Event Services in High Performance Systems

Greg Eisenhauer, Fabián E. Bustamante and Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA

The Internet and the Grid are changing the face of high performance computing. Rather than tightly-coupled SPMD-style components running in a single cluster, on a parallel machine, or even on the Internet programmed in MPI, applications are evolving into sets of cooperating components scattered across diverse computational elements. These components may run on different operating systems and hardware platforms and may be written by different organizations in different languages. Complete "applications" are constructed by assembling these components in a plug-and-play fashion. This new vision for high performance computing demands features and characteristics not easily provided by traditional high-performance communications middleware. In response to these needs, we have developed ECho, a high-performance event-delivery middleware that meets the new demands of the Grid environment. ECho provides efficient binary transmission of event data with unique features that support data-type discovery and enterprise-scale application evolution. We present measurements detailing ECho's performance to show that ECho significantly outperforms other systems intended to provide this functionality and provides throughput and latency comparable to the most efficient middleware infrastructures available.

1. Introduction

Wide area distributed computing has been a strong focus of research in high performance computing, resulting in the development of software infrastructures like PVM, MPI, and Globus, and in the creation of the National Machine Room and the Grid by DOE and NCSA/Alliance researchers. Increasingly, research focus in this domain has turned towards component architectures [2] which facilitate the development of complex applications by allowing the creation of generic reusable components and by easing independent component development. Some of the earliest requirements for component architectures in high-performance computing were derived from systems that attach scientific visualizations to running computations, but continuing research has generalized such models to include the ability to flexibly link general purpose computational elements as well [22,24,2]. Component-based software development has been proposed by the software engineering community over the last decade [25,29] and its advantages have been widely recognized in industry, resulting in the development of systems such as Enterprise Java Beans, Microsoft's Component Object Model and its distributed extension (DCOM), and the developing specification of the CORBA Component Model (CCM) in OMG's CORBA version 3.0.

A common technique for integrating the different components of a system is *event-based* invocation, also known as implicit or reactive invocation, which has historical roots in systems based on actors [15], daemons, and packet-switched networks. Event-based integration is attractive as it strongly supports software reuse and facilitates system evolution [12,11]. In bringing the benefits of component-based software development to the domain of high-performance computing, our work does not seek to create a complete component framework. Instead, we have concentrated on providing the integration mechanism that will allow the community to obtain the advantages of such architectures while maintaining high performance.

This paper discusses the results of our work, an efficient event-based middleware, ECho, through which systems of distributed collaborating components can be constructed. Several attributes of ECho distinguish it from related work:

High performance sharing of distributed data. ECho transports distributed data with performance similar to that achieved by systems like MPI. This level of performance is required if the integration mechanism is to support the normally large data flows that are part of high performance applications. For a distributed visualization, for example, this level of performance enables end users to interact via meaningful data sets generated at runtime by the computational models being employed.

This paper demonstrates ECho's high performance across heterogeneous hardware platforms, using networked machines resident at Georgia Tech. In previous work, we have used ECho in Internet-wide collaborations[17], and we have demonstrated its ability to represent both the control and the data events occurring in distributed computational workbenches.

Dynamic data provision and consumption. ECho supports the publish/subscribe model of communication. Thus new components can be introduced into an ECho-based system simply by registering them to the right set of events in the system, without need for recompilation or re-linking. In addition, components can be dynamically replaced without affecting other components in the system, facilitating system evolution. Event-based publish/subscribe models like the one offered by ECho have become increasingly popular and their utility within a variety of other environments, including Internet- and E-commerce applications[30], extensible systems[23], collaborative systems[14], distributed virtual reality[19] and mobile systems[32], has been well-established. ECho differs from such ongoing or past research in its efficient support for event transmission across heterogeneous machines, derived from its ability to recognize and provide runtime translation for user-defined event formats. While systems like InfoBus[20] and Schooner[16] have demonstrated the utility of making type information available to middleware, neither have attempted to attain the high performance achieved by ECho.

Dynamic type extension and reflection. One of the major features differentiating component-based applications from their tightly-coupled kin is the relative lack of *a priori* knowledge about data flows. In order to be able to "drop" a component into place in a system, the component must be able to discover the contents of the data flows it is to operate upon. Even the parts of an application that were designed to work together face difficulty maintaining *a priori* knowledge in a wide area Grid environment. As different pieces of an application are changed or upgraded over time it may be necessary to modify their data flows, invalidating other pieces that rely on previous knowledge and/or requiring their simultaneous upgrade. Because of these difficulties, component-based systems typically provide an integration mechanism that offer some degree of *type extension* and *reflection*. Those terms, borrowed from object-oriented systems, express the ability to transparently extend existing data types while preserving the validity of code using the old type (type extension) and the ability for third parties to discover the contents of and operate upon a data type without *a priori* knowledge (reflection). One of the most important contributions of ECho is that it provides these features without compromising performance, as measurements in this paper will demonstrate.

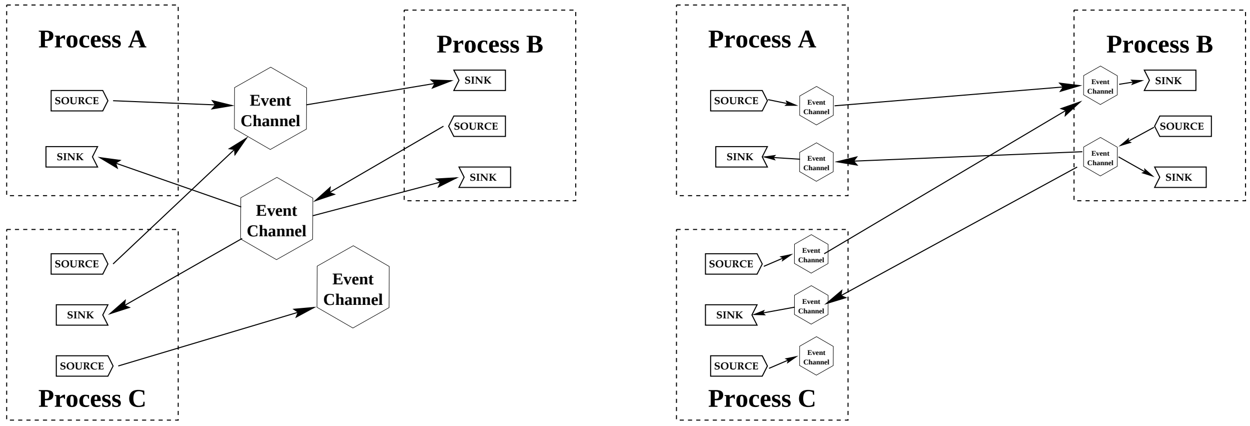
Interoperability. ECho-based applications can also interoperate with CORBA- or Java-based components, like those used in the Diesel Combustion Collaboratory or the Hydrology workbench. Thus, end users can continue to employ tools like the Java-based VisAD data visualization system or the CORBA-based collaboration services in Deepview, but gain high performance for data movement (in contrast to event rates attained for CORBA- or Java-based event systems[3,31]). Interoperability with Java- and CORBA-based systems will be demonstrated elsewhere.

ECho has been available since October 1997, and our group has used it for various large-scale, ongoing development and research efforts. Among such efforts, of principal interest to the high performance community are the atmospheric and hydrology applications mentioned earlier as well as two additional ones now being developed by our group: (1) a distributed materials design workbench, where multiple end users interact with each other and with computational tools in order to jointly design high performance materials, and (2) a distributed implementation of an NT-Unix-spanning system for molecular dynamics and/or for crystal plasticity studies done by collaborators in the departments of Mechanical Engineering and Physics in Georgia Tech. Finally, ECho events are one of the key building blocks of the DARPA-funded Infosphere Information Technology Expedition[26].

The remainder of this paper is organized as follows. Section 2 describes ECho's basic functionality. Section 3 compares ECho's event delivery performance to that of other communication systems which offer some form of type extension and reflection. In particular, we examine the performance of a set of middleware systems which might be considered as alternative candidates for the integration mechanism of a component infrastructure, including CORBA event channels, event distribution via Java's RMI, and an XML-based communication scheme; comparing the basic latency of each to that of ECho and using an MPI message exchange as a baseline for measurement. We also study the impact of machine heterogeneity on ECho's performance and explore the effects of its type extension features. Finally, Section 4 discusses some key areas of future work and summarizes our conclusions.

2. ECho Functionality

ECho shares semantics common to a class of event delivery systems that use *channel-based subscriptions*. That is, an *event channel* is the mechanism through which event sinks and sources are matched. Source clients submit events to a specific channel and only the sink clients subscribed to that channel are notified of the event. Channels are essentially entities through which the extent of event propagation is controlled. The CORBA Event Service[13] is also channel-based, with channels being distributed objects.



(a) Abstract view of Event Channels.

(b) ECho Realization of Event Channels.

Figure 1. Using Event Channels for Communication.

2.1. Efficient Event Propagation

Unlike many CORBA event implementations and other event services such as Elvin[28], ECho event channels are not centralized in any way. Instead, channels are light-weight virtual entities. Figure 1a depicts a set of processes communicating using event channels. The event channels are shown as existing in the space between processes, but in practice they are distributed entities, with bookkeeping data residing in each process where they are referenced as depicted in Figure 1b. Channels are *created* once by some process, and *opened* anywhere else they are used. The process which creates the event channel is distinguished, in that it is the contact point for other processes wishing to use the channel. The channel ID, which must be used to open the channel, contains the contact information for the creating process (as well as information identifying the specific channel). However, event distribution is not centralized and there are no distinguished processes during event propagation. Event messages are always sent directly from an event source to all sinks and network traffic for individual channels is multiplexed over shared communications links.

ECho is implemented on top of DataExchange[10] and PBIO[8], packages developed at Georgia Tech to simplify connection management and heterogeneous binary data transfer. As such, it inherits from these packages portability to different network transport layers and threads packages. DataExchange and PBIO operate across the various versions of Unix and Windows NT, have been used over the TCP/IP, UDP, and ATM communication protocols and across both standard and specialized network links like ScramNet[6].

In addition to offering interprocess event delivery, ECho also provides mechanisms for associating threads with event handlers allowing a form of intra-process communication. Local and remote sinks may both appear on a channel, allowing inter- and intra-process communication to be freely mixed in a manner that is transparent to the event sender. When sources and sinks are within the same address space, an event is delivered by directly placing the event into the appropriate shared-memory dispatch queue. While this intra-process delivery can be valuable, this paper concentrates on the aspects of ECho relating to remote delivery of events.

2.2. Event Types and Typed Channels

One of the differentiating characteristics of ECho is its support for efficient transmission and handling of fully typed events. Some event delivery systems leave event data marshalling to the application. ECho allows types to be associated with event channels, sinks and sources and will automatically handle heterogeneous data transfer issues. Building this functionality into ECho using PBIO allows for efficient layering that nearly eliminates data copies during marshalling and unmarshalling. As others have noted[18], careful layering to minimize data copies is critical to delivering full network bandwidth to higher levels of software abstraction. The layering with PBIO is a key feature of ECho that makes it suitable for applications which demand high performance for large amounts of data.

Base Type Handling and Optimization. Functionally, ECho event types are most similar to user defined types in MPI. The main differences are in expressive power and implementation. Like MPI's user defined types, ECho event types describe C-style structures made up of atomic data types. Both systems support nested structures and statically-sized arrays. ECho's type sys-

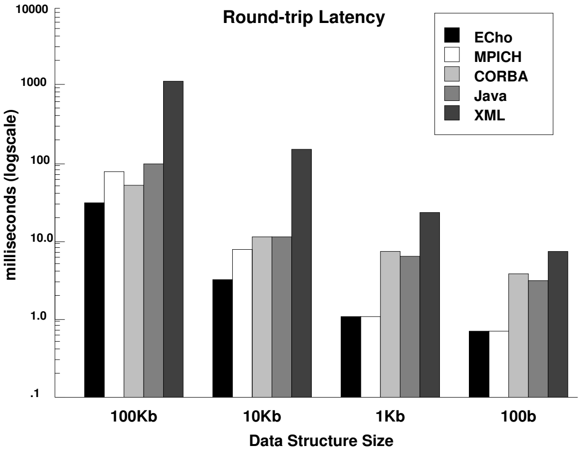


Figure 2. A comparison of latency in basic data exchange in event infrastructures

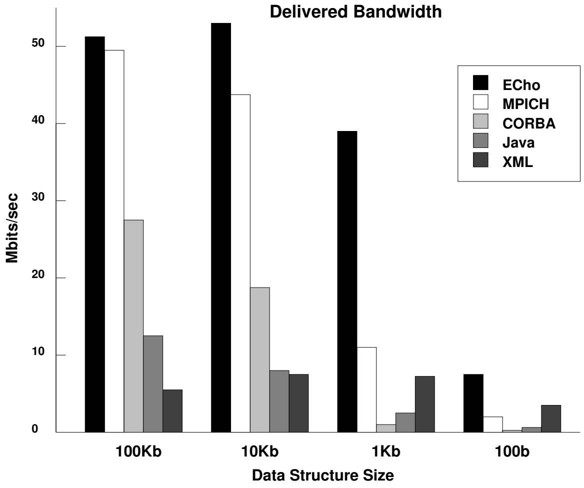


Figure 3. A comparison of delivered bandwidth in event infrastructures

tems extends this to support null-terminated strings and dynamically sized arrays.[1]

While fully declaring message types to the underlying communication system gives the system the opportunity to optimize their transport, MPI implementations typically do not exploit this opportunity and often transport user defined types even more slowly than messages directly marshalled by the application. In contrast, ECho and PBIO achieve a performance advantage by avoiding XDR, IIOP or other 'wire' representations different than the native representation of the data type. Instead, ECho and PBIO use a wire format that is equivalent to the native data representation (NDR) of the sender. Conversion to the native representation of the receiver is done upon receipt with dynamically generated conversion routines. As the measurements in [4] show, PBIO 'encode' times do not vary with data size and 'decode' times are much faster than MPI. Because as much as two-thirds of the latency in a heterogeneous message exchange is software conversion overhead[4], PBIO's NDR approach yields round-trip message latencies as low as 40% of that of MPI.

Type Extension. ECho supports the robust evolution of sets of programs communicating with events, by allowing variation in the data types associated with a single channel. In particular, an event source may submit an event whose type is a superset of the event type associated with its channel. Conversely, an event sink may have a type that is a subset of the event type associated with its channel. Essentially this allows a new field to be added to an event at the source without invalidating existing event receivers. This functionality can be extremely valuable when a system evolves because it means that event contents can be changed without the need to simultaneously upgrade every component to accommodate the new type. ECho even allows type variation in intra-process communication, imposing no conversions when source and sink use identical types but performing the necessary transformations when source and sink types differ in content or layout.

[1] In the case of dynamically sized arrays, the array size is given by an integer-typed field in the record. Full information about the types supported by ECho and PBIO can be found in [8].

The type variation allowed in ECho differs from that supported by message passing systems and intra-address space event systems. For example, the Spin event system [23] supports only statically typed events. Similarly, MPI's user defined type interfaces do not offer any mechanisms through which a program can interpret a message without *a priori* knowledge of its contents. Additionally, MPI performs strict type matching on message sends and receives, specifically prohibiting the type variation that ECho allows.

In terms of the flexibility offered to applications, ECho's features most closely resemble the features of systems that support the marshalling of objects as messages. In these systems, subclassing and type extension provide support for robust system evolution that is substantively similar to that provided by ECho's type variation. However, object-based marshalling often suffers from prohibitively poor performance. ECho's strength is that it maintains the application integration advantages of object-based systems while significantly outperforming them. As the measurements in the next section will show, ECho also outperforms more traditional message-passing systems in many circumstances.

3. ECho Performance

3.1. Breakdown of Costs

Figures 2 and 3 represent the basic performance characteristics of a variety of communication infrastructures that might be used for event-based communication in high performance applications. The values are of basic event latency and bandwidth in an environment con-

	ECho	CORBA (ORBacus)	MPICH	XML
Total Round-Trip	30.6	53.0	80.1	1249
Sparc Encode	0.037	0.74	13.3	176
Network Transfer	13.9	13.9	13.9	182
x86 Decode	1.6	1.6	11.6	276
x86 Encode	0.015	0.64	8.9	124
Network Transfer	13.9	13.9	13.9	182
Sparc Decode	1.2	0.58	15.4	486

Table 1
Cost breakdown for heterogeneous 100Kb event exchange (times are in milliseconds).

sisting of a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[2] Note the use of a logarithmic vertical scale in Figure 2. This is useful in presenting latencies for a range of message sizes on the same graph, but it tends to minimize the substantial performance advantage that ECho demonstrates as compared to the other infrastructures.

The infrastructures compared don't all share the same characteristics and features, a fact that accounts for some of their performance differences. ECho's strength is that it provides the important features of these systems while maintaining the performance achieved by traditional high-performance systems like MPICH.

In particular, ECho provides for event type discovery and dynamic type extension in a manner similar to that of XML, or that which can be achieved by serializing objects as events (as in Java RMI). CORBA is also gaining acceptance as distributed systems middleware and its Event Services provide similar features. This section will examine ECho's performance characteristics in more detail and contrast them with these other infrastructures.

Table 1 shows a breakdown of costs involved in the roundtrip event latency measures of Figure 2. We present round-trip times because they naturally show all the combinations of send/recv on two different architectures in a heterogeneous system. The time components labeled "Encode" represent the span of time between an application submitting data for transmission and the point at which the infrastructure invokes the underlying network 'send()' operation. The "Network Transfer" times are the one-way times to transmit the encoded data from sending to receiving machines. The "Decode" times are the time between the end of the "recv()" operation and the point at which the data is presented to the application in a usable form. This breakdown is useful for understanding the different costs of the communication and in particular, how they might change with different networks or processors.

[2] The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

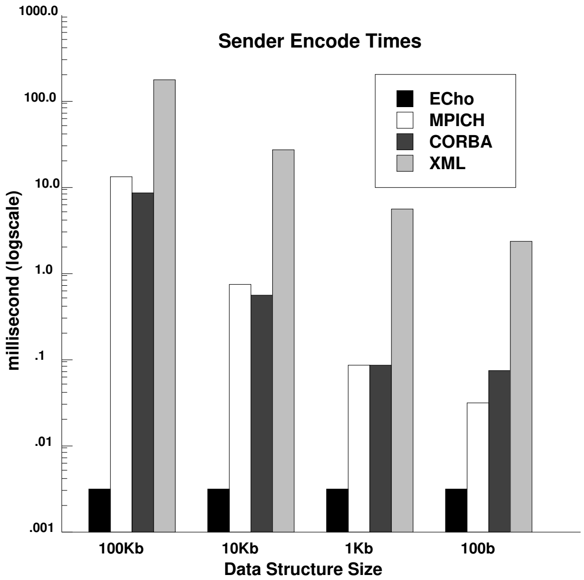


Figure 4. Send-side data encoding times.

We have excluded Java RMI from the breakdown in Table 1 because it performs its network 'send()' operations incrementally during the marshalling process. This allows Java to pipeline the encode and network send operations making a simple cost breakdown impossible. However, as a result of this design decision, Java RMI requires tens of thousands of kernel calls to send a 100Kb message, seriously impacting performance.

Additionally, while the round-trip times listed in Table 1 are near the sum of the encode/xmit/decode times, this is not true for the CORBA numbers. This is because implementations of the CORBA typed event channel service typically rely on CORBA's dynamic invocation interface to operate. In the ORBs we have examined, DII does not function for intra-address-space invocations. The result of this is that the CORBA typed event channel must reside in a different address space than either the event source or event sink, adding an extra hop to every event delivery. This could be considered an implementation artifact that might be handled differently in future CORBA event implementations.

3.1.1. Sending side costs

ECho's most significant performance feature is its use of the native data format on the sending architecture as its 'wire format'. The effects of this approach are most noticeable when comparing the "Encode" times for the different communication infrastructures. Figure 4 expands upon the summary data of Table 1 and shows sender-side data encoding costs for a variety of communications systems. For example, MPICH uses a very slow interpreted marshalling procedure for heterogeneous communication of MPI user-defined data types. That this has a significant impact on MPICH performance is apparent in Table 1 which shows MPICH devoting as much as 60% of its round-trip message time to encoding and decoding.

CORBA's IIOP wire format differs from the architectures' native data layout in its alignment requirements. As a result, CORBA must copy all application data before sending. In ORBacus, this copy is performed by compile-time-generated stub code, resulting in better performance than with the MPICH approach. However, ECho is significantly faster because it performs very little processing prior to the network send operation.

Using XML as a wire format is obviously a decision which has a significant performance impact on an event system. Table 1 makes clear two of the most significant issues: the large encode/decode times, and the expanded network transmission times. The former is a result of the distance between the ASCII representation used by XML and the native binary data representation. XML encoding costs represent the processing necessary to convert the data from binary to string form and to copy the element begin/end blocks into the output string. Just one end of the encoding time for XML is several times as expensive as the entire round-trip message exchange for the other infrastructures. Network transmission time is also significantly higher for XML because the ASCII-encoded data (plus the begin/end labels) can be much larger than the equivalent binary representation. How much larger depends upon the data, the size of the field labels and other details in the encoding. Thus, XML-based schemes transmit more data than schemes which rely on binary encoding.

3.2. Receiving side costs

ECho's NDR approach to binary data exchange eliminates sender-side processing by transmitting in the sender's native format and isolating the complexity of managing heterogeneity in the receiver. As a result, the receiver must perform conversion of the various incoming 'wire' formats to its 'native' format. Such a conversion may require byte-order changes (byte-swapping), movement of data from one offset to another, or even a change in the basic size of the data type (for example, from a 4-byte integer to an 8-byte integer).

This conversion is another form of the "marshaling problem" that occurs widely in RPC implementations[1] and in network communication. Marshaling can be a significant overhead[7,27], and tools like USC[21] attempt to optimize marshaling with compile-time solutions. Unfortunately, the dynamic form of the marshaling problem in ECho, where the layout and even the complete field contents of the incoming record are unknown until run-time, rules out such static solutions. The conversion overhead is nil for some homogeneous data exchanges, but as Table 1 shows, can be significantly high for some heterogeneous exchanges (up to 66%).

Generically, receiver-side overhead in communication middleware has several components:

- byte-order conversion,
- data movement costs, *and*
- control costs.

Byte order conversion costs are to some extent unavoidable. If the communicating machines use different byte orders, the translation must be performed somewhere regardless of the capabilities of the communications package.

Data movement costs are harder to quantify. If byte-swapping is necessary, data movement can be performed as part of the process without incurring significant additional costs. Otherwise, clever design of the communications middleware can often avoid copying data. However, packages that define a 'wire format' for transmitted data have a harder time being clever in this area. One of the basic difficulties is that the native format for mixed-datatype structures on most architectures has gaps, unused areas between fields, inserted by the compiler to satisfy data alignment requirements. To avoid making assumptions about the alignment requirements of the machines they run on, most packages use wire formats which are fully packed and have no gaps. This mismatch *forces* a data copy operation in situations where a clever communications system might otherwise have avoided it.

Control costs represent the overhead of iterating through the fields in the record and deciding what to do next. Packages that require the application to marshal and unmarshal their own data have the advantage that this process occurs in special-purpose compiler-optimized code, minimizing control costs. However, to keep that code simple and portable, such systems uniformly rely on communicating in a pre-defined wire format, therefore incurring the data movement costs described in the previous paragraph.

Packages that marshal data themselves typically use an alternative approach to control, where the marshaling process is controlled by what amounts to a table-driven interpreter. This interpreter marshals or unmarshals application-defined data, making data movement and conversion decisions based upon a description of the structure provided by the application and its knowledge of the format of the incoming record. This approach to data conversion gives the package significant flexibility in reacting to changes in the incoming data and was our initial choice when implementing the PBIO technology ECho is based on.

XML necessarily takes a different approach to receiver-side decoding. Because the 'wire' format is a continuous string, XML is parsed at the receiving end. The Expat

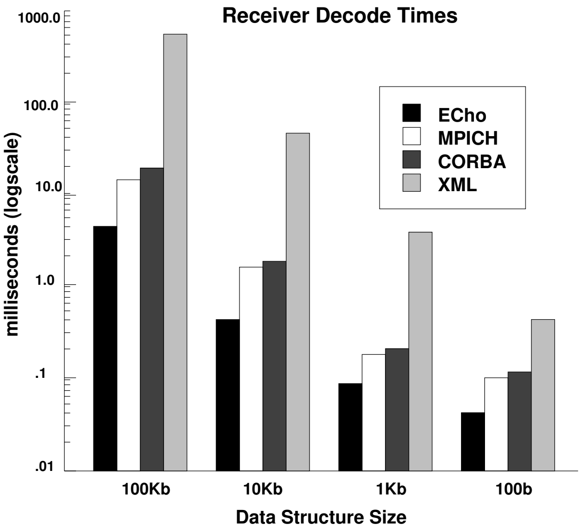


Figure 5. Receiver side costs for XML, MPI and NDR. *(Logarithmic scale used in vertical dimension)*

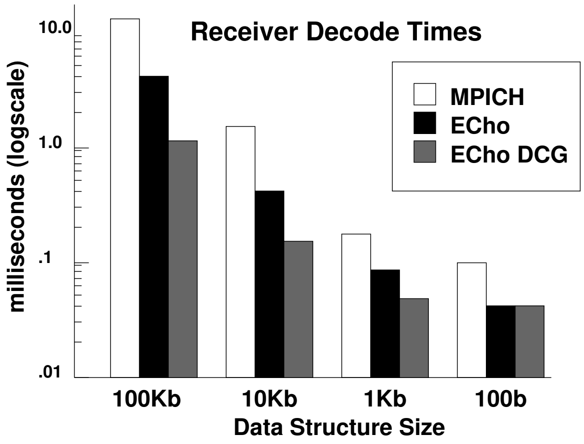


Figure 6. Receiver side costs for interpreted conversions in MPI and ECho and DCG conversions in ECho.

XML parser[3] [5] calls handler routines for every data element in the XML stream. That handler can interpret the element name, convert the data value from a string to the appropriate binary type and store it in the appropriate place. This flexibility makes XML extremely robust to changes in the incoming record. The parser we have employed is quite fast, but XML still pays a relatively heavy penalty for requiring string-to-binary conversion on the receiving side. (We assume that for most high performance computing functions, data is being sent somewhere for processing and that processing requires the event data to be in other than string form. Thus, XML decoding is not just parsing, but also the equivalent of a C `strtod()` or similar operation to convert the data into native representation.)

Comparison of receiver-side costs for XML, ECho, and non-optimized ECho wire formats. Figure 5 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by XML, MPICH (via the `MPI_Unpack()`) call, and NDR. XML receiver conversions are clearly expensive, typically between one and two orders of decimal magnitude more costly than our NDR-based converter for this heterogeneous exchange. (For an exchange between homogeneous architectures, ECho and MPI would have substantially lower costs, while XML's costs would remain unchanged.) Our NDR-based converter is relatively heavily optimized and performs considerably better than MPI, in part because MPICH uses a separate buffer for the unpacked message rather than reusing the receive buffer (as we do). However, ECho's receiver-side conversion costs still contribute roughly 20% of the cost of an end-to-end message exchange. While a portion of this conversion overhead must be the consequence of the raw number of operations involved in performing the data conversion, a significant fraction of this overhead is due to the fact that the conversion is essentially being performed by an interpreter.

Optimizing receiver-side costs for ECho. Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time, requiring a somewhat costly interpreted conversion. Our solution to this problem to the problem is to employ dynamic code generation to create a customized conversion subroutine for every incoming record type[4]. These routines are generated by the receiver on the fly, as soon as the wire format is known.

The execution times for these dynamically generated conversion routines are shown in Figure 6. (We have chosen to leave the XML conversion times off of this figure to keep the scale to a manageable size. Again, please note the use of logarithmic scale.) The dynamically generated conversion routine operates significantly faster than the interpreted version. This improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation. In particular, without dynamic code generation for conversion routines ECho's practice of eliminating the sender-side encoding cost by transmitting in the sender's native format might not be viable. It is the combination of these two techniques,

- transmitting in the sender's native format *and*
- using dynamic code generation for conversion routines on the receiving side,

that is the key to ECho's efficiency. The PBIO binary code generation facilities exploited by ECho exist for most popular machine architectures, including Sparc,

[3] A variety of implementations of XML, including both XML generators and parsers, are available. We have used the fastest known to us at this time, Expat [5].

[4] More details on the nature of the ECho infrastructure dynamic code generation can be found in [9].

	ORBacus		ECho	
Data size	Send side overhead	Receive side overhead	Send side overhead	Receive side overhead
100Kb	0.74	0.40	0.037	0.034
10Kb	0.22	0.046	0.037	0.034
1Kb	0.19	0.016	0.037	0.034
100b	0.17	0.010	0.037	0.034

Table 2
Cost breakdown for homogeneous event exchange. *(Times are in milliseconds.)*

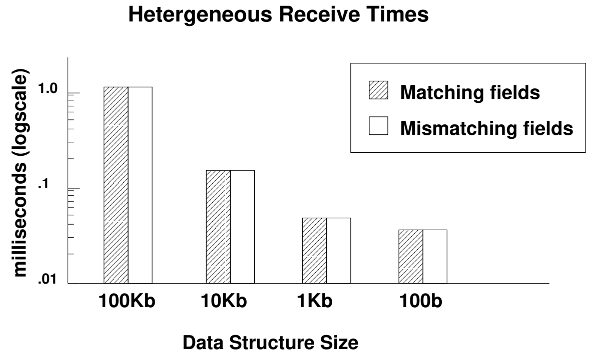


Figure 7. Receiver-side decoding costs with and without an unexpected field – heterogeneous case.

MIPS, x86, and i960 machines.

3.3. Costs for Homogeneous Exchanges

Because ECho has virtually no sender-side encoding costs and because its dynamic code generation achieves performance similar to that achieved through compile-time stub generation, ECho tends to outperform other communication infrastructures. This is particularly apparent in heterogeneous message exchanges because the encode/decode time can play a significant role in overall message costs.

However, ECho's approach also yields performance gains for transfers between homogeneous systems, as shown in Table 2. For simplicity, this table concentrates on the ECho and ORBacus infrastructures. The higher ORBacus costs for large data sizes represent the cost of the required data copy in converting the IIOP wire format to the native data representation. ECho requires no such copy.[5] As in the heterogeneous case, ECho does not pre-process data prior to sending, and because the 'wire format' corresponds to the native data representation, ECho can deliver received data directly to the application without copying it from the message buffer. This is not possible with IIOP because of potential data alignment conflicts between IIOP and the native data representation.

At common 100Mbps network speeds, these additional data copy operations are account for a relatively small fraction of the total exchange costs. However, minimizing data copies is critical to delivering full network bandwidth to higher levels of software abstraction[18]. As gigabit networks and specialized low-latency communications mechanisms come into more common use, the additional copy operations imposed on even homogeneous communications by fixed wire formats will become a more important limitation on communication speeds, increasing ECho's performance advantage.

[5] For the smaller data sizes, the extra copy overhead is small compared to the fixed delivery costs in these systems.

3.4. Costs for Type Extension

In addition to efficient operation in basic event transfer, ECho supports the creation and evolution of sets of collaborating programs through event type discovery and dynamic type extension. ECho events carry format meta-information, somewhat like an XML-style description of the message content. This meta-information can be an incredibly useful tool in building and deploying enterprise-level distributed systems because it (1) allows generic components to operate upon data about which they have no *a priori* knowledge, and (2) allows the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other terms, ECho allows *reflection* and *type extension*. Both of these are valuable features commonly associated with object systems.

ECho data type information is represented during transmission with *format tokens* which can be used to retrieve full type information. These tokens are small and are included in every ECho event transmission as part of the header information. As such they do not affect performance significantly.

ECho supports type extension by virtue of doing field matching between incoming and expected records by name. Because of this, new fields can be added to events without disruption because application components which don't expect the new fields will simply ignore them.

Most systems which support reflection and type extension in messaging, such as systems which use XML as a wire format or which marshal objects as messages, suffer prohibitively poor performance compared to systems such as MPICH and CORBA which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon ECho performance. In particular, we measure the performance effect of type extension by introducing an unexpected field into the incoming message and measuring the change in receiver-side processing.

Figures 7 and 8 present receive-side processing costs

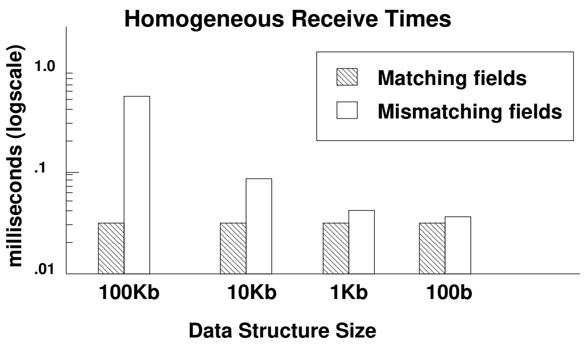


Figure 8. Receiver-side decoding costs with and without an unexpected field – homogeneous case.

for an exchange of data with an unexpected field. These figures show values measured on the Sparc side of heterogeneous and homogeneous exchanges, respectively, using ECho's dynamic code generation facilities to create conversion routines. It's clear from Figure 7 that the extra field has no effect upon the receive-side performance. Transmitting would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 8 shows the effect of the presence of an unexpected field in the homogeneous case. Here, the overhead is potentially significant because the homogeneous case normally imposes no conversion overhead in ECho. The presence of the unexpected field creates a layout mismatch between the wire and native record formats and as a result the conversion routine must relocate the fields. As the figure shows, the resulting overhead is non-negligible, but not as high as exists in the heterogeneous case. For smaller record sizes, most of the cost of receiving data is actually caused by the overhead of the kernel `select()` call. The difference between the overheads for matching and extra field cases is roughly comparable to the cost of `memcpy()` operation for the same amount of data.

The results shown in Figure 8 are actually based upon a worst-case assumption, where an unexpected field appears before all expected fields in the record, causing field offset mismatches in all expected fields. In general, the overhead imposed by a mismatch varies proportionally with the extent of the mismatch. An evolving application might exploit this feature of ECho by adding any additional at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

4. Conclusions and Future Work

This paper examines ECho, an event-based middleware designed to meet the demands of a new generation of Grid applications. In particular, we consider the communication/integration demands of component-based systems in a high-performance computing environment and how they might be different from those of more tightly-coupled applications. ECho meets those requirements by providing a publish-subscribe communication model that supports type extension and type discovery. While object-based and XML-based systems provide similar functionality, the measurements in Section 3 show that ECho does it with significantly better performance, both in terms of delivered bandwidth and end-to-end latency. The measurements also show that ECho matches and, in most cases, outperforms MPICH in both metrics supporting our assertion that ECho is suitable for use in the main data flows of Grid applications.

Future work will examine aspects of ECho which are beyond the scope of this paper. Those features include *derived event channels*, which support for source-side event filtering and remote data transformation, and *proto-channels*, a mechanism through which receivers can themselves control and customize source-side event generation. We will also expand upon ECho's ties to other systems, including CORBA and Java.

References

[1] Guy T. Almes. The impact of language and system on remote procedure call design. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 414–421. IEEE, May 1986.

[2] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings of the 8th High Performance Distributed Computing (HPDC8)*, 1999. http://www.acl.lanl.gov/cca.

[3] Yeturu Ashlad, Bruce E. Martin, Mod Marathe, and Chung Le. Asynchronous notifications among distributed objects. In *Proceedings of the Conference on Object-Oriented Technologies and Systems.* Usenix Association, June 1996.

[4] Fabián E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Proc. of Supercomputing 2000 (SC 2000)*, Dallas, TX, November 2000.

[5] James Clark. expat - XML parser toolkit. http://www.jclark.com/xml/expat.html.

[6] Systran Federal Corporation. Scramnet networks. http://www.systran.com/realtim.htm.

[7] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Synposium*, pages 200–208, Sept 1990.

[8] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[9] Greg Eisenhauer and Lynn K. Daley. Fast heterogeneous binary data interchange. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pages 90–101, Cancun, Mexico, May 2000.

[10] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, 24(12-13):1713–1733, November 1998.

[11] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Soft-*

ware Development Methods, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[12] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I*. World Scientific Publishing Company, New Jersey, 1993.

[13] Object Management Group. *CORBAservices: Common Object Services Specification*, chapter 4. OMG, 1997. http://www.omg.org.

[14] Habanero. NCSA and University of Illinois at Urbana. http://notme.ncsa.uiuc.edu/SDG/Software/Habanero.

[15] C. Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the First International Join Conference in Artificial Intelligence*, 1969.

[16] Patrick T. Homer and Richard D. Schlichting. Configuring scientific applications in a heterogeneous distributed system. *IEEE Distirbuted Systems Engineering Journal*, 1996 1996.

[17] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[18] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7)*, July 1998.

[19] Blaire MacIntyre and Steven Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proceedings of Symposium on User Interface Software and Technology (UIST'96)*, pages 83–95, November 1996.

[20] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus - an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[21] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[22] Carmen M. Pancerella, Larry A. Rahn, and Christine L. Yang. The diesel combustion collaboratory: Combustion researchers collaborating over the internet. In *Proceedings of SC 99*, November 13-19 1999. http://www.sc99.org/proceedings/papers/pancerel.pdf.

[23] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, Seattle, WA, October 1992. USENIX.

[24] Bahram Parvin, John Taylor, Ge Cong, Michael O'Keefe, and Mary-Helen Barcellos-Hoff. Deepview: A channel for distributed microscopy and informatics. In *Proceedings of SC 99*, November 13-19 1999. http://www.sc99.org/proceedings/papers/parvin.pdf.

[25] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[26] Calton Pu. Infosphere – smart delivery of fresh information. http://www.cse.ogi.edu/sysl/projects/infosphere//

[27] Michael D. Schroeder and Michael Burrows. Performance or Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pages 83–90, Litchfield Park, AZ, December 1989. ACM, SIGOPS.

[28] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the AUUG (Australian users group for Unix and Open Systems) 1997 Conference*, September 1997.

[29] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

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

[31] Sun Microsystems. The jini[tm] distributed event specification, version 1.0.1. Technical report, Sun Microsystems, Nov. 1999. http://www.sun.com/jini/specs/event101.html.

[32] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. Concordia: An infrastructure for collaborating mobile agents. In *Proceedings of the First International Workshop on Mobile Agents 97 (MA'97)*, April 1997.