Morphable Messaging: Efficient Support for Evolution 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

All but the most briefly used systems must evolve as their mission and roles change over time. Evolution in the context of large distributed systems is extraordinarily complex because of the difficulty of upgrading all components simultaneously, and the fact that such systems are often very sensitive to changes in the message formats that underlay their communication. Prior approaches to the problem of implementing changes in a deployed system have relied upon ad-hoc solutions or protocol negotiation to avoid message format mismatches. In this paper we present a novel approach that combines message meta-data and dynamic code generation to create a robust messaging system that naturally support application evolution.

Keywords
Message evolution, interoperability, distributed systems, dynamic code generation, transformation, PBIO

1. Introduction

Large scale distributed applications communicate among themselves through complex exchanges of messages, the structure and format of which is often known a priori when the application is developed. This prior information of messages can be exploited to effect very efficient communication because of low marshalling and unmarshalling overhead. But the needs and requirements of the distributed applications change over time. Independent developers implement new or similar functionalities using the same or different protocols and message formats. Heterogeneity and changes in hardware or system resources (like network bandwidth, system load) can require adequate change in message representation as well. This change in message format, to adapt to the new conditions or to provide new services, is called "*Message Evolution*". Message evolution in the context of large long-running distributed systems is inherently difficult because of the need to support interoperability between old and new clients and servers [20]. Emerging technologies like publish-subscribe systems [15, 2, 5] and peer-to-peer systems [21, 18, 24] make this problem even more difficult because there is no central control and no clear demarcation between the clients and the servers. Where possible, protocol designers take care to ensure that the message formats are backward compatible with the old ones but things become complicated when there are multiple designers making evolutionary changes at the same time and the problem of inter-operating between these new protocols becomes even more challenging. Interoperability is also important in the case of software reuse and legacy applications that may not be easy to replace [19, 12].

At the application and network services level, the problem is less tractable because of the diversity of application-level data structures and types. As such, application-level protocol evolution (or message evolution) is rarely addressed directly and is actively avoided wherever possible. Instead message formats and other elements of the protocol are frozen early in the development process, severely limiting the system's ability to evolve and serve changing needs. The problem is not entirely intractable, but known and emerging approaches [7] to it tend to be complex or impose overhead that is unsuitable for systems with high-bandwidth wide-area data flows.

For example, a common approach at the networking level is protocol version negotiation between peers and in a client/server situation. Negotiation to arrive at the best common protocol is a viable approach in small-scale situation, but it carries significant overhead on a per transaction basis and is impractical when information is

broadcast to many clients. Similarly, message representations that carry significant meta-data, such as XML and some object-based communication mechanisms, can be exploited to enable protocol evolution. However, including substantial meta-data in the communication has a significant impact on performance, making these systems inappropriate for high volumes of data. Even if the performance degradation is tolerable, the ability of these approaches to support protocol evolution is still limited. For example, XML-based systems using SAX or DOM interfaces can easily tolerate adding data or reordering fields in existing protocols without breaking old clients, but other restructuring, such as changing the structural association of the communicated data is impossible.

Typed invocation middleware architectures like OMG's CORBA [6], Microsoft's DCOM [8], Sun's Java RMI [17] and many others don't allow interactions between two different data types. But it makes sense to allow that interaction if such an interaction is semantically correct. For example, if the message from a new server contains an extra field that gives optional information, clients who don't understand or don't expect that field should still be able to handle the message. Disallowing such interactions only results in more rigid distributed systems.

In this paper, we introduce the concept of "*Message Morphing*" which combines message meta-data and dynamic code generation to conform to varying message formats and hence support *message evolution*. The basic idea is to associate a number of transformations with each new versions of the message so that they can be converted to other formats. If the receiver is not able to understand a particular message format, an appropriate transformation code associated with that format is downloaded and dynamically compiled to generate a conversion routine which then converts each incoming messages of this type to the one that the receiver can understand.

This technique has a number of advantages. First, as there is no negotiation, this can be easily applied to non client-server architectures and to components separated by space and time. Second, it not only guarantees syntactic compatibility but also addresses semantic compatibility. Third, there is no need to modify or restart an application because format conversion occurs on the fly and the application may be completely oblivious of the change. Third, it uses PBIO [9], a binary messaging system with out-of-band meta-information and very low marshalling overhead and is therefore suitable for use in high performance computing environment.

The remainder of the paper is organized as follows. The next section discusses about the various approaches that has been taken to addresses the problem of message evolution. Section 3 explains the algorithms and other techniques used in Message Morphing in great detail. In Section 4 we present an example scenario where we solve the interoperability problem using morphing. We evaluate the cost and overhead of PBIO and message morphing and compare it with that of XML in Section 5. We conclude in Section 6 and also present the future direction in this research.

2. Related work

There is a large body of work available in the literature which talks about the need for interoperability between components, network protocols and legacy applications. The main reasons for this interoperability are the assumptions that the two communicating parties make about each other [12]. Most of the current applications use various ad-hoc techniques and protocol negotiation mechanisms to decide upon a common language of communication. Shaw [19] discusses various possible alternatives to support interoperability between two components (say A and B), for example, rewriting A or B so that it complies with the other, publish A's abstractions (APIs, projections or views in databases), dynamic transformation to each other's forms, form negotiation, conversion plug-ins, intermediate representations (e.g. IDLs, postscripts), emulation, etc.

Object-based systems use sub-classing and other OO techniques to interoperate between different types. *Network Objects* [4], for example, uses 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 [6], which is another object-based system, supports interoperability through GIOP (General Inter-ORB Protocol) which describes the wire-representation of the messages exchanged between different ORBs (Object Request Brokers) and is implemented using IIOP (Internet Inter-ORB Protocol). In addition to static invocation (object types are known at run-time), CORBA also supports object invocation using runtime type information (usually retrieved from an *Interface Repository*).

The problem with the above systems is that they suffer from a deficiency that is inherent with the typed systems; namely, they don't handle a message that is semantically correct but not a subtype of the registered types. XML [22] supports some forms of evolution by allowing the use of

optional and ignorable items. It can probably be made more evolvable by the use of specially designed transformation languages like XSLT [23], fxt [3]. But XML-based approaches don't scale well in distributed environments because of high marshalling/unmarshalling overhead [11].

Spreitzer and Begel [20] observe that existing sub-typing techniques don't scale very well for decentralized evolution because of the additional latency incurred during the necessary ad-hoc negotiations between the client and the server. Also, the code size blows up exponentially by the number of extensions that a particular client understands. They proposed a flexible data type system which consists of extensible record types and coarse record types. The fields of extensible records are marked with a "*mode*" flag which indicates whether that field is optional or not. The coarse record types make sure that extensions are always compatible with sub-typing. Also, each field of a record value is marked whether it is ignorable or not. The receiver will be responsible for determining that it understood each piece of input and there is no need for negotiation.

Lee et. al. [13] uses object-oriented pattern matching to extract out relevant information from their self-describing messages. Applications specify handlers that need to be invoked when a particular pattern is matched. This technique allows optional fields to be included in a message. Developer don't have to write complex code to parse the messages. But text-based pattern-matching can be expensive and is not suitable for high performance systems.

Message Morphing also use XML-type mapping semantics based on field names. A field may be associated with a default value. It uses PBIO [9] to transfer data in heterogeneous environment. But the feature that is unique to our solution is the ability to dynamically convert formats using the transformations that are associated with each format type. This basically expands the *compatibility space*, as we will see in the next section.

3. Message Morphing Approach

3.1 Expanding Compatibility Space

The set of all message format types and/or protocol versions that an application can successfully interoperate with is called its *compatibility space*. If typical binary messaging is employed, the compatibility space is typically quite small. Any small change destroys compatibility. This is devastating for enterprise-scale application deployment, stifling change in already deployed applications. Text-based messaging formats with integrated meta-data (like XML) typically allow some changes (like field additions and rearrangement) to message data to occur without destroying compatibility. However complex changes still produce incompatibility. Message morphing is a technique to expand the compatibility space to a broader set of changes.

Our approach is based on some basic assumptions that enable the messaging system to implement message morphing. First, we assume that messages are associated in some way with meta-data that describes message content. In XML terms, this meta-data would be similar to a schema. Second, we assume that on the receiving side, the message system is aware of the set of message schemas or formats that the receiving application can process. In a complex distributed system, Application A may communicate with Application B using a set of messages 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 extra meta-data with Protocol Y messages so that they can be transformed, on-demand, into Protocol X messages (see Figure 1). This approach helps to remove fixed message formats as an obstacle in upgrading and evolving complex distributed systems.

In message morphing, most of the activity actually happens upon receipt of messages. Assume we have message M that 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 the 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 further below.

The technique is not without its limitations. There are many possible ways to change an application-level message pass-

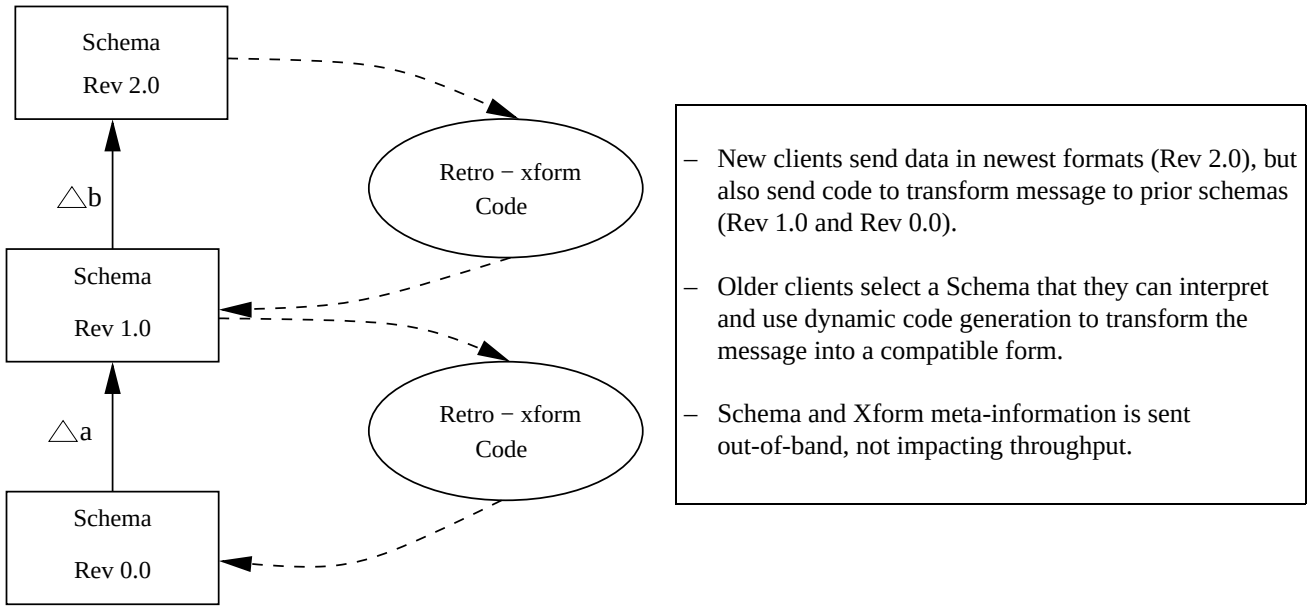

Figure 1. Retro - Transformation

ing 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 pretend to address all of those scenarios. Instead, message morphing should be viewed as another technique to be used in enlarging 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 interoperation in some situations where it would otherwise be impractical.

3.2 Implementation

In our system, messages are encoded using *Portable Binary Input/Output* (or PBIO) system [9]. 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) must provide similar information for the records that they are interested in reading. They also register handlers that need to be invoked when a message of a particular format type arrives. Figure 2 shows an example of a format declaration. Each format has a name associated with it, which may be 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).

On the reader's end, the format of the incoming record is compared with the reader's registered formats having the same name. If an exact match is found, the handler registered for that format type is invoked; otherwise 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 will be 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 is applied to all the 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*.

```
typedef struct {
        int avg_runqueue_length;
        int free_memory;
        int tx_rx_bytes;
} MonitoringMsg, *MonitoringMsgP;

IOField MonitoringMsg_field[] = {
        {"avg_runqueue_length", integer, sizeof(int),
                IOOffset(MonitoringMsgP,
                avg_runqueue_length)},
        {"free_memory", integer, sizeof(int),
                IOOffset(MonitoringMsgP,
                free_memory)},
        {"tx_rx_bytes", integer, sizeof(int),
                IOOffset(MonitoringMsgP, tx_rx_bytes)},
}
```

Figure 2. PBIO format declaration example

- 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 fields of 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:

1. diff(f_1, f_2) $<$ diff(f_1', f_2') or
2. diff(f_1, f_2) $=$ diff(f_1', f_2') and diff(f_2, f_1) $<$ 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, diff(f_1, f_2) is just 2. While another two formats (say f_1' and f_2') may have four uncommon

Algorithm 1 diff (f_1, f_2)

```
init: d12 = 0
for all field f in f1 do
  if f is of basic type then
    if f ∉ f2 then
      d12 ← d12 + 1
    end if
  else
    let f' be the complex field in f2 with the same field
    name and type as f
    if no such f' exists in f2 then
      increment d12 by Wf
    else
      d12 ← d12 + diff (f, f')
    end if
  end if
end for
return d12
```

fields (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.

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:

1. $f_1 \in F_1$,
2. $f_2 \in F_2$,
3. diff(f_1, f_2) $<=$ DIFF_THRESHOLD,
4. $M_r(f_1, f_2)$ $<=$ MISMATCH_THRESHOLD,
5. 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 to control 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.

Algorithm 2 Receiver-side message processing

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

if f_m seen previously **then**
  Use cached information to transform the message (if needed) and invoke the appropriate format handler
  return
end if
(f_1, f_2) ← 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
end if
(f'_1, f'_2) ← MaxMatch(F_t, F_r)
if there is no such (f'_1, f'_2) **then**
  Reject this message
  return
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 DCG
  Transform 'm' from its original format f_m to f'_1
end if
Put in the default values (if any) for the missing fields
Invoke handler for f'_2
return

Reader-side message processing:
The reader process the incoming encoded message in a number of steps which is outlined in the flowchart shown in Figure 3. The detailed algorithm is presented in Algorithm 2. There are a few things in the above algorithm that are worth mentioning. First, the expensive steps of the algorithm from 11-27 will be executed for only those formats which have not been seen previously by the reader. Once a format is seen, the transformation and the handler information is cached and will be used when messages of that format is received again. Second, it uses *MaxMatch* algorithm 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*

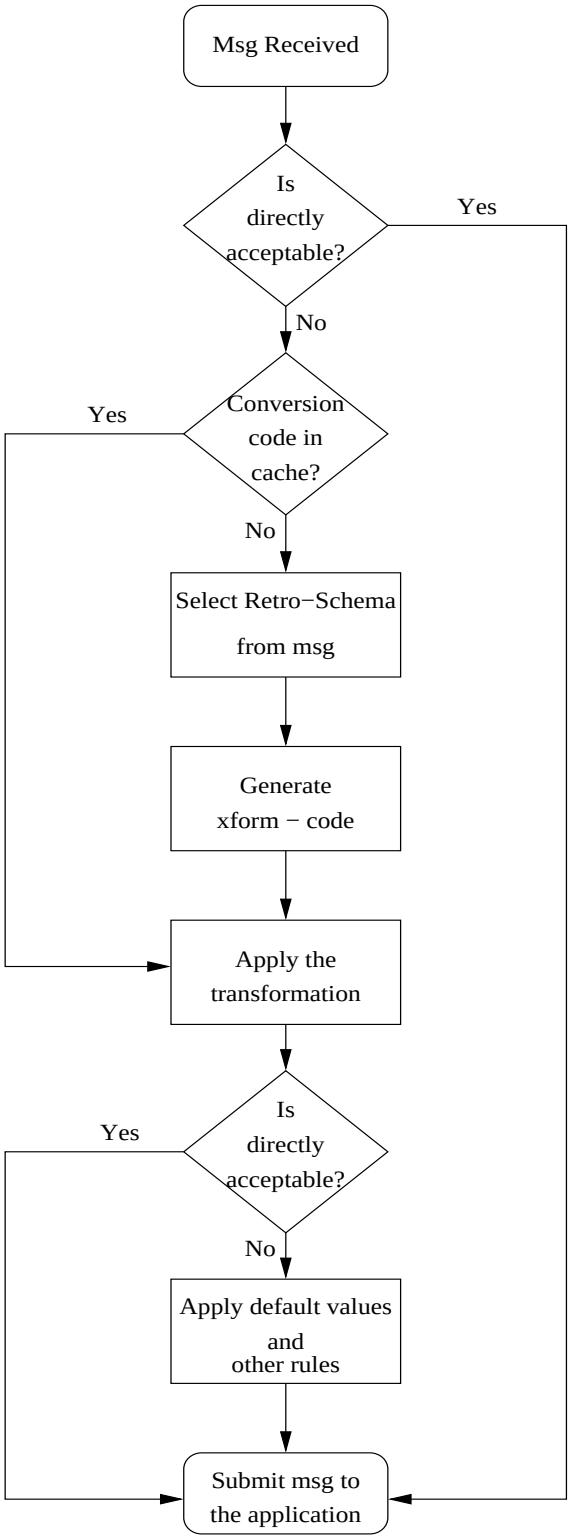

Figure 3. Message decoding flowchart

morphing is used to expand the *compatibility space*. New formats that were previously not understood by the reader get converted to match older, understood, formats, and hence come into the compatibility space of the reader. This leads to both backward and forward compatibility and allow for interoperability between both old server and new client and new server and old client.

Message morphing technique has many advantages, including:

- First, with the expansion of 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 the less efficient representation to ensure interoperability with all clients. This makes it easier to introduce new features and functionalities, servers become more scalable and enable more efficient use of resources (for e.g. network bandwidth).
- Independent specialization of system becomes easier.
- Complex systems become more malleable.

4. Example

To understand how message morphing works, we will work out a very simple example. Consider a *grid resource monitoring system* [1] in which each participating node exchanges their resource information with all other nodes in the grid by sending a message of format *MonitoringMsg* shown in Figure 4.

```
MonitoringMsg:
        int     avg_runqueue_length;
        int     free_memory;
        int     rx_tx_bytes;
```

Figure 4. Old message format

With the introduction of multi-CPU machines into the hardware mix, the grid designers felt the need for additional information in *MonitoringMsg* and came up with a new format version called *NewMonitoringMsg* which is shown in Figure 5.

There is no standard existing technique that will allow the clients of these two formats to interoperate although the

```
NewMonitoringMsg:
        int     number_of_cpus;
        int     *runqueue_length;
        int     total_memory;
        int     used_memory;
        int     rx_bytes;
        int     tx_bytes;
```

Figure 5. New message format

messages of the new format contains all the information which the messages of old format carried. One possible solution to the deployment of this new format is to stop the system and upgrade all existing clients simultaneously, but this is quite infeasible in environments like the grid. *Message Morphing* however, will allow this interoperability by the use of its transformation technique. When the old clients receive monitoring messages with the new format, the messages will be transformed into the old formats using the transformation shown in the form of *Ecode* in Figure 6 and the client need not be aware of it at all.

```
{
    int i, len = 0;
    for ( i = 0; i < new.number_of_cpus; i = i + 1)
        len += new.runqueue_length[i];
    old.avg_runqueue_length = len / new.number_of_cpus;
    old.free_memory = new.total_memory −
        new.used_memory;
    old.rx_tx_bytes = new.rx_bytes + new.tx_bytes;
}
```

Figure 6. Message transformation code

Thus new clients can take advantage of the extra information in the new format while still interoperating with the old ones. It should be noted that message morphing is not a magic bullet that can create compatibility in every situation. There are many possible ways to change an application-level message passing protocol that render it incompatible with unchanged clients, including changing the semantics of messages, removing previously-required information completely, etc. Our aim in message morphing is to enlarge the natural compatibility space in application-level protocols. PBIO provides XML-like compatibility without incurring the overheads of inline textual meta-data. Message morphing techniques extend that compatibility be-

yond simple structural similarity to cover situations where the new messages may be structurally dissimilar but still contain similar data in some form.

5. Evaluation

In this section, we compare the performance of PBIO-based message morphing technique with that of XML and show the feasibility of our method. XML was chosen because it is widely used form of communication and provide interoperability across heterogeneous environment as we do. It also supports some sort of message evolution in the form of plug-and-play flexibility as discussed earlier. The tests are carried on a 1266 MHz dual Pentium III machine with 1GB memory and running RedHat 7.3 linux distribution.

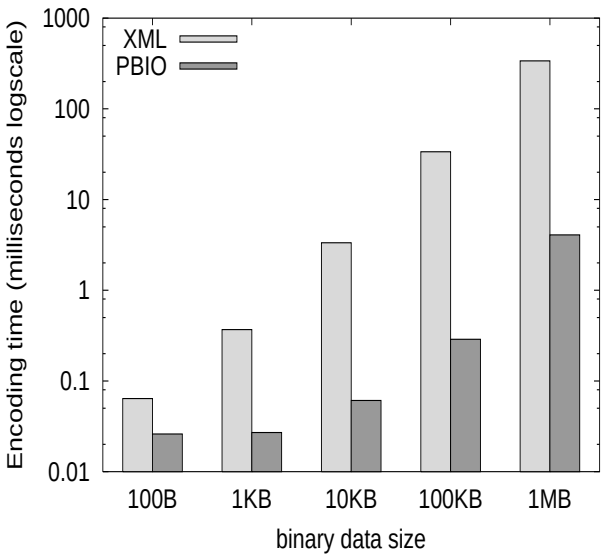

Figure 7. Message encoding cost with XML and PBIO.

Encoding Cost: Figure 7 shows the cost of encoding a message of type *NewMonitoringMsg* (shown in figure 5) for five different sizes (obtained by varying number_of_cpus field) 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 basically 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. The result is an encoding time for XML that is at least an order of magnitude higher than that of PBIO.

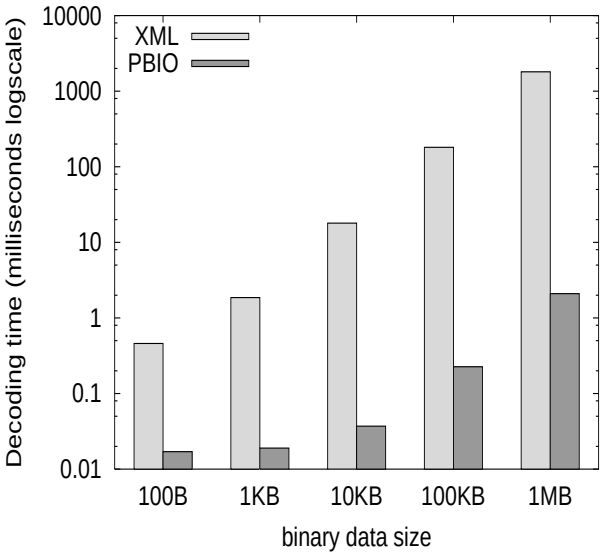

Figure 8. Decoding cost with XML and PBIO.

Decoding cost without evolution: The encoded message obtained from above is then decoded using their respective decoder. For parsing XML message, we use *libxml2* [14] version 2.6.4. Figure 8 show the results. It turns out that PBIO is far 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.

Decoding cost with evolution: We measure the overhead of message evolution by decoding the PBIO-encoded *NewMonitoringMsg* type to *MonitoringMsg* type using the transformation specified in Figure 6. The three different kind of bars in Figure 9 represent the following tests:

- The first bar (labeled "*XML*") is the cost of parsing a XML message without any transformation.
- The second bar (labeled "*PBIO: Homogeneous transfer*") shows the time required to decode a *NewMonitoringMsg* type PBIO-encoded message from the wire-format to its native-format in a homogeneous transfer.
- The last bar (labeled "*PBIO: Evolution*") indicates the cost of decoding the new format message to its native

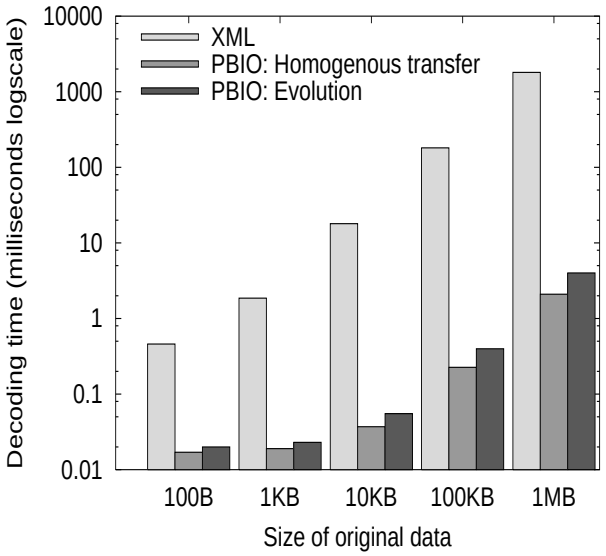

Figure 9. Decoding cost with message evolution.

type including time spent in transforming the message from the new format to the old format.[1]

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. It should be noted from this figure that the overhead of message morphing is quite small as compared to the costs of just receiving the message. The extra cost mostly depends on the nature of transformation and the degree of mismatch between the two message types involved. The transformation in Figure 6 takes O(n) time and involves scanning of the whole 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 overhead of morphing. This cost would be even lower for evolution in real-world situations like the *brittle parameter problem* [13], where unnecessary details in the newer message versions prevent interoperability with the old client. It is interesting to see that the overhead of XML parsing (which does not have the ability to support this kind of evolution) is far more than that of message morphing in all cases.

6. Conclusions and Future Work

In this paper, we highlighted the need of *message evolution* in the context of large long running distributed applications and proposed the concept of message morphing to deal with this evolution. Message morphing allows one or more snippets of "conversion code" to be associated with the message format which specify the transformations necessary to conform the incoming message into a format understood by the receiver. When receivers are presented with a message associated with a new protocol, they iterate through the set of supplied conversions to locate one that will make the message comprehensible. This conversion code can express much more general transformations and provide more interoperability than even XML-like data representation mechanisms, covering a significantly wider range of message evolution possibilities. Also, because this conversion code can be converted dynamically into a native conversion subroutine, this technique is appropriate even for high-volume, high-bandwidth data flows.

The evaluation results show that PBIO is well-suited for high performance communication because of its low encoding and decoding cost. Message evolution can be expensive as we saw in the last test. Although *message morphing* can support various kind of evolution, some transformations can be quite costly. With careful design, an application developer can specify transformations which are low in cost as well as facilitate evolution.

In future work we will evaluate the overheads of our message morphing technique in the context of a real-world application to augment the micro-measurements we have presented here. We also hope that further experience in using message morphing in real 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 features sets, such as giving different weight to different fields and subfields based on some measure of "importance". Finally, we wish to use this technique in the larger context of our Service Morphing [16] work in which we try to continuously meet application and end-users need in the presence of run-time variations by dynamically adapting the services as well as replacing old services with new ones to provided added functionalities. These kind of scenarios pose an interesting challenge for message evolution and make our work of *message morphing* all the more relevant.

[1] We have stretched this example somewhat to generate a wide range of data sizes. The 1Mb message represents a record from a machine with a few hundred thousand CPUs.

References

[1] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, , and M. Wolf. Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, Washington*, June 2003.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing (PODC), Atlanta, Georgia.*, pages 53–61, May 1999.

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

[4] A. Birrell, G. Nelson, S. Owicki, and T. Wobber. Network Objects. *Software Practice and Experience*, 25(S4):87–130, December 1995. Also appeared as SRC Research Report 115.

[5] 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.

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

[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. Portable Self-Describing Binary Data Streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(http://www.cc.gatech.edu/tech_reports)*.

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

[11] 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, December 2002.

[12] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering (ICSE-17)*, April 1995. A revised and extended version of this paper appeared in IEEE Software, Volume 12, Issue 6, Nov. 1995 (pp. 17-26).

[13] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-Oriented pattern matching for Evolvable Distributed Systems. In *ACM SIGPLAN Notices , Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA '03)*, volume 38, October 2003.

[14] The XML C parser. "http://www.xmlsoft.org/index.html".

[15] J. Pereira, F. Fabret, F. Llirbat, R. Preotiuc-Pietro, K. A. Ross, and D. Shasha. Publish/Subscribe on the Web at Extreme Speed. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt*, pages 627 – 630, September 2000.

[16] C. Poellabauer, K. Schwan, S. Agarwala, A. Gavrilovska, G. Eisenhauer, S. Pande, C. Pu, and M. Wolf. Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems. In *Proceedings of the 5th Annual International Workshop on Active Middleware Services (AMS)*, June 2003.

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

[18] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg.*, pages 329–350, November 2001.

[19] M. Shaw. Architectural Issues in Software Reuse: It's not just the Functionality, It's the Packaging. In *Proceedings of the 1995 Symposium on Software reusability*, volume 3, pages 3–6, April 1995.

[20] M. Spreitzer and A. Begel. More flexible data types. In *Proceedings of The Eighth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 1999.

[21] 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, Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, 31(4):149–160, August 2001.

[22] Extensible Markup Language (XML). World Wide Web Consortium: "http://www.w3.org/XML/".

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

[24] 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.