Submitted to the International Conference on Parallel and Distributed Computing Systems (PDCS 2000)

Fast Heterogeneous Binary Data Interchange for Event-based Monitoring

Beth Plale Greg Eisenhauer Lynn K. Daley
Patrick Widener Karsten Schwan

College of Computing
Georgia Institute of Technology

Abstract

Dramatic increases in available wide-area bandwidth have driven event-based monitoring to new heights. Monitoring services are widely used in today's distributed laboratories, where scientists interact with scientific instruments and collaborate with each other regardless of their, or the instrument's, location.

This paper addresses binary data transfer support in a distributed laboratory. Binary data transfer, a core service of most event-based monitoring approaches, is provided by PBIO (Portable Binary Input/Output). PBIO offers applications significantly more flexibility in message exchange than other approaches in current use. Further, comparison between PBIO and both MPI and XML show PBIO to be a favorable transport mechanism for today's high performance distributed laboratory applications.

*The paper demonstrates the need for fast heterogeneous binary data interchange in large-scale event-based monitoring applications (*e.g. *distributed laboratory) and argues its relevance even in the face of increasing scientist interest in science-centric approaches to data representation.*

1 Introduction

Event-based monitoring is experiencing renewed interest by the high performance computing community as dramatic increases in available bandwidth has enabled event-based monitoring to scale to hundreds of data sources and possibly thousands of user clients. *Event-based monitoring* is the extraction of data from a parallel or distributed application and subsequent analysis or visualization of the data while it is running, standing in contrast to post-mortem analysis of trace files or behavior analysis a debugging.

Where earlier research into event-based monitoring has focussed on performance evaluation and on-line interactivity, increased bandwidth has enabled scaling-up of event-based monitoring from a small group of scientists co-located with scientific instruments (*i.e.*, scientific models, trace files, observational data) to a *distributed laboratory* [20] wherein large numbers of instruments and scientists exist, the resources are heterogeneous, and the restriction of co-location removed.

Our group at Georgia Tech has for some time been investigating event-based monitoring for interactivity in a distributed laboratory. Areas in which work has been done, both by our group and others, are shown in Figure 1. Core services of an event-based monitoring system [2] include instrumenting the source code, buffering the instrumented data, and sending the data to interested clients. These core services have been addressed by such projects as Pablo [22], Paradyn [17], SCIrun [19], Falcon [12], and [8]. Event stamping, the first of the extension services listed, is often provided to enable ordering of events. Clock synchronization algorithms are often used to order events. Program steering, the feedback part of the monitoring-steering loop, enables scientists to control applications (*e.g.*, roll back programs to an earlier state, change parameters). A visualization service might provide an API to a scientific visualization tool (*i.e.,* visAD [14]) so that monitoring events can be received from the communication layer. Finally, as the number of users and the computational power of resources increases, there has been increasing interest in event-based monitoring techniques for data stream control. That is, techniques such as data filtering, data aggregation, or querying techniques [21] that reduce data flow, and hence end-to-end latency, from data generators to clients.

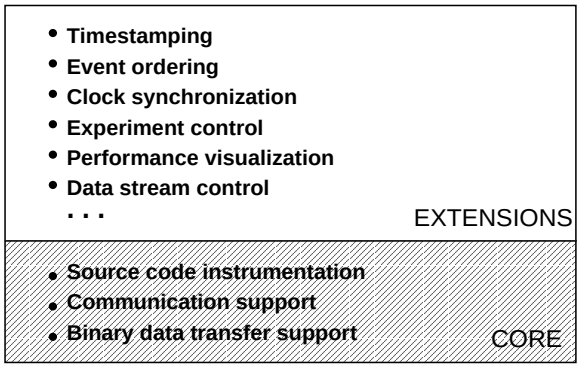


Figure 1: Services of Event-based Monitoring .

In this paper, we focus on the core monitoring service of binary data transfer. Popular approaches to binary data transfer, such as PVM [11] or Nexus [10], support message exchanges in which the communicating applications "pack" and "unpack" messages on a per field, per datatype basis. Other packages, such as MPI [9], allow the creation of user-defined datatypes for messages and message fields and provide some amount of marshalling and unmarshalling support for those datatypes internally.

While an approach requiring the application to build messages manually offers applications significant flexibility in message contents, relegating message packing and unpacking to the communicating applications means that those applications must have *a priori* agreement on the contents and format of messages. This is not an onerous requirement in small-scale stable systems, but in enterprise-scale computing, the need to simultaneously update all application components in order to change message formats can be a significant impediment to the integration, deployment and evolution of complex systems.

In addition, the semantics of application-side pack/unpack operations generally imply a data copy to or from message buffers. Such copies are known[16, 23] to have a significant impact on communication system performance. Packages which can perform internal marshalling, such as MPI, have an opportunity to avoid data copies and to offer more flexible semantics in matching fields provided by senders and receivers. However, existing packages have failed to capitalize on those opportunities.

This paper describes PBIO(Portable Binary Input/Output)[5], a multi-purpose communication middleware. PBIO focuses on flexible heterogeneous binary data transport for simple messaging of a wide range of application data structures, using novel approaches such as dynamic code generation (DCG) to preserve efficiency. In addition, PBIO's flexibility in matching transmitted and expected data types provides key support for *application evolution* that is missing from other communication systems.

The remainder of the paper begins with a description of PBIO. Following the description are performance results of using PBIO across a heterogeneous environment. The metrics are compared against the data communication measurements obtained by using MPI. The paper will show that the features and flexibility of PBIO do not impose overhead beyond that imposed by other communications systems. In the worst case PBIO performs as well as other systems, and in many cases PBIO offers a significant performance improvement over comparable communications packages. Performance results offer a comparison between PBIO and XML as well. We conclude by addressing the current interest of scientists in XML as a standard metalanguage for describing data in terms scientists understand (*e.g.*, atoms, molecules, clusters of molecules) with a discussion of mapping ASCII XML to binary PBIO, ongoing work being done by our group.

Much of PBIO's performance advantage is due to its use of dynamic code generation to optimize transla-

tions from wire to native format. Because this is a novel feature in communications middleware, its impact on PBIO's performance is also considered independently. In this manner, we show that for purposes of data compatibility, PBIO, along with code generation, can provide reliable, high performance, easy-to-migrate, heterogeneous support for distributed applications.

2 The PBIO Communication Library

In order to conserve I/O bandwidth and reduce storage and processing requirements, storing and transmitting data in binary form is often desirable. However, transmission of binary data between heterogeneous environments has been problematic. PBIO was developed as a portable self-describing binary data library, providing both stream and file support along with data portability.

The basic approach of the Portable Binary I/O library is straightforward. PBIO is a record-oriented communications medium. Writers of data must provide descriptions of the names, types, sizes and positions of the fields in the records they are writing. Readers must provide similar information for the records they wish to read. No translation is done on the writer's end, our motivation being to offload processing from data providers (e.g., servers) whenever possible. On the reader's end, the format of the incoming record is compared with the format expected by the program. Correspondence between fields in incoming and expected records is established by field name, with no weight placed on size or ordering in the record. If there are discrepancies in field size or placement, then PBIO's conversion routines perform the appropriate translations. Thus, the reader program may read the binary information produced by the writer program despite potential differences in: (1) byte ordering on the reading and writing architectures; (2) differences in sizes of data types (e.g. long and int); and (3) differences in structure layout by compilers.

Since full format information for the incoming record is available prior to reading it, the receiving application can make run-time decisions about the use and processing of incoming messages about whom it had no *a priori* knowledge. However, this additional flexibility comes with the price of potentially complex format conversions on the receiving end. Since the format of incoming records is principally defined by the native formats of the writers and PBIO has no *a priori* knowledge of the native formats used by the

program components with which it might communicate, the precise nature of this format conversion must be determined at run-time.

Since high performance applications can ill afford the increased communication costs associated with interpreted format conversion, PBIO uses dynamic code generation to reduce these costs. The customized data conversion routines generated must be able to access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Measurements[7] show that the one-time costs of DCG, and the performance gains by then being able to leverage compiled (and compiler-optimized) code, far outweigh the costs of continually interpreting data formats. The analysis in the following section shows that DCG, together with native-format data transmission and copy reduction, allows PBIO to provide its additional type-matching flexibility without negatively impacting performance. In fact, PBIO outperforms our benchmark communications package in all measured situations.

3 Evaluation

In order to thoroughly evaluate PBIO's performance and its utility in high-performance monitoring communication, we present a variety of measurements in different circumstances. Where possible, we compare PBIO's performance to the cost of similar operations in MPI or an XML-based system. In addition to basic data transfer costs, we also evaluate XML and PBIO for their performance in situations involving application evolution.

3.1 Analysis of costs in heterogeneous data exchange

Before analyzing the various packages in detail, it is useful to examine the costs in an exchange of binary data in a heterogeneous environment. As a baseline for this discussion, we use the MPICH[15] implementation of MPI, a popular messaging package in cluster computing environments. Figure 2 represents a breakdown of the costs in an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[1] We present a round-trip times both because they naturally show all the possible costs in the

[1] 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.

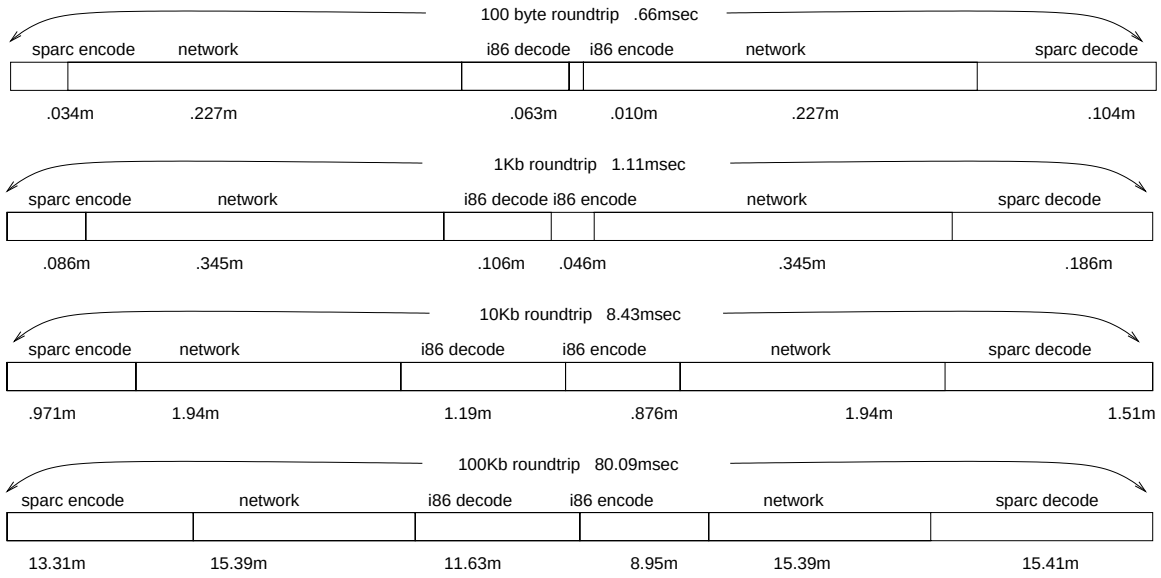


Figure 2: Cost breakdown for message exchange.

communication and because the efficiency of automatic steering of monitored computations depends directly upon round-trip latency. Architecture heterogeneity is also common in monitoring situations, where desktop x86-class machines may run display programs associated with big-endian workstation clusters.

The time components labeled "Encode" represent the time span between the time application invokes `MPI_send()` and the eventual call to write data on a socket. The "Decode" component is the time span between the recv() call returning and the point at which the data is in a form usable by the application. In generating these numbers network transmission times were measured with NetPerf[13] and send and receive times were measured by substituting dummy calls for socket send() and recv(). This delineation allows us to focus on the encode/decode costs involved in binary data exchange. That these costs are significant is clear from the figure, where they typically represent 66% of the total cost of the exchange.

Figure 2 shows the cost breakdown for messages of a selection of sizes, but in practice, message times depend upon many variables. Some of these variables, such as basic operating system characteristics that affect raw end-to-end TCP/IP performance, are beyond the control of the application or the communication middleware. Different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network, much of the time those differences are negligible, but where they

are not, they can have a significant impact upon the relative costs of a message exchange.

Another application characteristic which has a strong effect upon end-to-end message exchange time is the precise nature of the data to be sent in the message. It could be a contiguous block of atomic data elements (such as an array of floats), a stride-based element (such as a stripe of a homogeneous array), a structure containing a mix of data elements, or even a complex pointer-based structure. MPI, designed for scientific computing, has strong facilities for homogeneous arrays and strided elements. MPI's support for structures is less efficient than its support for contiguous arrays of atomic data elements, and it doesn't attempt to supported pointer-based structures at all. PBIO doesn't attempt to support strided array access, but otherwise supports all types with equal efficiency, including a non-recursive subset of pointer-based structures. The XML approach is more removed from actual binary data representations and can be used for both statically and dynamically sized elements with relative ease.

The message type of the 100Kb message in Figure 2 is a non-homogeneous structure taken from the messaging requirements of a real application, a mechanical engineering simulation of the effects of micro-structural properties on solid-body behavior. The smaller message types are representative subsets of that mixed-type message. In application-level monitoring, the precise nature of the data being transmitted may very widely from application to application, but the structured data types used here are not atypical.

The next sections will examine the relative costs of PBIO, MPI and XML in exchanging the same sets of messages.

3.2 Sending side cost

Figure 3 shows a comparison of sending-side data encoding times on the Sparc for an XML implementation[2], MPICH and PBIO. The figure shows dramatic differences in the amount of encoding necessary for the transmission of data (which is assumed to exist in binary format prior to transmission). In all cases, intermediate buffers have been pre-allocated and the encode overhead measured by replacing the network `send()` routine with a null operation. The XML costs represent the processing necessary to convert the data

[2] 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 [3].

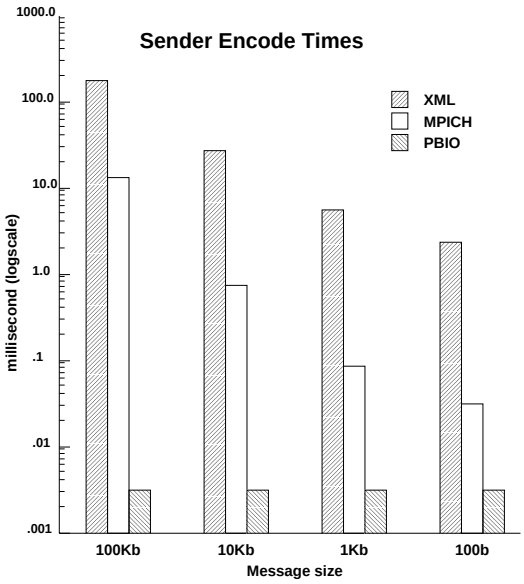


Figure 3: Send-side data encoding times.

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 MPI round-trip message exchange (as shown in Figure 2). As is mentioned in Section 2, PBIO transmits data in the native format of the sender. No copies or data conversions are necessary to prepare simple structure data for transmission. So, while MPICH's costs to prepare for transmission on the Sparc vary from 34μsec for the 100 byte record up to 13 msec for the 100Kb record, PBIO's cost is a flat 3 μsec. Of course, this efficiency is accomplished by moving most of the complexity to the receiver, where Section 3.4 tells a more complex story.

3.3 Data Transmission Costs

Table 1 shows the network costs for transmitting the encoded data with each package. In both PBIO

Original Data Size	Transmission time XML	MPICH	PBIO
100Kb	91ms	15ms	15.4ms
10Kb	9.1ms	1.9ms	1.9ms
1Kb	.85ms	0.35ms	0.35ms
100b	.265ms	0.23ms	0.23ms

Table 1: One-way network transmission costs for encoded data.

and MPI, the size of the encoded data is very close to the size of the original binary data, so their network overheads are very close to the minimum for the exchange of that quantity data. Data encoded in XML, however, is significantly larger in its encoded (string) form than it is in binary. While the amount of expansion depends to some extent upon the data and the size of the element labels, an expansion factor of 6-8 is not unusual. Thus XML-based schemes transmit significantly more data than schemes which rely on binary encoding. As Table 1 shows, this is of relatively little consequence for small messages where constant terms in the network cost equation tend to dominate. However, at medium and large message sizes the data expansion is more directly reflected in the network transmission time.

3.4 Receiving side cost

PBIO's 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. Essentially, the receiver must perform a conversion from the various incoming 'wire' formats to the receiver's 'native' format. PBIO matches fields by name, so 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. That marshaling can be a significant overhead is also well known[4, 24], and tools such as USC[18] attempt to optimize marshaling with compile-time solutions. Unfortunately, the dynamic form of the marshaling problem in PBIO, 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 Figure 2 shows, the overhead is high (66%) for some heterogeneous exchanges.

Generically, receiver-side overhead in communication middleware has several components which can be traded off against each other to some extent. Those basic costs are:

- 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 byteswapping 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 which 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, incurring the data movement costs described in the previous paragraph.

Packages that marshal data themselves typically use an alternative approach to control, where the marshalling 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 for PBIO.

XML necessarily takes a different approach to receiver-side decoding. Because the 'wire' format is a

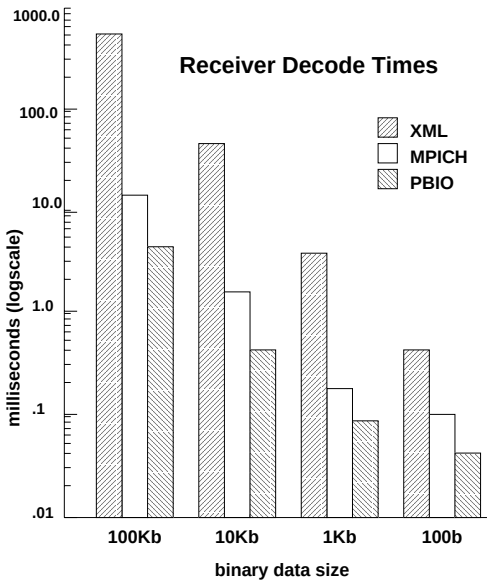


Figure 4: Receiver side costs for XML, MPI and PBIO.

continuous string, XML is parsed at the receiving end. The Expat XML parser[3] 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 also extremely fast, performing its principal function with pointer manipulations and in-place string modification rather than copying strings. However, XML still pays a relatively heavily penalty for requiring string-to-binary conversion on the receiving side. (We assume that for most monitoring functions, data is being sent somewhere for processing and that processing requires the monitoring 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.)

Figure 4 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by XML, MPICH (via the `MPI_Unpack()`) call and PBIO. XML receiver conversions are clearly expensive, typically between one and two orders of decimal magnitude more costly than PBIO's converter for this heterogeneous exchange. (On an exchange between homogeneous architectures, PBIO and MPI would have substantially lower costs, while XML's costs would remain unchanged.) PBIO's converter is relatively heavily

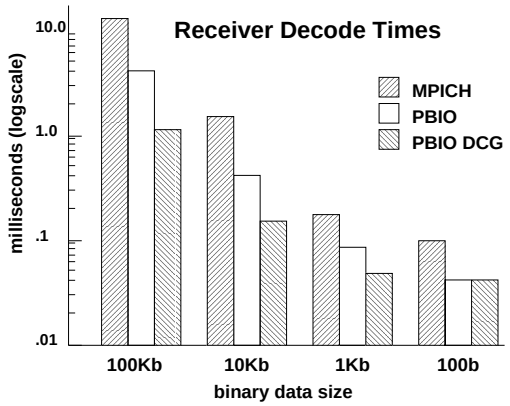


Figure 5: Receiver side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

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 PBIO does). However, PBIO'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, we believed that a significant fraction of this overhead was due to the fact that the conversion is essentially being performed by an interpreter.

Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time, making a remedy to the problem of interpretation overhead difficult. However, our solution to the problem was to employ dynamic code generation to create a customized conversion subroutine for every incoming record type[3]. These routines are generated by the receiver on the fly, as soon as the wire format is known, through a procedure that structurally resembles the interpreted conversion itself. However, instead of performing the conversion this procedure directly generates machine code for performing the conversion.

The execution times for these dynamically generated conversion routines are shown in Figure 5. (We have chosen to leave the XML conversion times off of this figure to keep the scale to a manageable size.) The dynamically generated conversion routine operates significantly faster than the interpreted version. This

[3]More details on the nature of PBIO's dynamic code generation can be found in [6].

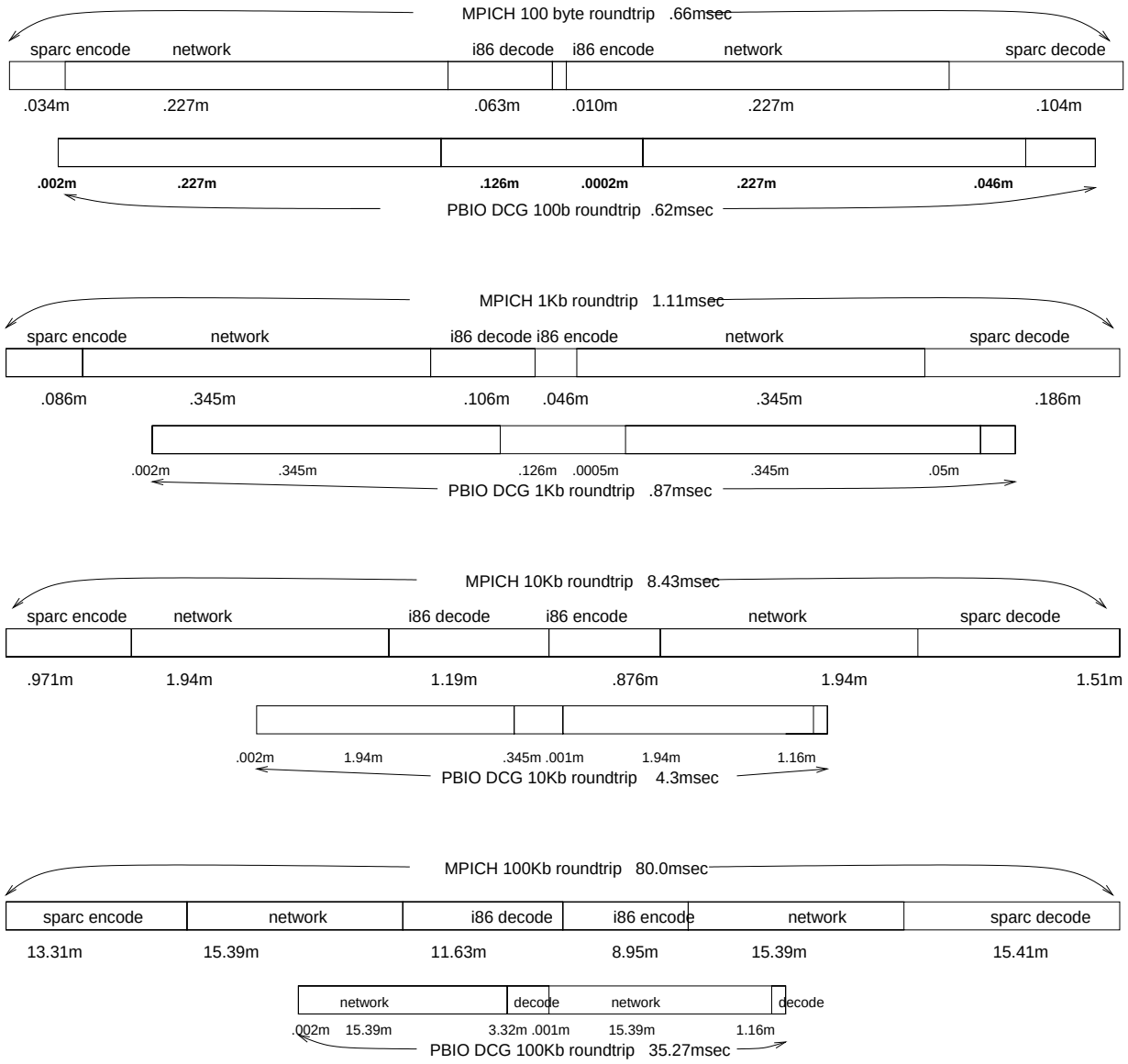


Figure 6: Cost comparison for PBIO and MPICH message exchange.

improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation, and is the key to PBIO's ability to efficiently perform many of its functions.

The cost savings achieved by PBIO through the techniques described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 6 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes. The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time

Original Data Size	Round-trip time		
	XML	MPICH	PBIO
100Kb	1200ms	80ms	35ms
10Kb	149ms	8.4ms	4.3ms
1Kb	24ms	1.1ms	0.87ms
100b	9ms	.66ms	0.62ms

Table 2: Cost comparison for round-trip message exchange for XML, MPICH and PBIO.

required by MPICH. The performance gains are due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to customize a conversion routine on the receiving side (currently not done on the x86 side).

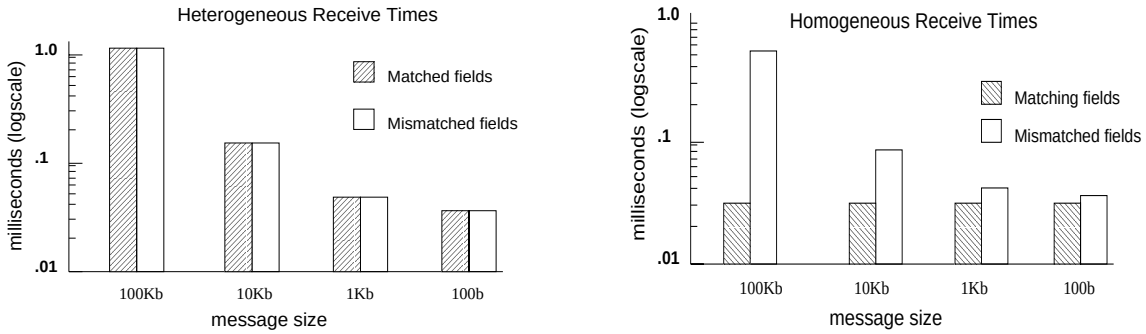
Once again, Figure 6 does not include XML times to keep the figure to a reasonable scale. Instead, Table 2 summarizes the relative costs of the round-trip exchange with XML, MPICH and PBIO.

3.5 Performance in application evolution

The principal difference between PBIO and most other messaging middleware is that PBIO messages 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, PBIO allows *reflection* and *type extension*. Both of these are valuable features commonly associated with object systems.

PBIO supports reflection by allowing message formats to be inspected before the message is received. It's support of type extension derives from doing field matching between incoming and expected records by name. Because of this, new fields can be added to messages 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



(a) heterogeneous case.

(b) homogeneous case.

Figure 7: Receiver-side decoding costs with and without an unexpected field

to systems such as MPI which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon PBIO 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 7a and 7b present receive-side processing costs 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 PBIO's dynamic code generation facilities to create conversion routines. It's clear from Figure 7a 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 7b 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 PBIO. 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.

As noted earlier in Section 3.4, XML is extremely robust to changes in the format of the incoming record. Essentially, XML transparently handles precisely the same types of change in the incoming record as can PBIO. That is, new fields can be added or existing fields reordered without worry that the changes will invalidate existing receivers. Unlike PBIO, XML's behavior does not change substantially when such mismatches are present. Instead, XML's receiver-side decoding costs remain essentially the same as presented in Figure 4. However, those costs are several orders of decimal magnitude higher than PBIO's costs.

For PBIO, the results shown in Figure 7 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 PBIO 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

As has been demonstrated in this paper, binary data transfer support is key to efficient event-based monitoring in a high performance computing environment. This particularly evident in a distributed laboratory where compute resources are heterogeneous, data streams large, number of scientists and instruments is also large, and distribution of scientists and instruments is broad. Performance results show that PBIO is a valuable addition to the mechanisms available for handling binary data interchange, particularly across heterogeneous resources. PBIO performs efficient data translations, and supports simple, transparent system evolution of distributed applications, both on a software and a hardware basis.

The measurements in this paper have shown that PBIO's flexibility does not impact its performance. In fact, PBIO's performance is better than that of a popular MPI implementation in every test case, and significantly better in heterogeneous exchanges. Performance gains of up to 60% are largely due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*

- using dynamic code generation to perform data conversion on the receiving side.

With the explosive growth of both the Internet and available bandwidth, scientists are beginning to demand computing more on their terms. That is, they are interested in abstractions that lets them name and manipulate objects in science terms rather than as data structures and primitive data types familiar to computer scientists. Scientists are also interested in standards that allow them to agree upon the scientific entities they manipulate. But in the excitement surrounding XML today, which provides both, one must point out that transmitting XML in ASCII and parsing it on the receiving end carries with it a heavy penalty: one to two orders of magnitude (decimal) more costly than PBIO. To address this problem, our group is currently exploring combining the abstraction and standardization features of XML with fast heterogeneous data exchange of PBIO.

In summary, PBIO is a novel messaging middleware that combines significant flexibility improvements with an efficient implementation. This paper has demonstrated the need for such a fast heterogeneous binary data interchange in large-scale event-based monitoring applications (*e.g.* distributed laboratory) involving scientists that are increasingly demanding science-centric approaches.

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] A. Bakic, M.W. Mutka, and D.T. Rover. Birsk: A portable and flexible distributed instrumentation system. In *Proceedings of International Parallel Processing Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)*. IEEE, 1999.

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

[4] 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.

[5] 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)*.

[6] Greg Eisenhauer and Lynn K. Daley. Fast heterogenous binary data interchange. In *Proceedings of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000. http://www.cc.gatech.edu/systems/papers/Eisenhauer00FHB.pdf.

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

[8] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[9] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[10] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[12] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[13] Hewlet-Packard. The netperf network performance benchmark. http://www.netperf.org.

[14] W. Hibbard. VisAD: connecting people to computations and people to people. *Computer Graphics*, 32(3):10–12, 1998.

[15] Argonne National Laboratory. Mpich-a portable implementation of mpi. http://www-unix.mcs.anl.gov/mpi/mpich.

[16] 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.

[17] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. Paradyn parallel performance measurement tools. *IEEE Computer*, 28, November 1995.

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

[19] S.G. Parker and C.R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proc. Supercomputing 95*, pages 1–1, 1995.

[20] Beth Plale, Volker Elling, Greg Eisenhauer, Karsten Schwan, Davis King, and Vernard Martin. Realizing distributed computational laboratories. *International Journal of Parallel and Distributed Systems and Networks*, 2(3), 1999.

[21] Beth Plale and Karsten Schwan. Run-time detection in parallel and distributed systems: Application to safety-critical systems. In *Proceedings of Int'l Conference on Distributed Computing Systems (ICDCS'99)*, pages 163–170, August 1999.

[22] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment.* Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, November 1992.

[23] Marcel-Catalin Rosu, Karsten Schwan, and Richard Fujimoto. Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, 1, January 1998.

[24] M. Schroeder and M. Burrows. Performance or firefly rpc. In *Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5*, pages 83–90. ACM, SIGOPS, Dec. 1989.