Lightweight Morphing Support for Evolving Middleware Data Exchanges in Distributed Applications

Sandip Agarwala, Greg Eisenhauer and Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, eisen, schwan}@cc.gatech.edu

Abstract

Most systems must evolve as their missions or roles change and/or as they adapt to new execution environments. When evolving large distributed applications, it is particularly difficult to make changes to the data formats that underlie their components' communications, because such 'format evolution' can affect all or many application components. Prior approaches to the problem of implementing changes in the communications of a deployed system have relied upon ad-hoc solutions or on protocol negotiation to avoid message format mismatches. Unfortunately, such solutions tend to increase the complexity of application code. This paper presents a novel approach to the problem of data format evolution that combines meta-data about the data being exchanged with dynamic binary code generation to create a robust data exchange system that naturally supports application evolution. The idea is to specialize the communications of application components by dynamically generating the code that can automatically transform incoming data into forms that receiving components can understand. A realistic example in the context of publish/subscribe middleware is used to illustrate how this technique can be applied to enhance interoperability between different version of distributed applications.

1 Introduction

Large-scale distributed applications communicate via complex data exchanges. Data structures and formats are often assumed known *a priori*, thereby permitting the creation of efficient marshalling and unmarshalling codes represented as stubs and generated by stub compilers[5]. However, the needs and requirements of distributed applications and the resources presented by the underlying distributed execution platforms can change over time: (1) independent developers may implement new or improved functionalities using enhanced message data formats or exchange protocols, and (2) heterogeneity or dynamic changes in hardware resources (e.g., low bandwidths of newly employed wireless links) may necessitate further changes in message format representation. We term such changes in data exchange messages "*format evolution*".

Format evolution presents many challenges to application developers. In large or long-running distributed applications, format evolution is difficult because of the need to support interoperability between old and new clients and servers [29]. Emerging middleware technologies like publish-subscribe [1, 4] and peer-to-peer [30, 35] further complicate matters because they lack central control and eliminate clear client-server demarcations. Interoperability is also important in the case of software reuse and legacy applications that may not be easy to replace [28, 13]. Finally, while protocol designers aim to ensure that new message formats are backward compatible, the control of such processes is difficult when multiple designers concurrently make evolutionary changes.

Previous work on message interoperability has focused on network-level solutions [16]. Such solutions are difficult to generalize to the middleware-level problem of format evolution addressed by our research because of the large diversity of application-level data structures and types. As a result, application-level protocol (or format) evolution is rarely addressed by application developers. Instead, message formats and other elements of message exchange protocols are frozen early in the development process, limiting an application's ability to evolve to better serve changing needs. Emerging approaches to the format evolution problem [7] tend to be complex or impose overheads that are unsuitable for systems with high-bandwidth data flows.

At the networking level, a common approach to message evolution is protocol version negotiation between interacting components. Negotiation is not viable for large-scale systems, because it is impractical to negotiate with many clients, where each such negotiation 'transaction' has significant overheads (e.g. multicast). Another approach is to include "*extensions*" identified by unique global IDs and processed by the components that understand them [21]. An alternative solution used by object-based systems assumes that messages carry substantial meta-data, which is exploited to enable message evolution. The performance impact of carrying meta-data on high-volume data transfers makes this approach problematic. Another issue is that the current abilities of this approach remains limited. For example, XML-based systems using SAX or DOM interfaces can easily tolerate the addition of data or the reordering of certain fields in existing protocols without 'breaking' old clients, but more radical restructuring (*e.g.*, changing the structure associated with messages) is not feasible. Finally, in middleware architectures with typed messaging, like OMG's CORBA [5], Microsoft's DCOM [8], and Sun's Java RMI [27], interactions between two different data types are not allowed, despite the fact that it may make sense to permit those interactions that may be semantically correct. For example, if a message from a new server contains an extra field that provides optional information, clients who do not understand or expect that field should still be able to operate. Disallowing such interactions unnecessarily constrain inter-operation in distributed systems.

This paper introduces the concept of "*Message Morphing*", which combines message meta-data and dynamic code generation to help data exchanges deal with varying message formats and support runtime *message evolution*. The basic idea is to associate a number of transformations with each new message data format so that the data can be converted to other formats. If the receiver is not able to understand a particular format, an appropriate transformation code associated with that format is provided and dynamically compiled. The conversion routine transforms each incoming message of that type to one the receiver is able to understand.

Dynamic message morphing has a number of advantages. First, because there is no negotiation, this technique is as easily applied to peer-to-peer as to client-server architectures, and it can address components separated in space and/or time. Second, its use can guarantee both syntactic and semantic compatibility, as per the transformations supplied to the application. Third, there is no need to modify or restart an application, because format conversions occur on the fly and without specific application involvement. In fact, the application may be entirely oblivious of the evolutions applied to its components. Fourth, our implementation of message morphing is highly efficient, as it uses out-of-band, binary meta-data (*i.e.*, the PBIO [12] binary data format), the use of which has been shown to result in marshalling overheads less than those of well-known high performance communication systems like MPI [20].

The remainder of the paper discusses prior approaches used to address the problem of message format evolution in Section 2. Section 3 explains in some detail the algorithms and techniques used in Message Morphing. In Section 4, we present two real scenarios for which an interoperability problem is solved using morphing. We evaluate the costs and overheads of message morphing with PBIO and compare it with that of XML/XSLT in Section 5. Section 6 concludes the paper and presents future research directions.

2 Related work

A large body of work addresses the need for interoperability between different components of a distributed application, across network protocols, or for legacy applications. Such work characterizes interoperability as the set of assumptions communicating parties make about each other [13]. Current practice in dealing with the difficulties caused by these assumptions is to use ad hoc techniques or to use protocol negotiation mechanisms to decide upon a common language of communication.

Object-based systems use sub-classing and similar techniques to interoperate across different types. *Network Objects* [3], for example, use the *narrowest surrogate rule*, by which a client chooses the narrowest super-type for which both the client and the owner of that object have a registered stub. This is possible because Modula-3 (which is used to implement network objects) only allows single inheritance. CORBA-based [5] systems support interoperability through GIOP (General Inter-ORB Protocol), which describes the wire-representation of the messages exchanged between different ORBs (Object Request Brokers).In addition to static invocation (object types are known at runtime), CORBA also supports object invocation using runtime type information (usually retrieved from an *Interface Repository*). Solutions to interoperability in systems like Network Objects or CORBA are less general than those offered by message morphing because they focus on type information. In other words, these solutions do not handle messages that may be semantically correct but are not subtypes of registered types. In comparison, message morphing can be used with syntactically mismatched messages as well, if developers specify suitable morphing code.

Spreitzer and Begel [29] observe that existing sub-typing techniques do not scale well because of the exponential increase in code size due to the number of extensions each particular client must understand. Message morphing manages this issue by using dynamic code generation (DCG), so that morphing code is installed in a component only if needed. The authors also discuss the additional latency in decentralized evolution incurred by the necessary ad hoc negotiations between clients and servers. The solutions described in [29] present another way in which certain message morphing actions may be automated. Specifically, the authors propose a flexible data type system consisting of extensible and coarse record types. The fields of extensible records are marked with a "*mode*" flag that indicates whether those fields are optional. The coarse record types ensure that extensions are compatible with sub-typing. The receiver is responsible for understanding of each input and determining if there is a need for negotiation.

Lee *et al.* [17] uses object-oriented pattern matching to extract relevant information from self-describing messages. Applications specify handlers that are invoked when a particular pattern is matched. This technique allows optional fields to be included in a message, and it implies that developers need not write explicit code for parsing messages. The overheads of text-based pattern-matching, however, prevents its use in the high performance environments addressed by message morphing[12]. The same argument applies to XML-based approaches, where (1) XML [33] already supports some forms of evolution by allowing the use of optional and ignorable items, and (2) message morphing techniques like those described in this paper could be applied to XML-structured messages by using transformation languages like XSLT [34] or fxt [2]. However, we borrow XML-style type mapping semantics based on field names, where a field may be associated with a default value. Message morphing generalizes that approach by using format information in addition to default field values. Using an efficient binary format description (*i.e.*, PBIO [12]), the feature unique to our solution is the ability to dynamically convert formats using the transformations that are associated with each format type. This approach expands the *compatibility space* of the application, as discussed in the next section.

3 Message Morphing Approach

3.1 Expanding Compatibility Space

The set of all message format types and/or protocol versions with which an application can successfully interoperate is called its *compatibility space*. The use of binary messaging typically implies that small changes negate compatibility. This is devastating for enterprise-scale applications, because it inhibits certain updates or improvements for currently deployed codes. Message morphing is a technique to expand the compatibility space to a broader set of changes.

Our approach makes some basic assumptions that enable a messaging system to implement message morphing. First, we assume that messages are associated with meta-data that describes message content, similar to 'schemas' in XML-based systems. Second, we assume that on the receiving side, the system is aware of the set of message schemas or formats the receiving application is able to process. In a complex distributed system, Application A may communicate with Application B using a set of message formats known and interpretable by both sides (*i.e.* Protocol X). As the distributed system evolves, some applications may be upgraded to use a newer Protocol Y, an evolutionary upgrade from Protocol X. If protocol negotiation or other techniques are not used, a newer client may send a Protocol Y message to a client that only understands Protocol X messages. Without message morphing, this situation will almost certainly result in some kind of failure. Message morphing avoids the failure scenario by associating additional meta-data with Protocol Y messages so that they can be transformed, on-demand, into Protocol X messages (see Figure 1). This approach removes fixed message formats as an obstacle in upgrading and evolving complex distributed applications.

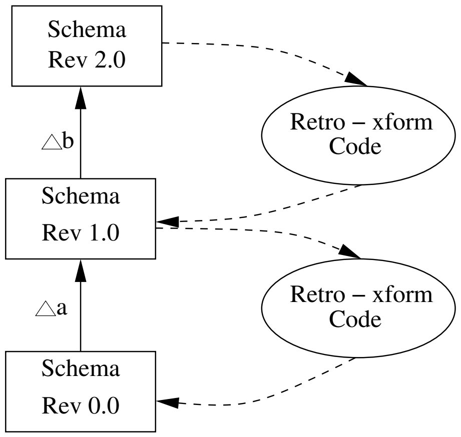

Figure 1. Retro - Transformation

In message morphing, most of the activity happens upon message receipt. Assume that message M is part of Protocol X. In Protocol Y, M has been upgraded to M', a variant of M that still fills the same role and contains largely the same data, but perhaps differs in the way message data is organized. Assume you have a distributed system consisting of some old clients that speak only Protocol X and some new clients that speak Protocol X and Protocol Y. In our scenario, clients that speak the newer Protocol Y always send Protocol Y messages, even to older clients. However, the Protocol Y message M' meta-data includes a specification of how to transform it into message M of Protocol X. Upon receipt at a newer client, message M' is processed normally. Upon receipt at an older client that expects only Protocol X messages, message M' is transformed into message M by the communication system before being delivered to the client. The details of this transformation will be discussed later in the paper.

The technique is not without its limitations. There are many possible ways to change an application-level message-passing protocol that render it incompatible with unchanged clients: adding messages, changing the semantics of messages, removing previously-required message information completely, *etc.* We do not address all of these cases. Instead, message morphing should be viewed as another technique that can enlarge the natural compatibility space in application-level protocols. In particular, message morphing allows natural compatibility in some situations that might otherwise require a costly or complex protocol negotiation phase. Where protocol negotiation is impossible or unwieldy, such as when data is multicast or when negotiation overheads would be too high, message morphing enables interoperability in some situations where it would otherwise be impractical.

3.2 Implementation

The meta-information about message data in our implementation of message morphing utilizes the *Portable Binary Input/Output* (or PBIO) system [12]. PBIO is a record-oriented binary communication mechanism that supports out-of-band meta-data. Writers (or encoders) of data provide a description of the names, types, sizes and positions of the fields in the records they are writing. Readers (or decoders) provide similar information for the records they are interested in reading. They also register handlers that are invoked when a message of a particular format type arrives. Figure 2 shows a sample format declaration. Each format has a name associated with it, which may be the same or different from that of other format. In addition to record formats, the writer may also specify a set of transformations, which can convert the message from one format to the other. These transformations are specified in the form of "*Ecode*" [10] (a language subset of C).

```
typedef struct {
   int cpu;
   int memory;
   int network;
} Msg, *MsgP;

IOField Msg_field[] = {
   {"load", integer, sizeof(int), IOOffset(MsgP, load)},
   {"mem", integer,sizeof(int), IOOffset(MsgP, memory)},
   {"net", integer, sizeof(int), IOOffset(MsgP, network)}};
```

Figure 2. PBIO format declaration

On the reader's end, the format of the incoming record is compared with the reader's registered formats of the same name. If an exact match is found, the handler registered for that format type is invoked, else a maximal matching algorithm called *MaxMatch* (described next) is used to decide the best possible conversion available for that incoming new record format. If a conversion is found, it is applied to the new format and the handler registered for the converted format is invoked; otherwise a default handler (if any) registered by the reader is called. This conversion is now cached at the reader, and will be applied to all incoming records of this format type.

***MaxMatch comparison algorithm*:** We have designed an algorithm called *MaxMatch* which gives the best possible matching pair of formats from the two set of formats. The best possible matching pair has least differences between them as compared to other pairs. Before going into the details of algorithm, we provide the following definitions:

- PBIO record formats consist of fields that can be of two types: *basic* and *complex*.
  - *basic* types includes integer, unsigned integer, float, char, enumeration and string
  - *complex* types are formed from a collection of other fields which can be both *basic* and *complex*.
- The top-level format, which defines an entire message record, is called *base* format. Note that *base* format is always *complex*.
- The *weight* of a format f, indicated by W_f is the total number of fields in f which includes the number of basic type fields within its complex type fields as well.

Given two formats f_1 and f_2, we define a recursive function *diff*, s.t.:

$$\text{diff}\ (f_1, f_2) = d_{12}$$

where, d_{12} is the total number of basic type fields that are present in f_1 but not in f_2. Algorithm 1 shows how to compute *diff*.
(f_1, f_2) is a *perfect matching* format pair iff:

$$\text{diff}(f_1, f_2) = \text{diff}(f_2, f_1) = 0$$

A format pair (f_1, f_2) is said to have less "*mismatch*" than another format pair (f_1', f_2') iff one of the following condition holds:

i. $\text{diff}(f_1, f_2) < \text{diff}(f_1', f_2')$ or
ii. $\text{diff}(f_1, f_2) = \text{diff}(f_1', f_2')$ and $\text{diff}(f_2, f_1) < \text{diff}(f_2', f_1')$

Note that a format pair with the least *mismatch* need not be the best matching pair. For example, two formats (say f_1 and f_2) each may have one field, both of which are different. Therefore, $\text{diff}(f_1, f_2)$ is just 2. While another two formats (say f_1' and f_2') may have four uncommon fields ($\text{diff}(f_1', f_2')$ = 4) but hundred matching fields. Clearly, the second one is a better match than the first one. So we define a normalization metric called *Mismatch Ratio* to find the best match.

Algorithm 1 diff (f_1, f_2)

```
init: $d_{12} = 0$
for all field f in $f_1$ do
  if f is of basic type then
    if f $\notin f_2$ then
      $d_{12} \leftarrow d_{12} + 1$
    end if
  else
    let $f'$ be the complex field in $f_2$ with the same field name
    and type as f
    if no such $f'$ exists in $f_2$ then
      increment $d_{12}$ by $W_f$
    else
      $d_{12} \leftarrow d_{12}$ + diff (f, $f'$)
    end if
  end if
end for
return $d_{12}$
```

Mismatch Ratio(M_r) of a format pair (f_1, f_2) is defined as the ratio between the total number of fields present in f_2 and absent in f_1 to the total number of fields in f_2. Thus,

$$M_r(f_1, f_2) = \frac{diff(f_2, f_1)}{W_{f_2}}$$

Now we are ready to define *MaxMatch*. *MaxMatch* pair between two sets of formats F_1 and F_2 is:

$$\text{MaxMatch } (F_1, F_2) = (f_1, f_2),$$

such that the following condition holds:

i. $f_1 \in F_1$,
ii. $f_2 \in F_2$,
iii. diff(f_1, f_2) $<=$ DIFF_THRESHOLD,
iv. $M_r(f_1, f_2) <=$ MISMATCH_THRESHOLD,
v. if there are more than one (f_1, f_2) pair that satisfies the above four conditions, choose the one with least M_r, then the one with least diff(f_1, f_2) and break ties arbitrarily.

The two constants above (DIFF_THRESHOLD and MISMATCH_THRESHOLD) add another dimension of flexibility by allowing control of the amount of mismatch that will be allowed in a particular system. This prevents two grossly incompatible messages from being matched. In order to allow just *perfect matches*, set DIFF_THRESHOLD to zero.

Reader-side message processing:

The reader processes the incoming encoded message in a number of steps, which are outlined in Algorithm 2. There are some interesting attributes of this algorithm. First, the expensive steps of the algorithm (from 11-27) are executed for only those formats that have not been seen previously by the reader. Once a format has been seen, the transformation and the handler information are cached and will be used

Algorithm 2 Receiver-side message processing

```
    Init:
    Let m be the incoming message.
    Let $f_m$ be the format of m.
    Let $F_r$ be the set of formats with the same name as $f_m$
    that this reader can interpret.
 5: Let $F_t$ be the set of formats that $f_m$ can be transformed
    to (using the transformations associated with it) includ-
    ing $f_m$.

    if $f_m$ seen previously then
      Use cached information to transform the message (if
      needed) and invoke the appropriate format handler
      return
10: end if
    ($f_1$, $f_2$) $\leftarrow$ MaxMatch($f_m$, $F_r$)
    if ($f_1$, $f_2$) is a perfect match then
      Invoke handler registered by the reader for $f_2$
      return
15: end if
    ($f'_1$, $f'_2$) $\leftarrow$ MaxMatch($F_t$, $F_r$)
    if there is no such ($f'_1$, $f'_2$) then
      Reject this message
      return
20: end if
    if $f'_1 \neq f_m$ then
      Generate and cache the code that would do the $f_m$ to
      $f'_1$ transformation using dynamic code generation
      Transform 'm' from its original format $f_m$ to $f'_1$
    end if
25: if ($f'_1$, $f'_2$) is not a perfect match then
      Put in the default values for the missing fields.
      Remove fields in $f'_1$ that are not in $f'_2$
    end if
    Invoke handler for $f'_2$
30: return
```

when messages of that format are received again. Second, the *MaxMatch* algorithm is used to find the best possible format match available. If there is no such match, it simply rejects the messages with that format. Third, the algorithm shows how *message morphing* is used to expand the *compatibility space*. New formats that were previously not understood by the reader are converted to match older, understood formats, and enter into the compatibility space of the reader. This provides both backward and forward compatibility and allows for interoperability between both old servers and new clients and new servers and old clients.

Message morphing has several advantages:

- With the expansion of the compatibility space, the client or the reader is now able to recognize far more message formats than was previously possible.
- Applications are no longer forced to use a less efficient

representation to ensure interoperability with all clients. This makes it easier to introduce new features and functionalities, servers become more scalable, and it enables more efficient use of resources (*e.g.*, concerning network bandwidth).

- Independent specialization of system is facilitated, and
- Complex systems become more malleable.

4 Examples

To understand how message morphing operates, we study evolution in two different systems. The first is an event-based publish-subscribe system, and the other describes evolution in a typical E-Commerce application.

4.1 The ECho Event Delivery System

ECho [11, 9] is a distributed event delivery middleware system, developed at Georgia Tech, that supports a publish-subscribe model of group communication. It uses channel-based subscriptions, similar to the CORBA Event Services [6]. An *event channel* is the mechanism through which event sinks and sources are matched. Source clients submit messages (or *events*) to a specific channel and only the sink clients subscribed to that channel are notified of these messages. Channels are the entities through which the extent of event propagation is controlled. Figure 3 depicts a set of processes communicating using event channels.

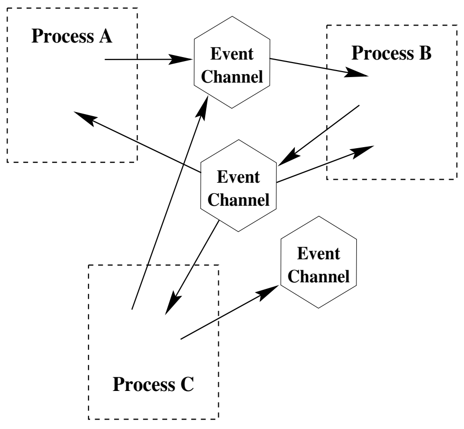

Figure 3. Processes using Event Channels for communication.

Since the release of the first version (v 1.0) of ECho in 1999, it has been used in research projects ranging from high performance applications [15, 23] and scientific visualization [32] to multimedia applications [24] and energy conservation in low power mobile devices [26]. ECho has evolved multiple times in order to adapt to changing needs and to accommodate improvements in its design. One such

ChannelOpenResponseMsg:
List of members:
CMcontact_info;
channel_ID;
List of Sources:
CMcontact_info;
channel_ID;
List of Sinks:
CMcontact_info;
channel_ID;
	ChannelOpenResponseMsg:
	List of members:
	CMcontact_info;
	channel_ID;
	is_Source; /*boolean*/
	is_Sink; /*boolean*/

a. Version 1.0	b. Version 2.0

Figure 4. ChannelOpenResponse message format in different ECho Version

instance of evolution was a change in its *ChannelOpenResponse* message. When a process wants to join a channel, it sends a *ChannelOpenRequest* message to the creator of that channel. The creator in turn replies back with a *ChannelOpenResponse* message, which consists of the list of processes already subscribed to that channel. Figure 4.a shows the format of this message in ECho version 1.0. *Member-list* is the list of all subscribers. The *src-list* and the *sink-list* contains all the processes subscribed as source and sink respectively. Note that the member-list in the message is a superset of the src-list and sink-list.

The fact that the contact information for a single remote client could appear three times in the *ChannelOpenResponse* message was an artifact of the incremental development of ECho. As ECho was extended to include Quality of Service information in connections, contact information became more complex and the multiple listings were an obvious liability, leading to duplicated code. In version 2.0, ECho changed the format of this message to simplify it and to reduce information duplication. The new format is shown in Figure 4.b. Note that two new boolean attributes were added to the list structure, which eliminated the need for two extra lists.

The *ChannelOpenResponse* change reduced the size of the response message by more than half, thereby reducing overhead, but it created an additional problem of interoperability with the older version of ECho. There is no standard existing technique that will allow the clients of these two formats to interoperate, although the messages of the new format contain all of the information carried by the messages with the old format. A quick workaround for this problem was to include version information in the *ChannelOpenRequest* message and send the appropriate version of the response. Though this solution worked, it had obvious disadvantages: extra computation at the creator of the channel, increased coding complexity (in terms of lines of code), and its lack of generality.

This situation is greatly simplified with message mor-

```
int i, sink_count = 0, src_count = 0;
old.member_count = new.member_count;
for (i=0; i < new.member_count; i++) {
   old.member_list[i].info = new.member_list[i].info;
   old.member_list[i].ID = new.member_list[i].ID;
   if(new.member_list[i].is_Source) {
     old.src_count = src_count + 1;
     old.src_list[src_count].info = new.member_list[i].info;
     old.src_list[src_count].ID = new.member_list[i].ID;
     src_count++;
   }
   if (new.member_list[i].is_Sink) {
     old.sink_count = sink_count + 1;
     old.sink_list[sink_count].info = new.member_list[i].info;
     old.sink_list[sink_count].ID = new.member_list[i].ID;
     sink_count++;
   }
}
```

Figure 5. Message transformation code

phing, by associating a transformation in the form of *ecode*(shown in Figure 5) with the new format at the channel creator. The morphing middleware at the subscriber side (with old ECho version) converts the response message to the old format by applying this transformation. Except for specifying the transformation code, no other changes are required anywhere in the system. Overhead is reduced because of smaller message sizes and the offloading of the processing to the subscriber.

The application based on the newer version of ECho can easily interoperate with those using older ECho versions, and they have lower overheads due to their more efficient message formats.

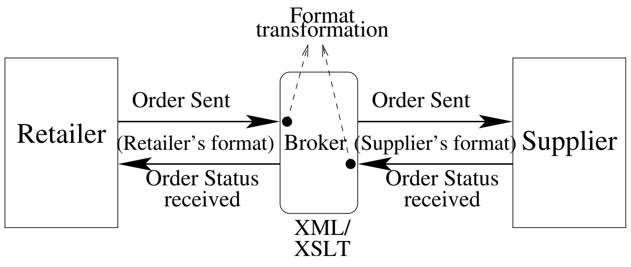

Figure 6. B2B Messaging with XML/XSLT.

4.2 Business Process Messaging

Business process messaging, (whether business-to-business (B2B) or business-to-consumer (B2C)) involves exchanging data between business applications to integrate the business work flow and processes. A major issue in such integration is the need to facilitate communication between different applications from different vendors, each generating data in their own formats. Figure 6 shows a *supply*

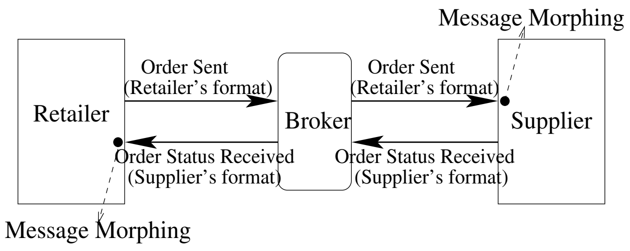

Figure 7. B2B Messaging with Morphing.

chain integration, where an intermediate broker acts as a bridge between retailers and suppliers by transforming the data exchanged between them. The broker may support one or both of point-to-point and publish/subscribe modes of communication. This is the typical architecture of today's many commercially-available solutions like Sun's Integration Server [31], IBM's WebSphere MQ [14], Oracle Advanced Queuing [22]. For illustration purposes, we will concentrate on the functionality of Oracle Advanced Queuing (AQ).

AQ provides the flexibility of configuring communication between different applications and uses XML as its data exchange format. In Figure 6, both retailer and supplier are using different message formats. To enable communication between them, the order data from the retailer is transformed by the AQ Broker using the appropriate XSL style-sheet into a format recognized by the intended supplier. Similarly, the order status data from the supplier is transformed by the AQ Broker into a format recognized by the specific retailer, again using an XSL style-sheet. This integration solution, though useful, is not without problems. First, it uses XSL transformations, which are computationally expensive. Second, all conversions are done by the broker, which can easily become a bottleneck. The problem might be temporarily solved by using a network of brokers, but this will again lead to increased costs and complexities.

Message Morphing provides an elegant solution to this problem. Here, the broker, instead of doing conversions by itself, simply associates an ECode segment with the incoming message. This code segment can transform it to a format recognized by its intended receiver (see Figure 7). The actual transformation is done at the receiver. Interoperability between the retailer and the supplier is still achieved, but with the added advantage of reduction in computational overheads at the broker. Also, adding new vendors with completely different formats becomes easier. The broker just has to be provided with the new ECode segments that can transform messages to the format understood by new vendors.

As noted earlier, message morphing is not a magic bullet that can create compatibility in every situation. Many types of changes to existing protocols, including changing the semantics of messages and removing previously-required information completely can make compatibility with unchanged clients impossible. Our aim in message morphing is to enlarge the natural compatibility space in application-level protocols. In our implementation of message morphing in the ECho publish/subscribe middleware, the use of PBIO for dynamic message formats provides some assistance because it provides XML-like capabilities without incurring the overheads of inline textual meta-data. Message morphing techniques extend that capability beyond simple structural similarity to cover situations where the new messages may be structurally dissimilar but still contain similar data in some form.

5 Evaluation

Systems that seek to aid application evolution can be evaluated on two basic criteria: the degree to which they expand the compatibility space of applications, and the extent of the overhead they add to basic communication. The former is difficult to evaluate in general, particularly when different techniques expand compatibility in different ways. In this paper, we consider the latter measure. In particular, we compare the performance of PBIO-based message morphing with that of XML/XSLT and demonstrate the feasibility of our method. XML is chosen because it is a widely-used form of communication and provides interoperability across heterogeneous environments. It also supports basic message evolution in the form of plug-and-play flexibility, as discussed earlier. XSLT (Extensible Stylesheet Language - Transformation) is a language for transforming the structure of an XML document. It is extensively used to produce HTML documents from XML as well as in business process messaging, as shown in the example above.

Because XSLT could be used to implement morphing of XML-encoded messages in the same fashion in which message morphing can transform binary messages, the most direct way to compare the two is to examine the costs associated with each transformation. However, to put those measures into proper context we also examine the encoding and decoding times associated with each of these encoding mechanisms, as well as the overall message sizes (which impacts network transmission time, a significant factor in overall message latency). In our evaluations, *libxml2* [18] Version 2.6.8 is used to parse XML messages, and the transformation of XML messages is carried out with *libxslt* [19] library Version 1.1.5. The tests are carried on a 2.2 GHz dual Intel XEON machine with 1GB memory and running the RedHat 9 linux distribution.

Encoding Cost: Figure 8 shows the cost of encoding a message of type *ChannelOpenResponse* Version 2.0 (shown in Figure 4.b) for five different sizes (obtained by varying the size of member list) in PBIO and XML. The XML string is created using *sprintf*() for data-to-string conversions and a modified version of *strcat*() which returns the pointer to the end of the string written rather than the start of the destination string. This saves time to re-locate the end of the string at each call. The cost of XML encoding in the figure includes the processing necessary to convert the data from binary to string form and to copy the element begin/end blocks into the output string. PBIO is optimized for marshalling structured binary data and not small strings.[1] Despite this, the encoding time for XML is still at least twice of that for PBIO.

Message size (KB)	.1	1	10	100	1000
Unencoded v2.0	.10	1.0	10	100	1000
PBIO Encoded v2.0	.13	1.0	10	100	1000
Unencoded v1.0	.23	2.9	30	300	2990
XML v2.0	.66	6.4	62	608	5956
XML v1.0	.79	12.0	121	1194	11712

Table 1. ChannelOpenResponse message size(in KB) in different format

Decoding cost without evolution: The encoded message obtained from above is then decoded using the respective decoder on the same machine. The decoder parses the encoded message and generates a data structure block similar to the one from which it was formed. Figure 9 shows the results of decoding XML- and PBIO-encoded messages. PBIO is much less expensive than XML for parsing encoded messages. This is possible because PBIO makes use of dynamic code generation to create a customized conversion subroutine for every incoming message type.

Message Size: Table 1 shows the sizes of two version of *ChannelOpenResponse* message when encoded with PBIO and XML. The unencoded *ChannelOpenResponse* message Version 2.0 is taken as the base line, and its size is varied from 100 bytes to 10MB. PBIO encoding adds less than 30 bytes of data to the original message, so the numbers in the first two rows are almost the same. On rollback to Version 1.0, the message size increases by three times because the list data in Version 2.0 is copied to two other lists in Version 1.0. The message size increases dramatically when encoded in XML. The size overhead of XML depends not only on data but also on the *tags* used in describing XML data. This size expansion adds to significant network and processing overheads and tends to make it inappropriate for use as a wire format where performance is a concern [12].

Decoding cost with evolution: We measure the overhead of message evolution by decoding the PBIO-encoded *ChannelOpenResponse* message in ECho Version 2.0 to Version 1.0 type shown in Figure 4 using the transformation specified in Figure 5. This is compared with the cost of XML/XSL transformation of the corresponding versions of

[1]Marshalling small strings is not well supported, and is therefore considerably more expensive than prior results for PBIO (reported in [12]).

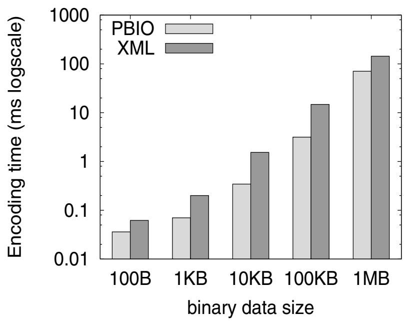

Figure 8. Encoding cost.

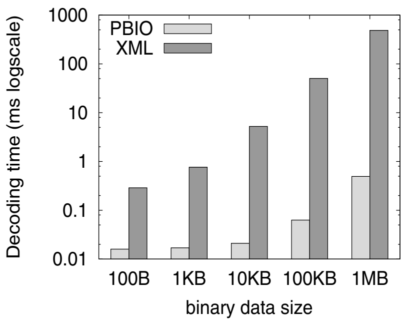

Figure 9. Decoding cost.

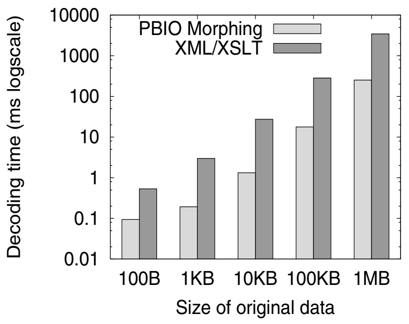

Figure 10. Decoding cost with msg evolution

the *ChannelOpenResponse* message. PBIO-based message morphing overhead has two main components: (i) cost of decoding the message to its native format (i.e. Version 2.0 *ChannelOpenResponse*), and (ii) cost of message transformation (from Version 2.0 to Version 1.0). The overhead of XML/XSL decoding consists of three components: (i) cost of preparing a XML parse-tree from the encoded message, (ii) cost of applying the XSL transformation and generating the new parse-tree, and (iii) cost of traversing the new-tree to form a data structure block of type *ChannelOpenResponse* Version 1.0. In each case, a pointer to the encoded message buffer was passed to the decoder routine and the cost was measured in the form of time spent in the decoder.

Figure 10 show the result of the above measurement. Time taken by XML/XSLT is an order of magnitude larger than that taken by PBIO-based message morphing. This is because the cost of parsing ascii-based XML and applying XSL transformation to it is much higher than binary-based message morphing.

It is interesting to compare these results with those in the earlier experiment (see Figure 9) where no format conversion is done. The time taken there is much smaller than in this case. The extra cost is mostly because of the cost of format translation, which depends on the nature of that transformation and the degree of mismatch between the two message types involved. The transformation (in Figure 5) used in this experiment involves scanning all of the input data before generating the final output. This is in fact a relatively expensive example of transformation, and we have chosen it deliberately to demonstrate the low overheads of message morphing. This cost would be even lower for evolution in real-world situations like the *brittle parameter problem* [17], where unnecessary details in the newer message versions prevent interoperability with the old client. Also, there are different coding optimization that we are currently doing in our message morphing library which will lower this cost further.

6 Conclusions and Future Work

This paper highlights the need for *message format evolution* in the context of large, long-running distributed applications, and it proposes Message Morphing to deal with such evolution. Message morphing allows snippets of "conversion code" to be associated with message formats, where each snippet specifies the transformations necessary to conform the incoming message into a format understood by the receiver. This conversion code can express more general transformations and provide more interoperability than XML-like data representation mechanisms, thereby offering a significantly wider range of message evolution possibilities. Also, because this code can be converted dynamically into a native conversion subroutine, this technique is appropriate even for high-volume, high-bandwidth data flows.

Experimental results show that message morphing is well-suited for high performance communication because of its use of an efficient binary data format, PBIO. Further, message morphing can support various kinds of evolution, but some transformations can still be costly. Developers must carefully design transformations that are low in cost but sufficiently general to enable desired evolutions.

In future work, we will evaluate the overheads of message morphing in the context of a large-scale application to augment the micro-measurements presented in this paper. We also hope that further experience in using message morphing in commercial applications will help us refine the *MaxMatch* algorithm. Our current approach works well for the examples we have considered so far, but more protocol evolution trials may show the utility of different feature sets, such as the ability to weight different fields and subfields based on some measure of "importance". Finally, we plan to use this technique in the larger context of our Service Morphing [25] work, where we meet application and end-user needs in the presence of run-time variation using dynamically-adapting services and dynamically-generated added functionality.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *18th annual ACM symposium on Principles of distributed computing.*, pages 53–61, May 1999.

[2] A. Berlea and H. Seidl. Transforming XML Documents using fxt. *Journal of Computing and Information Technology CIT, Special Issue on Domain-Specific Languages*, Jan 2001.

[3] A. Birrell, G. Nelson, S. Owicki, and T. Wobber. Network Objects. *Software Practice and Experience*, 25(S4):87–130, Dec 1995.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, August 2001.

[5] OMG. The Common Object Request Broker (CORBA): Architecture and Specification, 1999. Minor revision 2.3.1, OMG TC Document formal/99-10-07.

[6] Object Management Group. CORBAservices Specifications: Event Service., 2001. Rev. 1.1, OMG TC Document formal/2001-03-01.

[7] R. J. Cypser. Evolution of an Open Communications Architecture. *IBM Systems Journal*, 31(2):161–188, 1992.

[8] DCOM: Distibuted Component Object Model. `http://www.microsoft.com/com/tech/dcom.asp`.

[9] G. Eisenhauer. The echo event delivery system. Technical Report GIT-CC-99-08, College of Computing, Georgia Tech., 1999.

[10] G. Eisenhauer. Dynamic Code Generation with the E-Code Language. Technical Report GIT-CC-02-42, Georgia Tech, College of Computing, July 2002.

[11] G. Eisenhauer, F. Bustamante, and K. Schwan. Event Services for High Performance Computing. In *High Performance Distributed Computing (HPDC-9)*, pages 113–120, Aug 2000.

[12] G. Eisenhauer, F. Bustamante, and K. Schwan. Native Data Representations: An Efficient Wire Format for High Performance Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1234–1246, Dec 2002.

[13] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or, Why it's hard to build systems out of existing parts. In *17th International Conference on Software Engineering (ICSE-17)*, April 1995.

[14] IBM WebSphere MQ Buisness Integration Solution. `http://www.ibm.com/software/integration/wmq/`.

[15] C. Isert and K. Schwan. ACDS: Adapting Computational Data Streams for High Performance. In *Intl. Parallel and Distributed Processing Symposium*, May 2000.

[16] B. Krupczak, K. Calvert, and M. Ammar. Increasing the Portability and Re-usability of Protocol Code. *Trans. on Networking*, 5(4):445–459, Aug 1997.

[17] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-Oriented pattern matching for Evolvable Distributed Systems. In *18th ACM SIGPLAN conf. on Object-oriented programing, systems, languages, and applications (OOPSLA)*, Oct 2003.

[18] The XML C parser. `http://www.xmlsoft.org`.

[19] libxslt: The XSLT C library. `http://xmlsoft.org/XSLT/`.

[20] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. International Journal of Supercomputer Applications, 8(3/4):165 -414, 1994.

[21] H. F. Nielsen, P. J. Leach, and S. Lawrence. RFC 2774 - An HTTP Extension Framework. Feb 2000. `http://www.faqs.org/rfcs/rfc2774.html`.

[22] Oracle Corporation. Oracle$^{®}$ Application Developer's Guide - XML. 10g (9.0.4) Part Number B12099-01.

[23] B. Plale and K. Schwan. dQUOB: Managing Large Data Flows using Dynamic Embedded Queries. In *High Performance Distributed Computing (HPDC-9)*, pages 263–270, August 2000.

[24] C. Poellabauer, H. Abbasi, and K. Schwan. Cooperative Run-time Management of Adaptive Applications and Distributed Resources. In *10th ACM Multimedia Conference*, Dec 2002.

[25] C. Poellabauer et al. Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems. In *5th Int. Workshop on Active Middleware Services (AMS)*, June 2003.

[26] C. Poellabauer and K. Schwan. Power-Aware Video Decoding using Real-Time Event Handlers. In *5th International Workshop on Wireless Mobile Multimedia (WoWMoM)*, Sep 2002.

[27] Java Remote Method Invocation (Java RMI). `http://java.sun.com/products/jdk/rmi/`.

[28] M. Shaw. Architectural Issues in Software Reuse: It's not just the Functionality, It's the Packaging. In *Symposium on Software reusability*, April 1995.

[29] M. Spreitzer and A. Begel. More flexible data types. In *8th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 1999.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, August 2001.

[31] Sun Microsystems. Sun ONE Integration Server B2B Edition. `http://wwws.sun.com/software/products/integration_srvr_b2b/home_int_b2b%.html`.

[32] M. Wolf, Z. Cai, W. Huang, and K. Schwan. SmartPointers: Personalized Scientific Data Portals in your Hand. In *Proc. of ACM Supercomputing*, November 2002.

[33] Extensible Markup Language (XML). WWW Consortium: `http://www.w3.org/XML/`.

[34] Extensible Stylesheet Language Transformations (XSLT) Version 1.0. World Wide Web Consortium: `http://www.w3.org/TR/xslt`.

[35] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41– 53, January 2004.