Native Data Representation: An Efficient Wire Format for High Performance Distributed Computing

Greg Eisenhauer, Fabián E. Bustamante, and Karsten Schwan

College of Computing

Georgia Institute of Technology

Atlanta, Georgia 30332, USA

{eisen, fabianb, schwan}@cc.gatech.edu

Abstract

New trends in high-performance software development such as tool- and component-based approaches have increased the need for flexible and high-performance communication systems. High-performance computing applications are being integrated with a variety of software tools to allow on-line remote data visualization, enable real-time interaction with remote sensors and instruments, and provide novel environments for human collaboration. There has also been a growing interest among high-performance researchers in component-based approaches, in an attempt to facilitate software evolution and promote software reuse. When trying to reap the well-known benefits of these approaches, the question of what communications infrastructure should be used to link the various components arises.

In this context, *flexibility* and *high-performance* seem to be incompatible goals. Traditional HPC-style communication libraries, such as MPI, offer good performance, but are not intended for loosely-coupled systems. Object- and metadata-based approaches like XML offer the needed plug-and-play flexibility, but with significantly lower performance.

We observe that the flexibility and baseline performance of data exchange systems are strongly determined by their *wire formats*, or by how they represent data for transmission in the heterogeneous environments. After examining the performance implications of using a number of different wire formats, we propose an alternative approach for flexible high-performance data exchange, *Native Data Representation*, and evaluate its current implementation in the *Portable Binary I/O* library.

Index Terms: High-performance, distributed computing, communication, wire format.

1 Introduction

New trends in high-performance software development such as tool- and component-based approaches have increased the need for flexible and high-performance communication systems. High-performance computing applications are being integrated with a variety of software tools to allow on-line remote data visualization [1], enable real-time interaction with remote sensors and instruments, and provide novel environments for human collaboration [2]. There is also a growing interest among high-performance researchers in component-based approaches, in an attempt to facilitate software evolution and promote software reuse [3, 4, 5]. When trying to reap the well-known benefits of these approaches, the question of what communications infrastructure should be used to link the various components arises.

In this context, *flexibility* and *high-performance* seem to be incompatible goals. Traditional HPC-style communications systems like MPI offer the required high performance, but rely on the assumption that communicating parties have *a priori* agreements on the basic contents of the messages being exchanged. This assumption severely restricts flexibility and makes application maintenance and evolution increasingly onerous. The need for flexibility has led designers to adopt techniques like the use of serialized objects as messages (Java's RMI [6]) or to rely on meta-data representations such as XML [7]. These alternatives, however, have high marshalling and communications costs in comparison to more traditional approaches [8, 9].

We observe that the flexibility and baseline performance of a data exchange system are strongly determined by its *wire format*, or how it represents data for transmission in a heterogeneous environment. Upon examining the flexibility and performance implications of using a number of different wire formats, we propose an alternative approach that we call *Native Data Representation*.

The idea behind Native Data Representation (NDR) is straightforward. We avoid the use of a common wire format by adopting a "receiver-makes-right" approach, where the sender transmits the data in its own native data format and it is up to the receiver to do any necessary conversion. To reduce the possible costs of conversion, any translation on the receiver's side is performed by custom routines created through dynamic code generation (DCG). By eliminating the common wire format, the up and down translations required by approaches like XDR are potentially avoided. Furthermore, when sender and receiver use the same native data representation, such as in exchanges between homogeneous architectures, this approach allows received data to be used directly from the message buffer eliminating high copy overheads [10, 11]. When sender's and receiver's formats differ, NDR's DCG-based conversions have efficiency similar to that of systems that rely on *a priori* agreements to make use of compile- or link-time stub generation. However, because NDR's conversion routines are dynamically generated at data-exchange initialization, our approach offers considerably greater flexibility. The meta-data required to implement this approach and the runtime flexibility afforded by DCG together allow us to offer plug-and-play communication similar to that provided by XML or object marshalling without compromising performance.

NDR is not a new communication paradigm on par with message passing or RPC, but a low-level approach to communication which can be used to implement these (and other) paradigms more efficiently.

NDR has been implemented in a new version of the *Portable Binary I/O* library [12]. Experimentation with a variety of realistic applications shows that the NDR-based approach obtains the required flexibility at no significant cost to performance. On the contrary, the results presented in Section 4 demonstrate improvements of up to 3 orders of magnitude in the sender encoding time, 1 order of magnitude on the receiver side, and a 45% reduction in data roundtrip time, when compared to the exchange of data with known high performance communication systems such as MPI.

The remainder of this paper is organized as follows: in Section 2 we review related approaches to communication and comment on their performance and flexibility. Section 3 presents the Native Data Representation approach and its implementation in the Portable Binary I/O (PBIO) communication library. We compare the performance and flexibility of PBIO with that of alternative communication systems in Section 4. After examining the different costs involved in the communication of binary data on heterogeneous platforms, we evaluate the relative costs of MPI, XML and CORBA's IIOP communications in exchanging the same sets of messages, and compare them to those of PBIO. Next, the performance effects of dynamic type discovery and extension are compared for PBIO *vs.* XML-based systems. Finally, we show microbenchmark results that demonstrate the overheads incurred by the creation and maintenance of the meta-information needed by PBIO, as well as the additional costs incurred for DCG at the time a new data exchange is initiated. We present our conclusion and discuss some directions for future work in Section 5.

2 Related Work

In this section we review work on communication approaches, with a focus on their support for high-performance and flexibility, and relate them to NDR.

Not surprisingly, efficiency has been the single most important goal of high-performance communication packages such a PVM [13], Nexus [14] and MPI [15]. Most of these packages support message exchanges in which the communicating applications "pack" and "unpack" messages, building and decoding them field by field [13, 14]. By manually building their messages, applications have full control over message contents while ensuring optimized, compiled pack and unpack operations. However, relegating these tasks to the communicating applications means that the communicating components must agree on the format of messages. In addition, the semantics of application-side pack/unpack operations generally imply a costly data copy to or from message buffers [16, 10].

Other packages, such as MPI, support the creation of user-defined data types for messages and fields and provide some marshalling and unmarshalling support for them. Although this provides some level of flexibility, MPI does not have any mechanisms for run-time discovery of data types of unknown messages, and any variation in message content invalidates communication. In addition, while internal marshalling could avoid the aforementioned cost of data copies and offer more flexible semantics in matching senders' and receivers' fields, most implementations fail to capitalize on these opportunities. MPI's type-matching rules, for example, require strict agreement on the content of messages, and most implementations marshal user-defined datatypes via mechanisms that amount to interpreted versions of field-by-field packing.

In summary, the operational norm for high-performance communication systems is for all parties to a communication to have an *a priori* agreement on the format of messages exchanged. The consequent need to simultaneously update all system components in order to change message formats is a significant impediment to system integration, deployment and evolution.

New trends in high-performance software development, such as tool- and component-based approaches [3, 4, 5], have increased the need for more flexible communication systems. This need has promoted the use of object-oriented systems and meta-data representations. Although object technology provides some amount of plug-and-play interoperability through subclassing and reflection, this typically comes at the price of communication efficiency. For example, CORBA-based object systems use IIOP [17] as a wire format. IIOP attempts to reduce marshalling overhead by adopting a "receiver-makes-right" approach with respect to byte order (the actual byte order used in a message is specified by a header field). This additional flexibility in the wire format allows CORBA to avoid unnecessary byte-swapping in message exchanges between homogeneous systems, but it does not eliminate the need for data copying by both sender and receiver. At issue here is the contiguity of atomic data elements in structured data representations. In IIOP, XDR and other wire formats, atomic data elements are contiguous, without intervening space or padding between elements. In contrast, the native, in-memory representations of those structures in the actual applications must contain appropriate padding to ensure that the alignment constraints of the architecture are met. On the sending side, the contiguity of the wire format means that data must be copied into a contiguous buffer for transmission. On the receiving side, the contiguity requirement means that data cannot be referenced directly out of the receive buffer, but must be copied to a different location with appropriate alignment for each element. Therefore, the way in which marshalling is abstracted in these systems prevents copies from being eliminated even when analysis might show them unnecessary.

While all of the communication systems above rely on some form of fixed wire format for communication, most XML-based systems take a different approach to communication flexibility. Rather than transmitting data in binary form, they rely on ASCII-text as a wire format with each record represented in textual form with header and trailer information identifying each field. This allows applications to communicate without previous knowledge of each other. However, XML encoding and decoding costs are substantially higher than those of other formats due to the conversion of data from binary to ASCII and vice versa. In addition, XML-based approaches have substantially higher network transmission costs because the ASCII-encoded record is often substantially larger than the binary original. The amount of expansion caused by XML depends upon tag sizes, data contents, the use of arrays and attributes and a variety of other factors. We have found that an expansion factor of 6-8 above the size of binary data is not unusual.

In this paper we propose and evaluate an alternative wire format approach, *Native Data Representation*, with combined goals of flexibility and high performance. NDR increases application flexibility by allowing receivers to make run-time decisions on the use and processing of incoming records without *a priori* knowledge of their formats. NDR achieves its goal of high performance by reducing copy and conversion overheads at senders and avoiding the potential cost of potentially complex format conversions on the receiving end through the use of dynamic code generation.

3 Native Data Representation

This section describes the details of Native Data Representation and its implementation in the PBIO communication library. A comparative evaluation of NDR and other approaches appears in Section 4.

Instead of imposing a single network standard, NDR allows the sender to use its own *native* format. Data is placed "on the wire" in the sender's internal format and it is up to the receiver

```
typedef struct small_record {
  int ivalue;
  double dvalue;
  int iarray[5];
} small_record, *small_record_ptr;


IOField small_record_fld[] =
{
  {"ivalue", "integer", sizeof(int), IOOffset(small_record_ptr,ivalue)},
  {"dvalue", "float", sizeof(double), IOOffset(small_record_ptr,dvalue)},
  {"iarray", "integer[5]", sizeof(int), IOOffset(small_record_ptr,iarray)},
  {NULL, NULL, 0, 0}
};
```

Figure 1: An example of message format declaration. `IOOffset()` is a simple macro that calculates the offset of a field from the beginning of the record.

to do any necessary conversion. More concretely, PBIO puts data on the wire in exactly the sender's memory representation, including gaps that reflect the senders alignment requirements. By avoiding the use of a lowest-common-denominator format, it maximizes the possibility that program-internal data can be transmitted and used directly without costly copy operations. For this to be possible, however, senders and receivers must provide format descriptions of the records they wish to exchange, including the names, types, sizes and positions of the fields in the records. Figure 1 shows an example of this format declaration.

On the receiver's end, the format of the incoming record is compared with the format expected by the process, with correspondence between incoming and expected records' fields established by field name (with no weight placed on the field's size or its position in the record). If there are discrepancies in size or placement of a field, then appropriate conversion routines perform the necessary translations. In this way, a receiver program can read the binary information produced by a sender despite potential differences in: byte ordering, sizes of data types (*e.g.* `long` and `int`), and record layout by compilers.

PBIO's implementation of NDR separates the detailed format descriptions from the actual messages exchanged. Format descriptions are registered with a format service and messages are prefixed with a small *format token* that identifies them. From this format service, receivers can request (and cache) previously unseen format descriptions. Thus, format operations are one-time events associated with the registration of a new format description or the first occurrence of a message of a particular format.

In the following subsections we describe the marshalling and unmarshalling of PBIO's messages and provide some details on the use of record format descriptions.

3.1 Marshalling and Unmarshalling

Minimizing the costs of conversions to and from wire formats is a well known problem in network communication [18, 19]. Traditional marshalling/unmarshalling can be a significant overhead [20, 21], and tools like the Universal Stub Compiler (USC) [19] attempt to minimize this cost with compile-time solutions. Although optimization considerations similar to those addressed by USC apply in our case, the dynamic nature of the marshalling 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.

3.1.1 Marshalling

Because PBIO's approach to marshalling involves sending data largely as it appears in memory on the sender's side, marshalling is computationally inexpensive. Messages are prefixed with a small (32-128 bits[1]) *format token* that identifies the format of the message. If the message format contains variable length elements (strings or dynamically sized arrays), a 32-bit length element is also added at the head of the message. Message components that do not have string or dynamic subfields (such as the entire message of Figure 1) are not subject to any processing during marshalling. They are already in the "right" wire format. However, components with those elements contain pointers by definition. The PBIO marshalling process copies those components to temporary memory (to preserve the original) and converts the pointers into offsets inside the message. The end-result of PBIO's marshalling is a vector of buffers which together constitute an encoded message. Those buffers can be written on the wire directly by PBIO or transmitted via another mechanism to their destination.

3.1.2 Unmarshalling

The NDR approach can reduce or even eliminate the sender-side processing required for data transmission. In addition, NDR increases application flexibility by allowing receivers to make run-time decisions about the use and processing of incoming messages without any previous knowledge of their formats. These benefits, however, come at the cost of potentially complex format conversions on the receiving end. Because the format of incoming records is principally defined by the native formats of the writers, and because PBIO has no *a priori* knowledge of the native formats of the communicating parties, the precise nature of this format conversion

[1]The exact size depends upon the particular way PBIO is used and the configuration of format servers. This topic is covered in more detail in Section 3.2.2.

must be determined at run-time. While in homogeneous exchanges most of these costs can be avoided, unmarshalling in heterogeneous exchanges can be expensive as it may require byte-order conversions (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).

In order to reduce the cost of heterogeneous exchanges, PBIO employs to dynamic code generation for unmarshalling. For each incoming wire format, PBIO creates a specialized native subroutine that converts incoming records into the receiver's format. These native conversion subroutines are cached, based on the format token of the incoming record, and reused in subsequent conversions allowing the amortization of the costs associated with code generation. The run-time generation of conversion subroutines is essentially a more dynamic approach to the problems addressed by tools like USC[19]. A more detail discussion of the unmarshalling costs and the contribution of dynamic code generation is provided in Section 4.3.

3.2 Dealing with Formats

PBIO's implementation of NDR separates the detailed format descriptions from the actual messages exchanged. Format descriptions are registered with a format service and messages are prefixed with a small *format token* that identifies them. Record format descriptions in PBIO include the names, types, sizes and positions of the fields in the messages exchanged. Figure 1 shows a C language declaration that builds a format description for use in PBIO. Because the size and byte offset of each field may change depending upon the machine architecture and compiler in use, those values are captured using the C `sizeof()` built-in and the PBIO `IOOffset()` macro.

The format description in Figure 1 may look somewhat obscure, but it could easily be generated from the C typedef. In fact, both the typedef and the PBIO field list declaration can be generated from a higher level specification, such as a CORBA IDL struct declaration or even an XML schema (see Figure 2).

Different forms of specification are appropriate for different applications. The form of Fig-

```
interface small {
   struct small_record {
      long value;
      double dvalue;
      long< 5 > iarray;
   };
}
```

(a) CORBA IDL specification

```
<schema>
   <element name="ivalue" type="integer"/>
   <element name="dvalue" type="double"/>
   <element name="iarray" type="integer"
            minOccurs=5 maxOccurs=5/>
</schema>
```

(b) XML Schema specification

Figure 2: Alternative message structure definitions.

ure 1 is easiest for integrating PBIO-based messaging into an existing C application, but forms such as those in Figure 2 may be more convenient for new applications. Regardless of the form of the specification, PBIO's capabilities are defined by the types of messages that it can represent and marshal, namely C-style structures whose fields may be atomic data types, substructures of those types, null-terminated strings, and statically- or dynamically-sized arrays of these elements. In the case of dynamically-sized arrays, the array is represented by a pointer to a variable-sized memory block whose length is given by an integer-typed element in the record.[2]

Dynamic Formats. Because PBIO formats roughly correspond to a description of a C-style `struct`, the formats used by individual applications tend to be relatively static (as are those structures), and the field lists of locally-used records are known at compile time. However, unlike many marshalling and communication mechanisms, PBIO does not depend in any way upon compile-time stub generation or any other compile-time techniques for its efficiency or normal operation. Field lists (of the form of Figure 1) supplied at run-time are all that PBIO requires to build formats for marshalling and unmarshalling. These highly-dynamic capabilities of PBIO are useful in creating plug-and-play components that operate upon data that may not be specified until run-time. These more highly-dynamic features of PBIO are also exploited

[2]PBIO does not attempt to represent recursively-defined pointer-based data structures such as trees or linked lists.

by an XML interface that "dehydrates" XML into a PBIO message for transmission and "re-hydrates" it at the receiver based on a run-time specified schema described in detail in [22].

3.2.1 Format Description Representation and Size

One factor affecting the cost of dealing with formats is the actual size of the format information to be exchanged. Unlike the records they describe, PBIO format information is represented on the wire by a well-defined structure that includes some general format information of fixed size ($\approx$ 16 bytes), the format name, and information for each of the fields in the format's field list. The information for each field consists of a fixed size portion (currently 12 bytes) and a variable size portion (the field name and base type). A general expression[3] for the approximate wire-size of format information is:

$$\texttt{size} \approx 16 + strlen(format_{name}) + \sum_{f \epsilon Fields} (12 + strlen(f_{name}) + strlen(f_{type}))$$

The first two bytes of the format information give its overall length and are always in network byte order. One of the next bytes specifies the byte order of the remaining information in the format. PBIO format operations that involve the transfer of format descriptions always use this wire format for their exchanges. It is important to note that such operations are associated only with one-time events, such as a new format description registration or the first occurrence of a message of a particular format.

3.2.2 Format Servers and Format Caches

The *format service* in PBIO is provided by servers that issue *format tokens* when formats are registered with them. For identical formats (same fields, field layout, format name, and machine representation characteristics), a format server issues identical format tokens. These

[3]The constants in this expression are an artifact of the exact details of format representation, largely irrelevant to this discussion. The expression itself is included only to provide a rough idea of the characteristics of format communication. A full analysis of the one-time costs associated with various format service schemes is beyond the bounds of this paper.

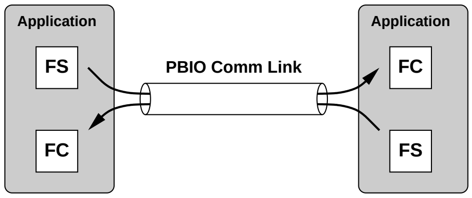

Figure 3: Single PBIO connection showing per-connection format servers (FS) and caches (FC).

tokens can be presented to format servers in order to retrieve complete format information. *Format caches* are repositories of format information, indexed by format tokens, that exist on both the encoding and decoding clients to optimize communication.

The details of format communication in PBIO depend to some extent upon the circumstances of its use. Two principal modes are:

- **Connected PBIO:** where PBIO performs marshalling/unmarshalling and directly controls transmission on the network, and
- **Non-connected PBIO:** where PBIO performs marshalling/unmarshalling, but is not in direct control of network transmission.

The first case is the simplest one because PBIO can ensure that the format information for a given record is sent across the wire before the first record of that format. In this case, format information is issued (by the sender) and cached (by the receiver) on a per-connection basis. Because formats are always interpreted in the context of a particular connection, format tokens in this mode are simple 32-bit integers where the token with value i is the i-th format transmitted on that connection. Since the sender always transmits format information first, essentially pre-loading the receiver's format cache, there are no requests for it. This situation is depicted in Figure 3.

The case of non-connected PBIO is more interesting. Here a message (along with its format token) is sent by non-PBIO means to a third party. Because PBIO does not control transmission, it cannot pre-load the format cache of the receiver. Instead, the third-party receiver

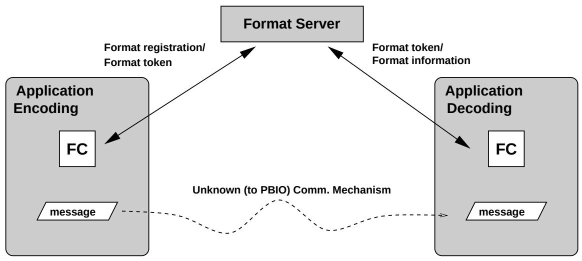

Figure 4: Simple format service arrangement in non-connected PBIO. A single PBIO format server is shared by all communicating applications. Each application has its own format cache (FC) to minimize communication with the server.

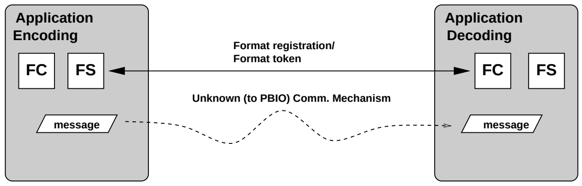

Figure 5: The "self-service" format arrangement in non-connected PBIO. Each application acts as a format server (FS) for its own formats and maintains a cache (FC) for formats registered elsewhere.

must be able to use the format token to retrieve the full format information. This is essentially a naming problem, and there are a number of possible implementation options. In the simplest one, depicted in Figure 4, a format server, located at a well-known address, serves all communicating parties.

The format service can be supported by a number of alternative client/server configurations. It can be provided by a particular server or, at the other end of the spectrum, each communicating client can act as the format server for its own formats, as shown in Figure 5.

The self-server arrangement is similar to the connected PBIO arrangement of Figure 3, except that there is one server and one cache per process instead of one per connection. Because the communication mechanism is unknown to PBIO and because format tokens are only meaningful when presented to the issuing server, the format token must contain enough information for the client to identify the correct server. Communication with the issuing format server is generally *not* done via the channels which propagate the message, though PBIO can be made to use one-to-one third-party communication links for format communication through special callbacks.

These two schemes, and variants between the extremes, have different performance characteristics with respect to communication startup costs. For example, the single format server approach maximizes the benefits of caching because identical formats always have identical format tokens, to the benefit of long-running clients. However, format registration for a new client always requires a communication with the central format server. In the scheme where each client is its own format server, format registration requires no network communication and is therefore quite cheap. However, caches will be less well utilized because two clients of the same machine architecture and transmitting the same records will have non-identical format tokens. Because a full analysis of the tradeoffs in this performance space is outside the bounds of this paper, we concentrate largely on evaluating PBIO performance in steady-state message exchange.

4 Evaluation

This section compares the performance and flexibility of PBIO's implementation of NDR with that of systems like MPI, CORBA, and XML-based ones. None of systems compared share PBIO's stated goals of supporting both flexible and efficient communication. MPI is chosen to represent traditional HPC communication middleware that depends upon *a priori* knowledge and sacrifices flexibility for efficiency. XML-based mechanisms are examined because they emphasize plug-and-play flexibility in communication without considering performance. CORBA is chosen as a relatively efficient representative of the object-based systems becoming popular in distributed high-performance computing. Some object-based notions of communication, such as exchanging marshalled objects, can potentially offer communication flexibility as good or better than PBIO's. However, current implementations of object marshalling are too inefficient

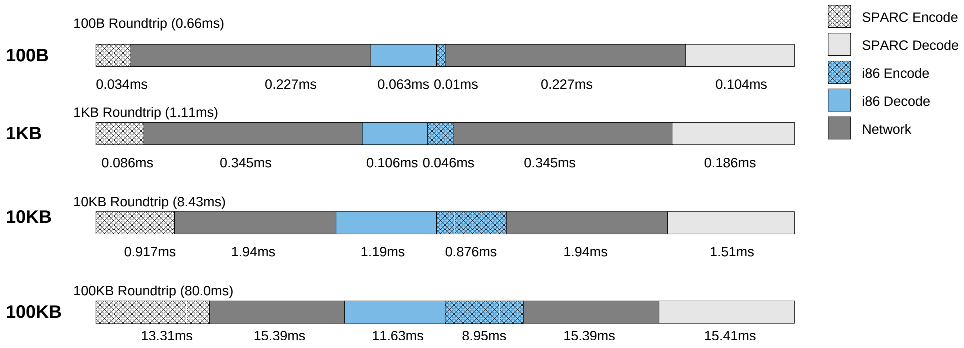

Figure 6: Cost breakdown for heterogeneous sparc/x86 roundtrip message exchange.

for serious consideration in high performance communication. Because of this, our measurements of CORBA in this context are only for one-way invocations where the data is carried as a single CORBA `struct` parameter. In this style of communication, CORBA offers little flexibility (no reflection or subclassing are possible).

4.1 Analysis of Costs in Heterogeneous Data Exchange

Before analyzing the various packages in detail, it is useful to examine the costs of a binary data exchange in heterogeneous environments. As a baseline for this discussion, we use the MPICH [23] implementation of MPI, as it provides a challenging comparison. Complete details of why this is the case will be explained in the following sections, but in summary:

- The heterogeneous exchange maximizes PBIO's relative costs. (In a homogeneous exchange, PBIO imposes virtually no cost above network transport times.)
- The costs of the other approaches do not vary significantly between heterogeneous and homogeneous exchanges.
- Though MPI lacks any of PBIO's flexibility in matching incoming data with the receiver's expectations, MPICH is representative of the best performing approaches discussed here.

```
typedef struct KSdata1_Record {
  int Cnstatv;
  double Cstatev[12];
  int Cnprops;
  double Cprops[110];
  int Cndi[4];
  int Cnshr;
  int Cnpt;
  double Cdtime;
  double Ctime[2];
  int Cntens;
  double Cdfgrd0[3][373];
  double Cdfgrd1[3][3];
  double Cstress[106];
  double Cddsdde[106][106];
} KSdata1;
```

Figure 7: Format of the 100KB message used in the evaluation. This format was extracted from a mechanical engineering simulation of the effects of micro-structural changes on solid-body behavior. The smaller messages are subsets of this structure.

Figure 6 represents a breakdown of the costs of an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[4]

The format of the 100KB messages exchanged is described in Figure 7 as a C structure declaration. The data structure is part of a mechanical engineering simulation of the effects of micro-structural changes on solid-body behavior. The smaller messages (10K, 1KB and 100B) are subsets of the original data structure.

[4]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.

The time components labeled *Encode* represent the time span between the point at which the application invokes `MPI_send()` and its 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. 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 6 shows the cost breakdown for various sized messages (using examples drawn from a mechanical engineering application), but in practice, message times depend upon many variables. Of these measurements, the network cost is largely determined by basic operating system characteristics that affect raw end-to-end TCP/IP performance and (aside from the message size implications of various encoding mechanisms) is beyond the control of the application or the communication middleware. Of course, different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network and therefore the network time. Those differences are generally negligible, but where they are not (such as in the case of XML), they can have a significant impact upon the relative costs of a message exchange.

The next subsections will examine the relative costs of PBIO, MPI, CORBA[5], and an XML-based system in exchanging the same sets of messages. We first compare the sending- and receiving-side communication costs for all the alternatives evaluated. Then we discuss our use of dynamic-code-generation to reduce unmarshalling costs and evaluate the resulting performance improvements. We conclude the section with an analysis of the performance effects of flexibility in PBIO.

4.2 Sending Side Cost

We first compare the sending-side data encoding times on the Sun Ultra-30 Sparc for an XML-based implementation[6], MPICH, CORBA, and PBIO. Figure 8 shows the different encoding

[5]The ORBacus for this evaluation.

[6]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 [24]. We create XML manually with sprintf() for data-to-string

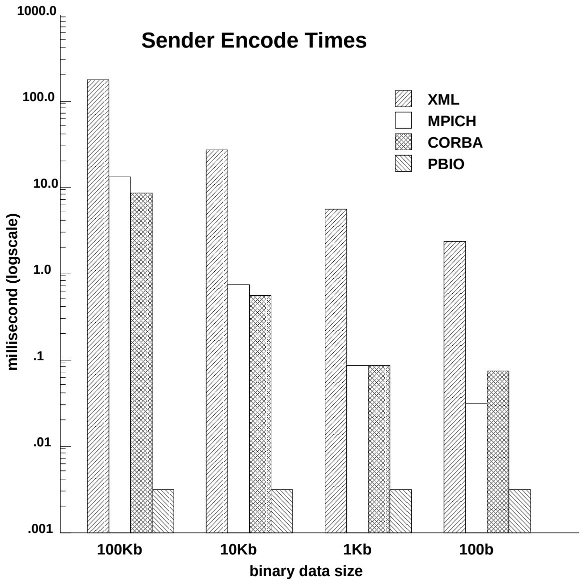

Figure 8: Send-side encoding times.

times in milliseconds. An examination of this plot yields two conclusions:

- **XML wire formats are inappropriate.** 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). The XML 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. The result is an encoding time that is at least an order of magnitude higher than other systems. 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 6). Further, the message represented in the ASCII-based XML format is significantly larger than in the

conversions and strcpy() for concatenation. (We track the end of the string and use strcpy() rather than use strcat() because the latter would have to re-locate the end of the string at each call.) Faster XML creation mechanisms might be possible, but our approach is reasonable and a faster creation mechanism would not change the overall results of our analysis for XML. Its relative size (and the resulting large transmission costs) compared to binary encoding mechanisms largely precludes it from use when performance is a consideration.

binary-based representations. This gives significantly larger network transmission times for XML messages, increasing latency and decreasing possible message rates substantially.

- **The NDR-approach significantly improves sending-side costs.** As is mentioned in Section 3, we transmit data in the native format of the sender. As a result, 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 and CORBA costs are comparable, PBIO's costs are a flat 3 μsec.

While these cost savings are significant, they are not unexpected. A basic consequence of adopting NDR is the reduction of sender processing to the absolute minimum, requiring that neither byte order or alignment be modified for transmission and therefore avoiding the need for copying data. On the receiving end, however, adopting this approach translates on relatively more complicated processing. The next section will examine the costs associated with that processing and their impact on the communication.

4.3 Receiving Side Costs

Broadly speaking, receiver-side overheads in communication middleware have 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.[7]

[7]We mainly discuss differences in byte orders because those are the principal representation differences found in modern machines. Although historically machines have used other integer and floating point represen-

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 communication 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 that 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 applications to marshal and unmarshal their own data have the advantage that this process occurs in specially-written compiler-optimized code, minimizing control costs. Systems such as CORBA, where the marshalling code can conceivably be pre-generated and compiled based upon static stubs, have a similar advantage. 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. Additionally, many CORBA IIOP implementations do not actually marshal data entirely with precompiled stub routines. Because CORBA type-codes tend to drive IIOP marshalling, many CORBA implementations use interpreted marshalling mechanisms, such as described by Gokhale and Schmidt [25].

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

tations, the ubiquity of twos-complement and IEEE floating representations make the potential disadvantage of a receiver-makes-right approach (the needed support of $N \times N$ different translations given N different representations) mainly theoretical.

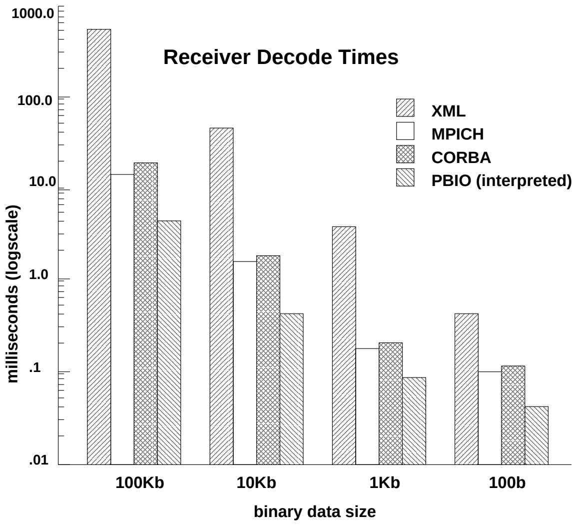

Figure 9: Receive side decode times.

package significant flexibility in reacting to changes in the incoming data and was our initial choice when implementing NDR.

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 XML parser [24] 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 employed is quite fast, but XML still pays a relatively heavy penalty for requiring string-to-binary conversion on the receiving side.

4.3.1 Comparison of Receiver-Side Costs

Figure 9 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by XML, MPICH (via the `MPI_Unpack()` call), CORBA, and PBIO. In this case, we use an early PBIO implementation in which a generalized routine walked a data structure to convert the data from the incoming format to the native format. XML-receiver's[8] conversions are clearly expensive, typically between one and two orders of decimal magnitude more costly than our NDR-based converter for this heterogeneous exchange. On an exchange between homogeneous architectures, PBIO, CORBA and MPI would have substantially lower costs, while XML's costs would remain unchanged. Our NDR-based converter is highly 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 is done in PBIO). However, NDR'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 attributed to the raw number of operations involved in performing the data conversion, we believe that a significant fraction of this overhead is due to control costs in what is, essentially, an interpreter-based approach.

4.3.2 Optimizing Receiver-Side Costs for PBIO

As previously described, our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time. As a result, the nature of the wire-to-native conversion is also unknown until runtime, and we cannot employ to compile-time techniques that generate efficient and specialized unmarshalling code. However, we can use a similar approach in a more dynamic way. In particular, PBIO's implementation of NDR makes use of dynamic code generation to create a customized conversion subroutine for every incoming record type. These routines are generated by the receiver on the fly, as soon as the wire format is known. PBIO dynamic code generation is performed using a Georgia Tech DCG package (DRISC) that provides a virtual RISC instruction set. Early versions of PBIO used the MIT Vcode system [26].

The instruction set provided by DRISC is relatively generic, and most instruction generation calls produce only one or two native machine instructions. Native machine instructions are generated directly into a memory buffer and can be executed without reference to an external

[8] As noted earlier, we used the fastest known to us, Expat [24].

Record Size	Number of generated instructions	time required to generate
100Kb	884	849μs
10Kb	540	732μs
1Kb	308	505μs
100b	156	446μs

Table 1: The generation times for DCG conversion routines for various record sizes. The counts are for Sparc instructions.

compiler or linker.[9]

Employing DCG for conversions means that PBIO must bear the cost of generating the code as well as executing it. Because the format information in PBIO is transmitted only once on each connection and data tends to be transmitted many times, conversion routine generation is not normally a significant overhead. The proportional overhead encountered varies significantly depending upon the internal structure of the record. To understand this variability, consider the conversion of a record that contains large internal arrays. The conversion code for this case will consist of a few `for` loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field.

Dynamic code generation on the receiver happens only once for each message type (when it first arrives). As a result, the overhead imposed by creating the conversion routines is not typically an important factor in steady-state message exchange and a full analysis of the factors controlling generation time is beyond the scope of this paper. However, for completeness we have included in Table 1 the generation times for the Sparc-side conversion routines used in the x86-to-Sparc exchange for each message size.

For the reader desiring more information on the precise nature of the code that is generated, we include a small sample subroutine in Figure 10. This particular conversion subroutine

[9] More details on the nature of PBIO's dynamic code generation can be found in [27].

converts message data received from an x86 machine into native Sparc data. Figure 1 shows the structure of the message being exchanged.

Since the record is being sent from an x86 to a Sparc, the "wire" and the receiver native formats differ in both byte order and alignment. In particular, the floating point value is aligned on a 4-byte boundary in the x86 format and on an 8-byte boundary on the Sparc. The subroutine takes two arguments. The first argument in register %i0 is a pointer to the incoming "wire format" record. The second argument in register %i1 is a pointer to the desired destination, where the converted record is to be written in native Sparc format.

The exact details of the code are interesting in two ways. First, we make use of the SparcV9 Load from Alternate Space instructions, which can perform byteswapping directly as part of the load operation. Since this is not an instruction that is normally generated by compilers in any situation, being able to use it directly in this situation is one of the advantages of dynamic code generation.

Second, from an optimization point of view, the generated code is actually quite poor. Among other things, it performs two instructions when one would obviously suffice, and unnecessarily generates an extra load/store pair to get the double value into a float register. There are several reasons for this suboptimal code generation, including the generic nature of the virtual RISC instruction set used as an intermediate language, the lack of an optimizer to repair it, and the fact that we have not seriously attempted to make the code generation better. Even when generating poor code, DCG conversions are a significant improvement over other approaches.

Examining the generated code may also bring to mind another subtlety in generating conversion routines: data alignment. The alignment of fields in the incoming record reflects the restrictions of the sender. If the receiver has more stringent restrictions, the generated load instruction may end up referencing a misaligned address, a fatal error on many architectures. This situation would actually have occured in the example shown above, because the incoming `double` array is aligned on a 4-byte boundary and the Sparc requires 8-byte alignment for 8-byte loads. The dynamic code generator detects the fact that a direct 8-byte load of the `double` values would be misaligned and loads the two halves of the incoming doubles with separate `ldswa` instructions instead of a single `lddfa` instruction.

start of procedure bookkeeping
```
save  %sp, -360, %sp
```
byteswap load and store the 'ivalue' field.
```
clr  %g1
ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
st  %g2, [ %i1 ]
```
byteswap load and store the 'dvalue' field
```
mov  4, %g1
ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
mov  8, %g1
ldswa  [ %i0 + %g1 ] #ASI_P_L, %g3
st  %g3, [ %sp + 0x158 ]
st  %g2, [ %sp + 0x15c ]
ldd  [ %sp + 0x158 ], %f4
std  %f4, [ %i1 + 8 ]
```
loop to handle 'iarray'
save 'incoming' and 'destination' pointers for later restoration
```
st  %i0, [ %sp + 0x160 ]
st  %i1, [ %sp + 0x164 ]
```
make regs i0 and i1 point to start of incoming and destination float arrays
```
add  %i0, 0xc, %i0
add  %i1, 0x10, %i1
```
setup loop counter
```
mov  5, %g3
```
loop body.
```
clr  %g1
ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
st  %g2, [ %i1 ]
```
end of loop, increment 'incoming' and 'destination', decrement loop count, test for end and branch
```
dec  %g3
add  %i0, 4, %i0
add  %i1, 4, %i1
cmp  %g3, 0
bg,a   -7
clr  %g1
```
reload original 'incoming' and 'destination' pointers
```
ld  [ %sp + 0x160 ], %i0
ld  [ %sp + 0x164 ], %i1
```
end-of-procedure bookkeeping
```
ret
restore
```

Figure 10: A sample DCG conversion routine.

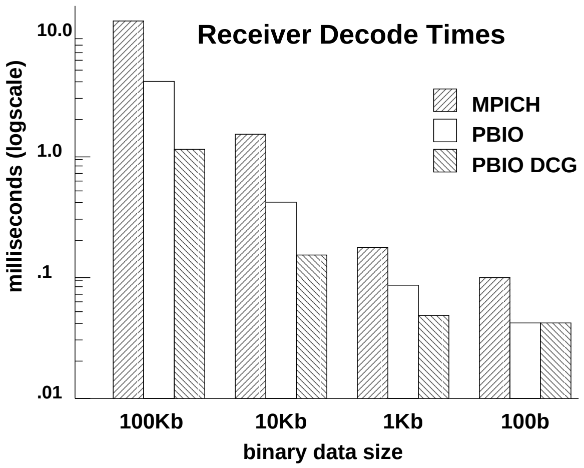

Figure 11: Receiver side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

The benefits derived from the use of DCG are apparent from the execution times for these dynamically generated conversion routines, which are shown in Figure 11 (we have chosen to leave the XML conversion times off of this figure to keep the scale to a manageable size). From these measurements, it is clear that 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, and it is the key to PBIO's ability to efficiently perform many of its functions.

The cost savings achieved for PBIO described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 12 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes.[10] The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time required by MPICH. The performance gains are due to:

[10]We choose MPICH for detailed comparison because it most closely matches PBIO performance. The results for our sample CORBA IIOP implementation would be similar. Despite the fact that CORBA has a receiver-makes-right marshalling mechanism, the copies performed by the marshalling mechanism give it a very similar performance profile to MPICH in this heterogeneous exchange.

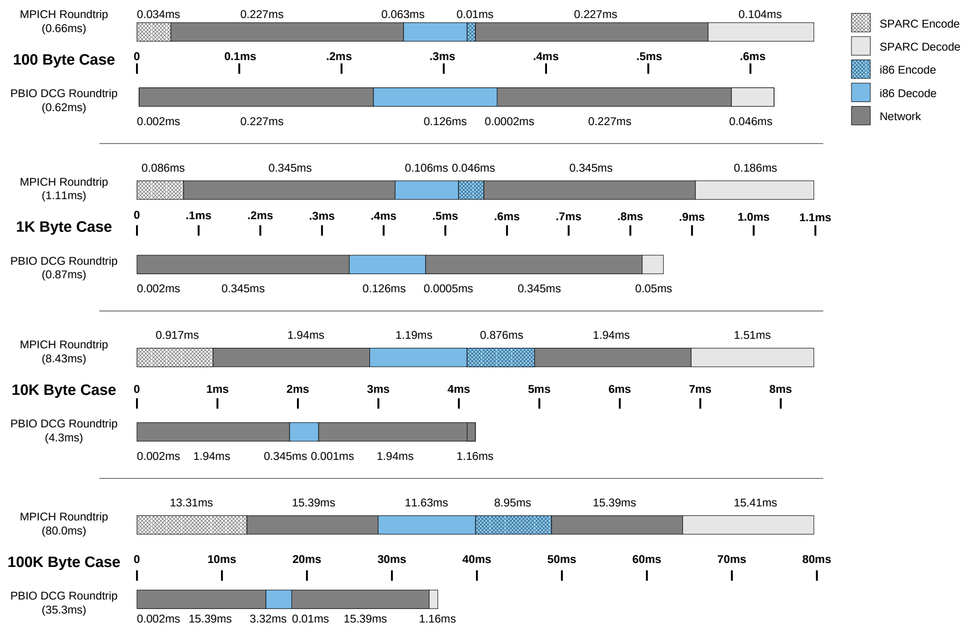

Figure 12: Cost comparison for PBIO and MPICH message exchange. The PBIO total costs are shown with a bar whose length is normalized to the corresponding MPI cost.

Original	Round-trip time			
Data Size	XML	MPICH	CORBA	NDR
100Kb	1200ms	80ms	67ms	35ms
10Kb	149ms	8.4ms	8.8ms	4.3ms
1Kb	24ms	1.1ms	1.0ms	0.87ms
100b	9ms	.66ms	0.6ms	0.62ms

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

- 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.

Once again, Figure 12 does not include XML times to keep the figure to a reasonable scale. We use MPI as representative to keep the figure simple and so that the reader can see the relative contribution of the costs that go into a round-trip exchange.

Table 2 summarizes the relative costs of the round-trip exchange with XML, MPICH, CORBA, and PBIO and essentially provides a summary of the results presented in this section. XML, the only other marshalling approach that supports the PBIO-like meta-information necessary for plug-and-play communication, suffers the triple disadvantages of high encoding costs, high decoding costs and high network transmission times (because of expansion to ASCII and inline meta-information). The result is a message round-trip time that is at least an order of magnitude longer than the others. When compared with the less flexible approaches, PBIO offers a small but noticeable advantage for the smallest message sizes. This advantage grows to be more significant as the message size grows and the effects of data copying begin to have a larger impact on end-to-end message times.

4.4 High Performance and 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 a useful tool in building and deploying enterprise-level distributed systems because it allows generic components to operate upon data about which they have no *a priori* knowledge, and because it allows the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other words, PBIO offers limited support for *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. Its support of type extension derives from performing 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 that support reflection and type extension in messaging, such as systems using XML as a wire format or marshalling objects as messages, suffer prohibitively poor performance compared 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 effects of type extension by introducing an unexpected field into the incoming message and measuring the change in receiver-side processing. Note that we limit ourselves to comparing different aspects of PBIO's own performance because we are not aware of another reasonably high-performance communication system that supports these features.

Figures 13 and 14 present receiver-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 is clear from Figure 13 that the extra field has no effect upon the receiver-side performance. Transmission would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 14 shows the effect of the presence of an unexpected field in the homogeneous case.

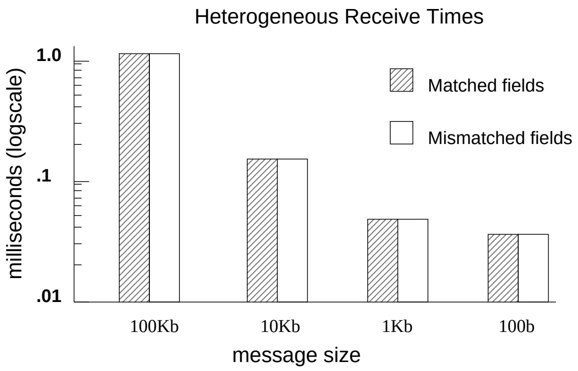

Figure 13: Receiver-side decoding costs with and without an unexpected field: Heterogeneous case.

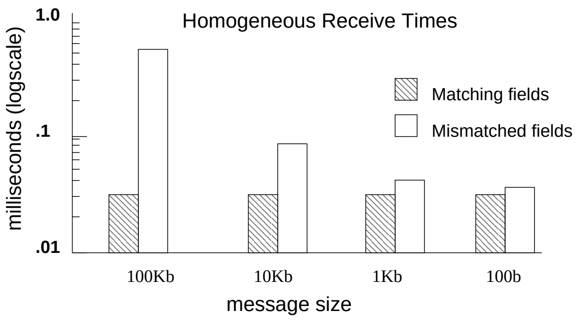

Figure 14: Receiver-side decoding costs with and without an unexpected field: Homogeneous case.

Here, the overhead is potentially significant because a homogeneous exchange would normally not impose any conversion cost in PBIO. The presence of the unexpected field creates a layout mismatch between the wire and native record formats that requires the relocation of fields by the conversion routine. As the figure shows, the resulting overhead is not negligible, but it is never as high as that incurred by the receiver of a heterogeneous exchange (Figure 13). 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 a `memcpy()` operation for the same amount of data.

As noted earlier in Section 4.3, XML is extremely robust with respect to changes in the format of the incoming record. Essentially, XML transparently handles precisely the same types of change in the incoming record as 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 9. However, those costs are several orders of magnitude higher than those of PBIO.

For PBIO, the results shown in Figures 13 and 14 are actually based upon a worst-case assumption, where an unexpected field appears before all expected ones 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 actual mismatch. An evolving application might exploit this feature of PBIO by adding additional fields at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

5 Conclusions and Future Work

This paper describes and analyzes a basic factor determining the performance and flexibility of modern high performance communication infrastructures: the choice of "wire formats" for data transmission. After examining the performance implications of different wire formats, we propose Native Data Representation, an alternative approach for flexible high-performance data exchange in heterogeneous distributed systems and evaluate its current implementation.

Our NDR approach increases application flexibility by allowing receivers to make run-time decisions on the use and processing of incoming records without *a priori* knowledge of their formats. NDR achieves high performance by reducing copy and conversion overheads at senders, and by avoiding the potential costs of complex format conversion on the receiver's end through runtime binary code generation. As a result, the additional flexibility comes at no cost and, in fact, the performance of PBIO transmissions exceeds that of data transmissions performed in modern HPC infrastructures like MPI.

The current implementation of PBIO has some practical limitations that could be addressed with additional development. For example, PBIO relies on the underlying machine for floating point data conversions. Even with the ubiquity of IEEE floating point, this can be a problem if the particular size of a sender's IEEE floating representation (such as 128-bit long double) does not exists on the receiving end. This difficulty could be addressed with the creation of a set of synthetic type conversion routines. As we previously noted, supporting machines that use of non-IEEE floating point representations or integer representations other than twos-complement would also complicate PBIO, although this does not seem to be an issue with modern machines. Finally, PBIO does not have any facility to detect or report under/overflow in any conversion, but such checks could be easily added to the generated conversion code.

In conclusion, this paper has demonstrated that the goals of *flexibility* and *high-performance* for dataexchange communication are not inherently incompatible. NDR supports the flexibility of XML-style plug-and-play communication with performance comparable to that of more fixed wire format based approaches. Lastly, we have shown that dynamic code generation, while not absolutely required, can play a valuable role in reducing end-to-end communication delays.

Several areas for future work exist, including detailed investigation of the one-time-costs required to implement NDR. The general approach taken by PBIO includes a facility for out-of-band transmission of message format information. Section 3.2.2 discussed our current arrangement for format servers which provide this format information, but many others issues remain to be addressed in this area, including making such a service more highly available and understanding the performance implication of server placement and replication, particularly in mobile or less well-connected environments. We are also generalizing the notion of message format to include XML "markup" information [22]. This allows XML/PBIO messages to be transmitted in binary and "hydrated" to XML text or DOM tree upon arrival. Because the format is cached at the receiver, this approach should have significant advantages over transmitting textual XML directly.

We are continuing to develop the compiler techniques necessary to generate efficient binary code, including the development of selected runtime binary code optimization methods and the development of code generators for additional platforms, most notably the Intel StrongArm and new HP/Intel RISC platforms. PBIO's approach of transmitting self-describing messages directly in binary also allows very efficient processing of messages at intermediate points in the communication [28]. We have extended the dynamic code generation tools with a simple high-level language that allow "in-flight" access to message contents for content-based routing purposes, quality of service, message data reduction or other purposes. Future papers will address the broader implications of these techniques.

In related research, our group is developing high performance communication hardware and firmware for cluster machines, so that PBIO-based messaging may be mapped to zero-copy communication interfaces and so that selected message operations may be placed directly 'into' the communication co-processors being used [10, 29].

References

[1] B. Parvin, J. Taylor, G. Cong, M. O'Keefe, and M.-H. Barcellos-Hoff, "Deepview: A channel for distributed microscopy and informatics," in *Proceedings of Supercomputing '99 (SC1999)*, November 13-19 1999.

[2] C. M. Pancerella, L. A. Rahn, and C. L. Yang, "The diesel combustion collaboratory: Combustion researchers collaborating over the Internet," in *Proceedings of Supercomputing '99 (SC1999)*, November 13-19 1999.

[3] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukji, B. Temko, and M. Yechuri, “A component based services architecture for building distributed applications,” in *Proceedings of the 9th High Performance Distributed Computing (HPDC-9)*, August 2000.

[4] S. Parker and C. R. Johnson, “SCIRun: A scientific programming environment for computational steering,” in *Proceedings of Supercomputing ’95 (SC1995)*, December 4-8 1995.

[5] T. Haupt, E. Akarsu, and G. Fox, “Webflow: A framework for web based metacomputing,” in *High-Performance Computing and Networking, 7th International Conference (HPCN Europe)*, pp. 291–299, April 1999.

[6] A. Wollrath, R. Riggs, and J. Waldo, “A distributed object model for Java system,” in *Proceedings of the USENIX COOTS 1996*, 1996.

[7] W3C, “Extensible markup language (XML).” http://w3c.org/XML.

[8] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener, “Efficient wire formats for high performance computing,” in *Proceedings of Supercomputing ’00 (SC2000)*, November 4-10 2000.

[9] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen, “JECho - interactive high performance computing with Java event channels,” in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.

[10] M.-C. Rosu, K. Schwan, and R. Fujimoto, “Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture,” *Cluster Computing, Special Issue on High Performance Distributed Computing*, vol. 1, pp. 51–67, January 1998.

[11] M. Welsh, A. Basu, and T. V. Eicken, “Incorporating memory management into user-level network interfaces,” in *Proceedings of Hot Interconnects V*, pp. 27–36, 1997.

[12] G. Eisenhauer, "Portable self-describing binary data streams," Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994.

[13] V. S. Sunderam, A. Geist, J. Dongarra, and R. Manchek, "The PVM concurrent computing system," *Parallel Computing*, vol. 20, pp. 531–545, March 1994.

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

[15] M. P. I. M. Forum, "MPI: A message passing interface standard," tech. rep., University of Tennessee, 1995.

[16] M. Lauria, S. Pakin, and A. A. Chien, "Efficient layering for high speed communication: Fast messages 2.x," in *Proceedings of the 7th High Performance Distributed Computing (HPDC-7)*, July 1998.

[17] Object Management Group, "The common object request broker architecture and CORBA 2.0/IIOP specification," tech. rep., OMG, December 1998. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[18] G. T. Almes, "The impact of language and system on remote procedure call design," in *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pp. 414–421, May 13-19 1986.

[19] S. W. O'Malley, T. A. Proebsting, and A. B. Montz, "Universal stub compiler," in *Proceedings of the symposium on Communications architectures and protocols (SIGCOMM '94)*, August 1994.

[20] D.D.Clark and D.L.Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proceedings of the symposium on Communications architectures and protocols (SIGCOMM '90)*, pp. 200–208, September 26-28 1990.

[21] M. Schroeder and M. Burrows, "Performance of Firefly RPC," in *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 83–90, December 1989.

[22] P. Widener, G. Eisenhauer, and K. Schwan, "Open metadata formats: Efficient XML-based communication for high performance computing," in *Proceedings of the 10th High Performance Distributed Computing (HPDC-10)*, August 2001.

[23] Unknown, "MPICH - a portable implementation of MPI." Argonne National Lab, http://www-unix.mcs.anl.gov/mpi/mpich.

[24] J. Clark, "expat - XML parser toolkit." http://www.jclark.com/xml/expat.html.

[25] A. Gokhale and D. Schmidt, "Principles for optimizing corba internet inter-orb protocol performance," 1998.

[26] D. R. Engler, "Vcode: a retargetable, extensible, very fast dynamic code generation system," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[27] G. Eisenhauer and L. K. Daley, "Fast heterogenous binary data interchange," in *Proceedings of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000.

[28] F. E. Bustamante, *The Active Streams Approach to Adaptive Distributed Applications and Services.* Ph.d. Thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA, November 2001.

[29] R. Krishnamurthy, K. Schwan, R. West, and M. Rosu, "A network coprocessor based approach to scalable media streaming in servers," in *International Conference on Parallel Processing (ICPP 2000)*, August 2000.