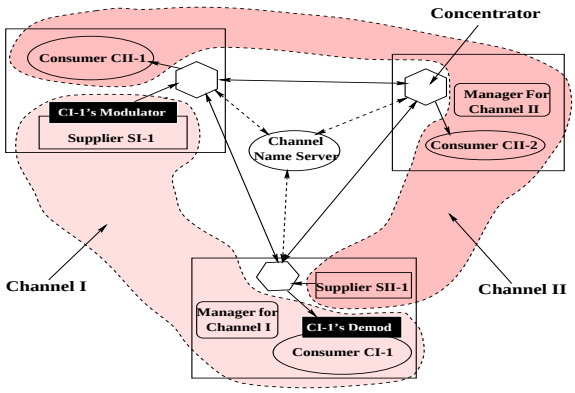

FIGURE 2. Components of JECho Distributed Event System.(Modulator and Demodulator are explained later)

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.

4.1 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.

4.1.1 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 this 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 meta-data, thereby distributing meta-data generation and storage across multiple managers. 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. The name server's address is the IP address (and TCP port number) of the channel name server, and the channel name is a user-defined string. This scheme helps avoid naming conflicts in a large-scale system as a system can deploy multiple independent name servers.

4.1.2 Optimizing/Customizing Object Serialization

Object serialization accounts for a large portion of communication cost in Java. Similar to more efficient RMI implementations[26], JECho improves communication performance with optimizations like using persistent stream state, eliminating extra layers of buffering, and customizing commonly used objects (please refer to [28] for a detailed description of JECho's optimizations). In addition, JECho supports object serialization for embedded Java environments where standard serialization is not available, although it does support features like distributed garbage collection and network classloading in such environment.

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 serialize the event multiple times, JECho serializes the event only once and then sends the resulting byte array directly through sockets. The benefits of this are obvious when sending complex objects to multiple destinations.

4.1.3 Flexible Event Delivery

Collaborative applications, as well as multimedia or sensor processing codes running in wireless domains, are typically comprised of multiple sequences of code modules operating on streaming data. These pipeline/graph-structured applications expect that different code modules 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. Synchronous event delivery, however, offers stronger 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).

4.2 Implementation of Eager Handlers

The idea of eager handlers 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. More detailed descriptions of MOE appear in [28].

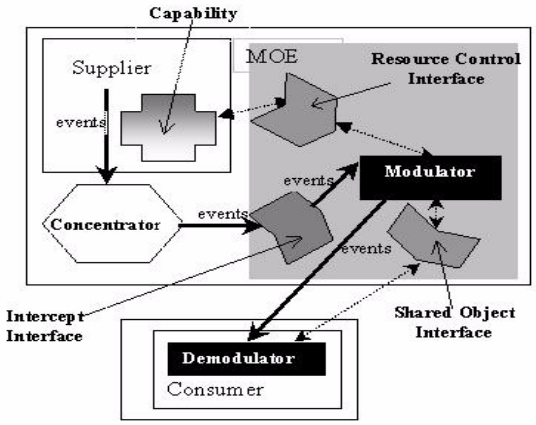

FIGURE 3. MOE Architecture

Given the MOE facility in 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 provide 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 is described in detail in [28].

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 (a commercial product from ObjectSpace) messaging system, although Voyager provides functionality in addition to basic messaging[20]. Results show that JECho's performance exceeds that of RMI and Voyager, sometimes by substantial margins, thereby demonstrating that JECho, and thus Java, can potentially support large-scale applications.

5.1 Simple Case Latency and Throughput

This set of experiments compares the unicast end-to-end latency of Java ObjectStream, Java RMI, JECho ObjectStream and JECho synchronous event delivery (see Table 1). JECho has better performance because it uses persistent -state object stream, and it has lower base runtime overhead because it eliminates extra layers of buffering and because it internally customizes serialization for commonly used objects[28].

Table 1 also lists the corresponding throughput rates for JECho's asynchronous event delivery. JECho Async outperforms all others because it uses one-way messaging and because it batches events (see Section 4.1.3).

5.2 Multi-sink Throughput and Latency

Figure 4 shows the measurement numbers for JECho Sync, JECho Async, RMI and Voyager multicast one-way messaging for varying numbers 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:

TRMI(n,o)=TRMI(1,o)+(n-1)*TOS(1,byte[sizeof(o)]),

where TRMI(n, o) is the latency for RMI to send object o to n sinks, TOS(n, o) is the roundtrip latency of the standard object stream. Note that we use 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 is reused and sent to all other sinks, exactly as with the current implementation of JECho. 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

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.)

Object Types	ObjectStream (JDK1.3, reset)	RMI (JDk1.3, NO reset)	RMI (JDK1.3)	JECho ObjectStream	JECho Sync	JECho Async[a]
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

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

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 event throughput rates (50+ times better for 'null', and 18+ times better for 'composite' objects). JECho Async also experienced less overhead for each additional sink. For instance, for '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 these performance disparities are caused by two factors: (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 like fault-tolerance, which JECho lacks.

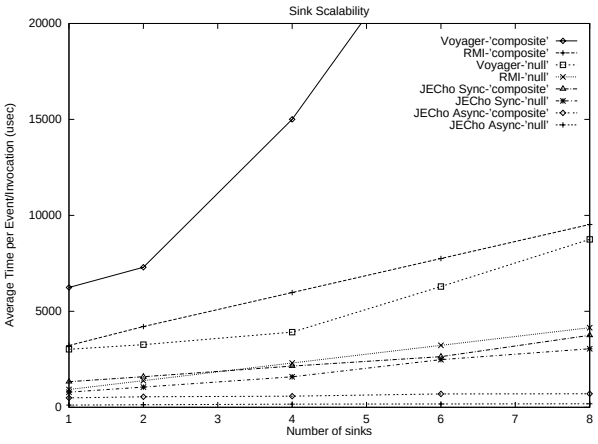

FIGURE 4. Average Time (in usec) for Sending an Event/Invocation for Different Number of Sinks

5.3 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 a single event traverses 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. As shown in the figure, JECho Async's curves are relatively flat after pipeline length of 2.

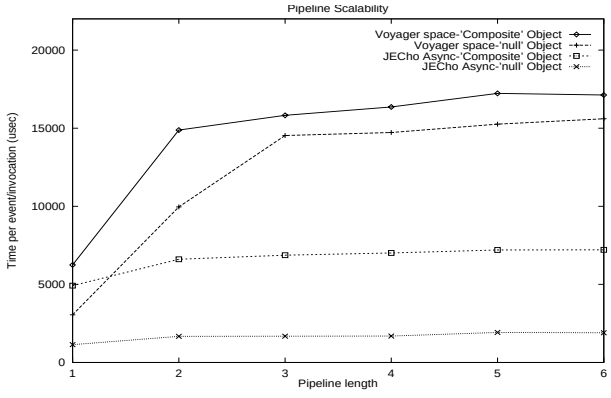

FIGURE 5. Average Time (in usec) for an Event/Invocation to Travel Through a Pipeline of Components, with Changing Pipeline Length (JECho numbers are timed by 10 for the ease of comparison)

5.4 Multi-channel Throughput

Larger scale applications may use a large number of logical channels, reflecting their complex control and data transmission structures. Figure 6 depicts JECho Async's

throughput rate under varying numbers 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.

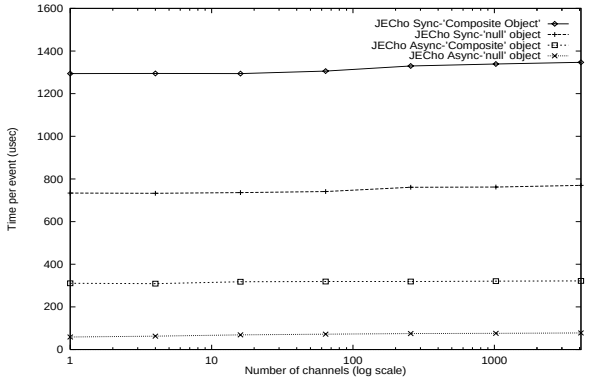

FIGURE 6. Average Time (in usec) for Sending an Event Using Different Numbers of Channels.

5.5 Costs/Benefits of Eager Handlers

5.5.1 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 one of our experiments in [28], an update to a certain shared object has a latency of about 0.5ms when there is one supplier.
- *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 classloader and hence is out of JECho's control. 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).

5.5.2 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.

6. Related Work

There has been a considerable work on high performance messaging in Java[21][22][24][25]. Some of these systems are native-code libraries with Java interfaces[21][24], while pure Java systems have performance limits, especially concerning roundtrip latencies[23][25].

Jini[12]'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[11] and Voyager[20] are also based on unicast remote method invocations. While these systems provide features like 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[27] is a content-based publish/subscribe system that implements the JMS distributed messaging system specification[17]. 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 handlers may be used to implement arbitrary and application-depending routing, transformation, and filtering strategies.

The use of code migration for performance improvement is not novel. In particular, some database systems[18] 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, removed, and modified 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 higher throughput than other pure-Java based communication systems. Our results and examples using eager handlers also show that JECho is lightweight, scalable and adaptable, thereby making it a useful basis for creating the large-scale, heterogeneous communication infrastructures required for collaborative HPC and cluster applications.

Our future work will focus on improving MOE (such as designing an efficient consistency control protocol customized for high performance event communication systems) and on automating the process of eager handler generation with the help of program analysis.

References

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

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

[3] B. Plale, G. Eisenhauer, K. Schwan, J. Heiner, V. Martin and J. Vetter, "From Interactive Applications to Distributed Laboratories", IEEE Concurrency, Vol. 6, No. 2, 1998.

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

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

[6] G. Eisenhauer and K. 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.

[7] W. Ribarsky, Y. Jean, T. Kindler, W. Gu, G. Eisenhauer, K. Schwan and F. Alyea, "An Integrated Approach for Steering, Visualization, and Analysis of Atmospheric Simulations", Proceedings IEEE Visualization '95, 1995.

[8] NCSA Environmental Hydrology Demo, http://scrap.ssec.wisc.edu/~rob/sc98.

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

[10] C. Isert and K. Schwan, "ACDS: Adapting Computational Data Streams for High Performance, Proceedings of IPDPS '00, 2000.

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

[12] Sun Microsystems, "Jini Distributed Event Specification", http://www.sun.com/jini/specs/index.html.

[13] Sun Microsystems", "Remote Method Invocation Specification", http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.

[14] Persistence of Vision, Povray, http://www.povray.org.

[15] D. C. Schmidt and S. Vinoski, "OMG Event Object Service", SIGS, Vol. 9, No. 2, Feb. 1997.

[16] D. C. Schmidt and S. Vinoski, "Object Interconnections: Overcoming Drawbacks with the OMG Events Service", SIGS, Vol. 9, No. 6, June 1997.

[17] Sun Microsystems, "JMS Specification", http://www.sun.com/forte/jmq/documentation/jms-101-spec.pdf.

[18] Oracle, Oracle8i, http://www.oracle.com/database/oracle8i.

[19] National Center for Supercomputing Applications} and University of Illinois at Urbana-Champaign, Habanero, http://havefun.ncsa.uiuc.edu/habanero/.

[20] ObjectSpace Inc., Voyager, http://www.objectspace.com/products/voyager/.

[21] 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.

[22] 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.

[23] N. Yalamanchilli and W. Cohen, "Communication Performance of Java based Parallel Virtual machines". In ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, 1998.

[24] 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.

[25] 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.

[26] C. Nester, M. Philippsen and B. Haumacher, "A more efficient RMI for Java", Proceedings of the ACM 1999 conference on Java Grande, June 1999.

[27] Gryphon, IBM Research, "http://www.research.ibm.com/gryphon/Gryphon_ASSR/gryphon_assr.html".

[28] D. Zhou and K. Schwan, G. Eisenhauer and Y. Chen, "JECho - Supporting Distributed High Performance Applications with Java Event Channels", Tech Report, GIT-CC-00-25, College of Computing, Georgia Institute of Technology.