Extreme Markup Languages 2007®

Montréal, Québec
August 7-10, 2007

Using XML Compression to Increase Efficiency of P2P Messaging in JXTA-based Environments

Brian Demmings
Jodrey School of Computer Science, Acadia University, Wolfville, Canada

Tomasz Müldner
Jodrey School of Computer Science, Acadia University, Wolfville, Canada

Gregory Leighton
Department of Computer Science, University of Calgary, Calgary, Canada

Andrew Young
Jodrey School of Computer Science, Acadia University, Wolfville, Canada

Abstract

P2P [Peer-to-Peer] systems use messaging for communication amongst peers, and therefore the efficiency of messaging is a key concern for any P2P environment; particularly environments with a potentially large number of peers. One of popular representations of a P2P system is JXTA, which uses XML-based messaging. In this paper, we describe how the use of XML compression can increase efficiency of P2P messaging in JXTA-based environments. In the proposed solution, message elements containing XML data are compressed using an XML-aware compressor. Our design can be used not only for compression but also for other kinds of encodings of XML data, such as encryption. Experimental results demonstrate that our compression technique results in a substantial decrease in message transport time along with a corresponding decrease in the size of messages. Therefore, the application of XML compression for messaging in JXTA-based P2P environments results in an increase in the efficiency of messaging and a decrease in network traffic.

Using XML Compression to Increase Efficiency of P2P Messaging in JXTA-based Environments

Table of Contents

Using XML Compression to Increase Efficiency of P2P Messaging in JXTA-based Environments

Brian Demmings, Tomasz Müldner, Gregory Leighton, and Andrew Young

§ Introduction

As the power and affordability of general-purpose computers grew in the mid-1980s [Tanenbaum 2002], researchers increasingly looked for ways to improve the performance of complex computational tasks. The introduction of computer networking resulted in a new programming paradigm known as *distributed computing*, in which the performance of applications is improved by harnessing the computing power of multiple computers. However, working with distributed architectures has introduced challenges that directly affect the usability of the system. For instance, response time lags greater than 100 milliseconds in an application can impede a user's productivity [Seigneur 2003]. Decreasing system response time in a typical *client-server* system often involves increasing the server's computing power as the number of users grows, and the resulting monetary costs can be prohibitively high.

P2P technology addresses some of these concerns by moving the processing load of the application to the peer computers connected to the P2P network. The immediate benefit is an improvement of the application's ability to scale (with notable exceptions, e.g. early versions of the Gnutella protocol [Chawathe 2006]), since the presence of each additional user increases the effective computational power of the overall network. However, there is a cost associated with using P2P: as more computers join the network, increasing quantities of network bandwidth are required to support these computers [Sen 2004]. The increase is not solely due to an increase in inter-computer *signaling traffic*, but also the increased *content traffic*. Improved scalability allows a large number of users to access more content/ services than in a traditional system. One well-known and widely used P2P framework is called JXTA [JXTA][Sun JXTA], which was developed by Sun Microsystems, and then released to the open source community.

XML [Bray 2006] is a de-facto standard for data formats for various applications, including distributed applications such as JXTA. However, XML-encoded data is verbose, and when sent over networks, it can significantly increase traffic. As with other P2P systems, JXTA relies on peer messaging, using so-called pipes that allow both XML data and raw data. In this paper, we investigate the possibility of improving the performance of JXTA by *compressing* messages before sending them through pipes (and *decompressing* messages at the receiving end of the pipe). Our design supports three methods of creating these messages:

1. With no compression.
2. Using XML-aware compression.
3. Using standard data compression, such as `gzip`.

This selective approach allows applications to use knowledge of the structure of data contained in messages to improve overall compression, and consequently improve application response time. Note that our design can be also be used for other kinds of XML encodings, such as encryption.

For compressing the XML content of JXTA messages, we use two proven *XML-aware compressors*. The first such compressor, called TREECHOP [Leighton 2005][Leighton 2005a][Leighton 2005b] which is more effective at compressing XML than other XML-aware compressors, such as XGRIND [Tolani 2002], and has been shown to be efficient in WWW communications [Müldner 2005]. TREECHOP-compressed XML is query-able without decompression, allowing sections of the compressed XML to be located and extracted without reforming the XML document and *then* querying. Another advantage of TREECHOP is that it has been implemented in Java, and so it can be easily interfaced with JXTA-J2SE, which is also written in Java. The second compressor is XMLPPM [Cheney 2001], which is more space-efficient than TREECHOP, but it has been implemented in C++. Therefore, to interface this compressor with our code, we use Java Native Interface (JNI) [Sun 2003]. To compress non-XML data, we use `gzip` [Gailley 1991] as it is, on average, less space-efficient but more time-efficient than other XML compressors.

Any kind of data compression decreases the size of data at the expense of time to compress and decompress the data. We demonstrate, however, that in some cases the benefit of sending smaller (compressed) messages outweighs this overhead. Our experimental results support our hypothesis that applying compression technology results in a marked decrease in the average send-time for messages, a decrease in the size of messages, and consequently, a decrease in network traffic that improves the scalability of JXTA-based P2P applications.

This paper is organized as follows. Section "Related Work and Contributions" describes related work and summarizes our contributions. Section "JXTA Environment" provides a short overview of JXTA, and Section "Using Compressed Messages" provides a brief discussion of how a user can send (and receive) a compressed message. Section "Architecture" describes details of our architecture. Section "Testing" discusses our testing procedure and Section "Results and Interpretation" presents experimental results. Finally, Section "Conclusions" discusses our conclusions and future work.

§ Related Work and Contributions

Investigations of the performance of JXTA pipes have focused on measuring their speed in a variety of networking environments. Seigneur, et al. [Seigneur 2003] focused on identifying the overhead of sending messages of increasing size through pipes in loop-back pipe connections (referred to as "local" connections) and pipe connections over a Local Area Network (LAN). The authors measured the Round Trip Time (RTT) required for a message to travel from the sender to the receiver and back again. The RTTs showed that as the size of the message content increases linearly, so does the time required to send that message. Their experiments also quantify the performance loss experienced when using secure (encrypted) pipes, indicating a 100% - 300% increase in RTT time when using secure pipes compared to using insecure pipes.

Antoniu, et al. [Antoniu 2005] showed that JXTA pipes and JXTA Sockets could achieve performance levels similar to plain sockets on Fast Ethernet (100BT) networks. Their experiments used the so-called *bidirectional benchmark* to measure the time required to send a message and then receive a receipt acknowledgement from the receiver. Furthermore, JXTA-J2SE [CollabNet] Version 2.2.1 was found to provide a throughput of 145MB/s over Myrinet Version 2.0.11 [Boden 1995], and 101MB/s over Gigabit Ethernet. These findings suggest that the JXTA infrastructure is capable of high-performance message delivery.

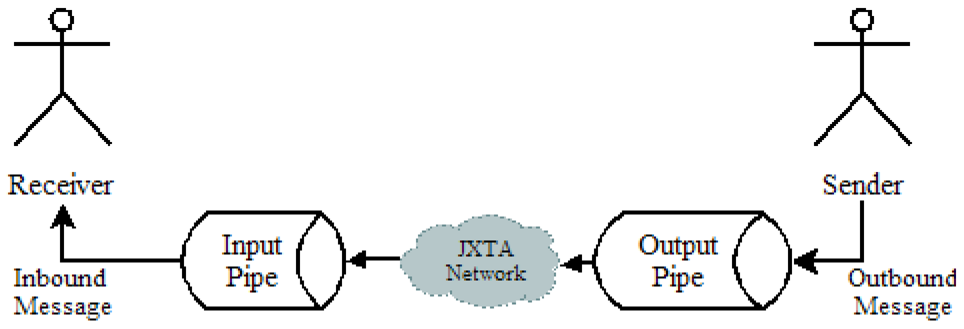
Contributions

We present the design, implementation, and test results of a technique that uses XML compression to increase efficiency of transporting JXTA messages. More specifically, we investigate the results of applying XML-aware and standard data compression techniques to XML data in JXTA messages. Unlike other research [Seigneur 2003][Antoniu 2005] which focused on the "out-of-the-box" performance of pipes, we focus on performance improvements that can be realized at the Application Layer. Specifically, we present an implementation that transparently applies compression to messages sent through an API called the CPA [Compression Pipe Adapter].

The internal compression and decompression processes of the CPA are transparent to the user. Using a CES [Compression Encoding Scheme], the adapter selectively compresses user-selected elements of JXTA messages before sending them through generic JXTA pipes. This design supports *selectivity*, which allows a user to specify those elements of a message that are to be compressed. To illustrate this technique, consider a system that may send large quantities of very small messages and occasional large messages. For very small data sizes, it is less likely that compression will decrease the size of the messages, since the overhead of using a compressor may increase the size of the data on very small data segments. For such an application, only the message elements with *large amounts* of data should be compressed. The principle of selectivity is consistent with the P2P philosophy of giving peers a lot of freedom.

While the focus of this paper is on the application of *compression* to JXTA messages, the design of the Encoding Scheme is flexible enough to allow applications to "plug in" virtually any type of "encoding-and-decoding" mechanism to transform the content of messages. This has immediate applications to domains such as encryption, allowing applications to selectively encrypt small segments of messages, avoiding the expensive [Seigneur 2003] use of JXTA secure pipes in cases where data privacy is the only requirement.

Figure 1: JXTA Environment Overview



Tests of the efficacy of our technique using TREECHOP show a slight 2.4% increase in message send-time for small (254 byte) XML content, but a pronounced 68.7% *decrease* in message send-time for larger (172491 byte) XML content. Similarly, for XMLPPM a 20.4% increase was observed for small message content but a 65.5% *decrease* was recorded for large content. Finally, `gzip` produced a 0.4 - 82.2% decrease in message send-time. For all but the smallest message content tested, our experiments also showed a decreased message content size of 15.0 - 89.4% for TREECHOP, 40.9 - 94.1% for XMLPPM and 24.8 - 88.6% for `gzip`, thereby reducing network traffic. Note that XMLPPM achieves a higher compression ratio than both TREECHOP and `gzip`, but is *much* slower.

Finally, our approach permits the selection of TREECHOP a query-able XML-conscious compression to aid in applications where query-ability is desired along with compression.

§ JXTA Environment

JXTA provides a decentralized, implementation-independent framework for the design, implementation and development of P2P applications. A full overview of the functionality of JXTA is beyond the scope of this paper; see [JXTA]. We focus on JXTA messages sent through pipes.

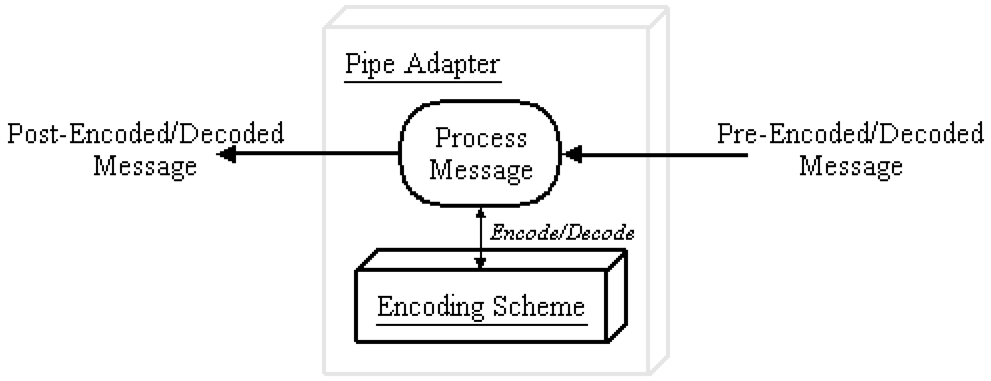
JXTA Messages

In JXTA, communication relies on messages to encapsulate data that a peer wishes to send to another peer. A message is similar to a SOAP envelope [Gudgin 2005]; it contains a payload, the *data* the user wishes to send, and a destination, referred to as *endpoint* [Duigou 2007]. A message consists of one or more *message elements* that store the message's data. There are two categories of content permitted in a message element: well-formed XML data and application-specific byte/text/string data. The latter type of data may be associated with a MIME Media Type [Freed 1996].

JXTA Pipes

To ensure that applications and users do not deal directly with network transports JXTA provides a connection-oriented abstraction known as the *pipe*. Each pipe is a connection between endpoints in JXTA, which may span networks that are *geographically* diverse (e.g. different physical networks) or networks that are *network layer* diverse (e.g. different network layers, such as TCP/IP and HTTP). The JXTA message protocol has provisions for either binary or XML messages [Duigou 2007], and so JXTA transports are categorized as being either *text-based* (e.g. SOAP and SMTP) or *binary* (e.g. TCP/IP Socket). Text-based transports use the XML message format, while binary transports rely on a predefined binary message format. JXTA pipes allow applications to send messages from one endpoint to one or more endpoints, and so they provide either *unicast* or *multicast* functionality. As shown in Figure 1, the user interacts with the underlying JXTA communications using the pipe, rather than creating a transport specific connection.

Figure 2: Pipe Adapter Architecture (High-Level).



There are two *directional* categorizations of pipes: *unidirectional* pipes transport messages from a sender to a receiver, and *bidirectional* pipes transport messages in both directions. Additionally, there are three *types* of JXTA pipes: *unicast*, *unicast-secure* and *propagate*. *Unicast* pipes are unidirectional, unreliable and insecure, and so they do not guarantee that the remote party will actually receive a message. *Unicast-secure* pipes are similar to unicast pipes, but are reliable and rely on Transport Layer Security (TLS) [Dierks 2006] to encrypt the content of messages [Gong 2002]. Finally, *propagate* pipes are unreliable and insecure, and are used to multicast messages to all peers connected to the pipe.

§ Architecture

In this section, we introduce the architecture that implements the goals of this paper. The primary component of this architecture is the *pipe adapter*, which provides an interface to a JXTA pipe allowing for transparent performance enhancements with minimal application changes. As illustrated in Figure 6, the sender and receiver rely on pipe adapters to send and receive encoded messages, in the same manner as peers use JXTA pipes. A high-level view of the pipe adapter is shown in Figure 2.

The detailed architecture of both directions of a *pipe adapter* is shown in Figure 3. The architecture of the Encoding Scheme component is discussed in the following section, and details of the implementation of the pipe adapter are discussed in Section "Pipe Adapter".

Encoding Scheme

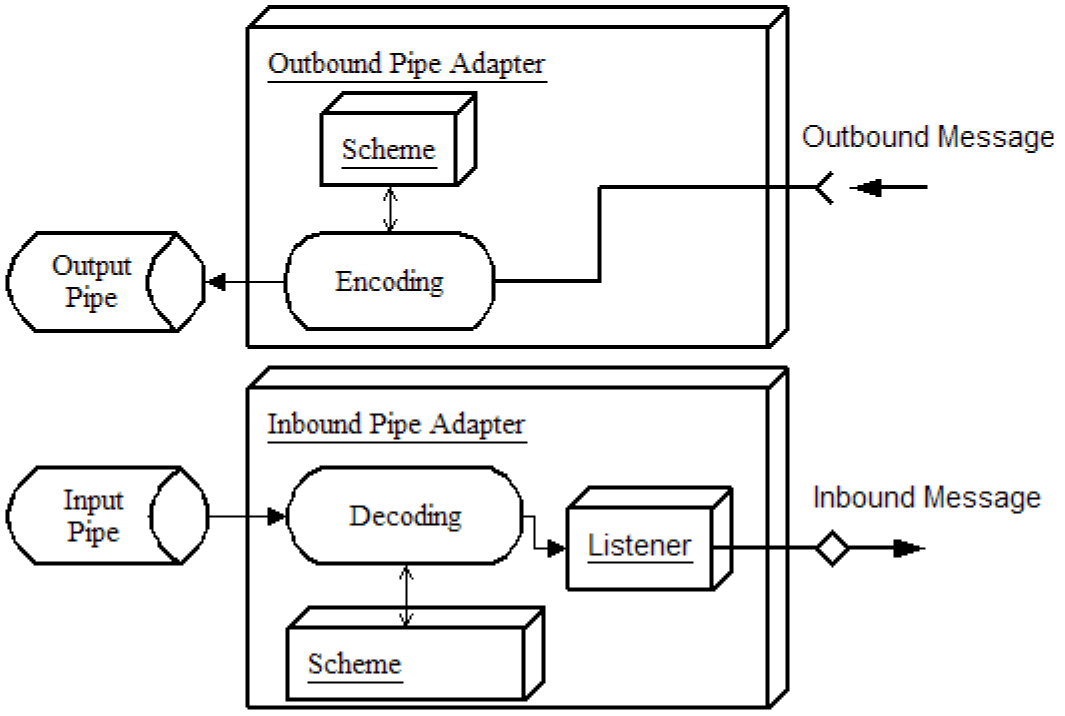
The *Encoding Scheme* component provides the application-specific encoding and decoding functionality for a *pipe adapter*. As discussed, we use this component to support compression, but its design is sufficiently general to support other encodings, e.g. XML encryption. An *Encoding Scheme* contains three embedded components: the *Encoder*, the *Decoder*, and the *Message Element Producer* component; see Figure 4.

The *Encoder* component is responsible for encoding the contents of message elements that have been produced by the MEP. The MEP is an implementation-specific component that allows an application to *select* those message elements that are to be encoded. The term *selected* denotes a message element that an application has decided to encode; the *Encoding Scheme* will encode *selected* message elements, leaving *unselected* message elements unaltered. For instance, consider an *Encoding Scheme* which has an attached MEP *M*. If an application creates a message element *E* and then creates a message element *F* using *M*, then the *Encoding Scheme* will encode *F* but not *E*.

Implementation

Figure 5 shows the overview of our implementation. Abstract class `EncodingScheme` has three methods; the `encode()` and `decode()` methods, and the `getElementProducer()` method. Applications use

Figure 3: Pipe Adapter Architecture (Detailed).



the first two methods to respectively encode and decode the content of JXTA message elements. The third method retrieves the MEP attached to the Encoding Scheme, and is described at the end of this section.

Figure 5 also shows the class `CompressionEncodingScheme`, CES, a concrete extension of the abstract `EncodingScheme` class, which provides selective compression of message elements in a message using either TREECHOP or XMLPPM for XML content, or `gzip` for generic data. When creating a CES, the user passes the `CompressionType` that the CES should use to compress XML content: TREECHOP, XMLPPM, or `gzip`; the application chooses the most efficient compressor for its data.

In the CES, the `encode()` function compresses the content of selected message elements, and adds a prefix to the name of those message elements; see Table 1.

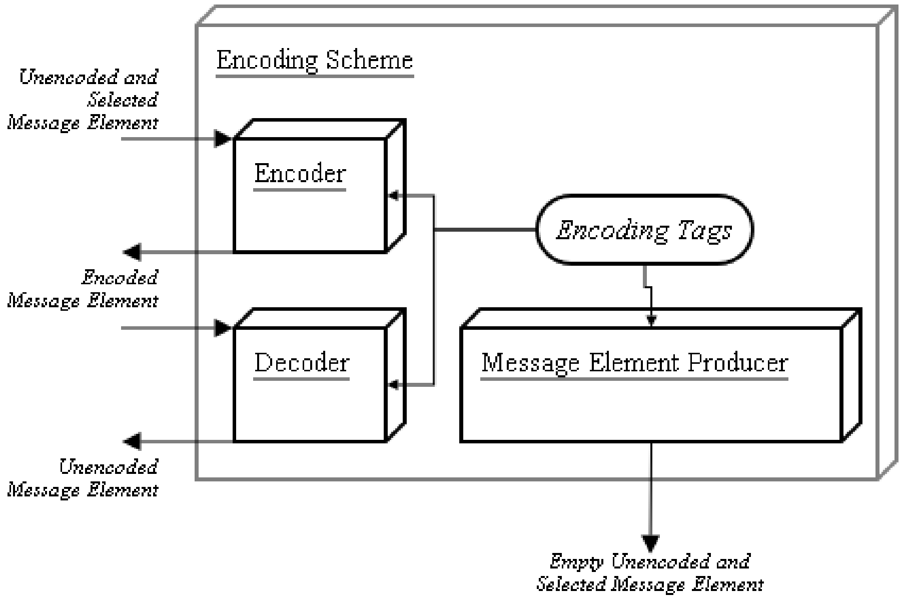
Table 1: CES: Message Prefixes

Compression Type	**Prefix**
TREECHOP	`__TREECHOPPEDMESSAGEEL__`
XMLPPM	`__XMLPPMMESSAGEEL__`
gzip	`__GZIPPEDMESSAGEEL__`

The `decode()` function decompresses the content of message elements whose name has been prefixed with a CES prefix, and removes the prefixes added by the encode process. It is unlikely that the increased length of the prefixed names will cause overflow issues since tests performed on the JXTA-J2SE 2.3.6 implementation showed that message element names may be greater than 100000 characters in length.

The CES has an attached CMEP [Compression Message Element Producer] that creates compressible message elements. The CES only compresses message elements produced with the CMEP, which implements the requirement of selectivity.

Figure 4: Encoding Scheme Architecture


Encoding Scheme
Unencoded and Selected Message Element
Encoder
Encoding Tags
Encoded Message Element
Decoder
Message Element Producer
Unencoded Message Element
Empty Unencoded and Selected Message Element

Pipe Adapter

Pipe adapters *encode* messages before sending them across a standard JXTA pipe, and *decode* encoded messages upon receipt from an inbound pipe (see Figure 6) using an *Encoding Scheme* which contains the “rules” used to perform the encoding (e.g. XML compressor/decompressor, Base64 Encoder/Decoder, etc.).

As illustrated in Figure 6, the user interacts with the pipe adapter rather than directly with the underlying pipe. Figure 7 illustrates this in terms of a communications “stack”.

The Encoding Scheme contained within a Pipe Adapter contains a MEP [Message Element Producer] which is used to create message elements (see Section “JXTA Messages”) that can be encoded and decoded using that Encoding Scheme. A given pipe adapter will only encode or decode messages that have been created with the MEP from the pipe adapter’s Encoding Scheme.

The architecture of a *pipe adapter* depends on the directionality of the flow of messages; a *pipe adapter* can be *Incoming*, *Outgoing* or *Bi-Directional*. Here we discuss the BiDi-PA [Bi-Directional pipe adapter] as it illustrates encoding of outbound messages and the decoding of those encoded messages on the receiving end; see Figure 8.

As with all *pipe adapters*, a BiDi-PA encapsulates an instance of an *Encoding Scheme*. Figure 8 illustrates the use of the Encoding Scheme in both inbound and outbound scenarios. When a sender sends a message through the BiDi-PA, the *pipe adapter* uses the *Encoding Scheme* during the encoding process to encode outbound messages. When a peer receives an encoded message, the BiDi-PA uses the *Encoding Scheme* to decode inbound messages. It is easy to see that by relying on the *Encoding Scheme* and a JXTA Bi-Directional pipe, a *pipe adapter* is able to facilitate encoded, bi-directional communications.

§ Using Compressed Messages

Compressed messages are created and sent with the aid of a CPA, which wraps a standard JXTA pipe. A CPA contains a CES which in turn provides a CMEP for use in creating compressed message elements.

Let us consider the example of a user wishing to create a *Message* with a compressed message element *A* and an uncompressed message element *B*. In this scenario, the sender uses the steps in Figure 9 to create a compressed message and send it.

Figure 5: Encoding Scheme Class Overview

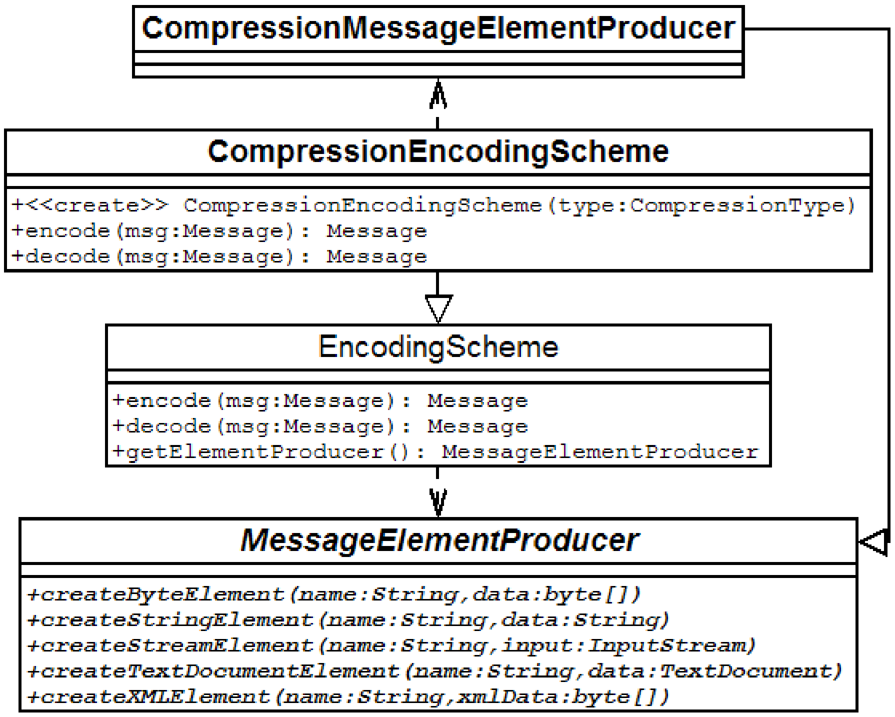
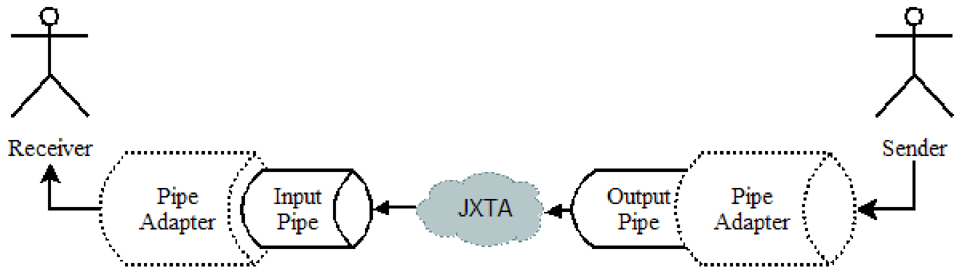

CompressionMessageElementProducer
CompressionEncodingScheme
+<<create>> CompressionEncodingScheme(type:CompressionType)
+encode(msg:Message): Message
+decode(msg:Message): Message
EncodingScheme
+encode(msg:Message): Message
+decode(msg:Message): Message
+getElementProducer(): MessageElementProducer
MessageElementProducer
+createByteElement(name:String,data:byte[])
+createStringElement(name:String,data:String)
+createStreamElement(name:String,input:InputStream)
+createTextDocumentElement(name:String,data:TextDocument)
+createXMLElement(name:String,xmlData:byte[])

Figure 6: Pipe adapter Overview



When the message M is sent, the message element A is detected by the CPA and compressed into A_c, then both A_c and B are sent as the content of M using the standard JXTA technique [Duigou 2007].

On the receiving side, A_c is uncompressed as A before the receiving user receives it, and B remains untouched. The receiving user receives M, and, since decompression occurred transparently, is unaware that A was compressed before transport.

Figure 7: CPA Communications “stack”

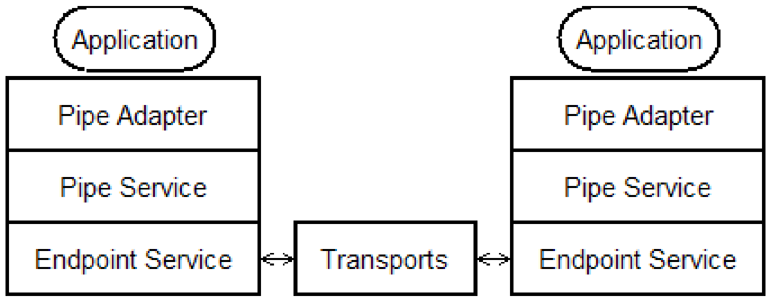
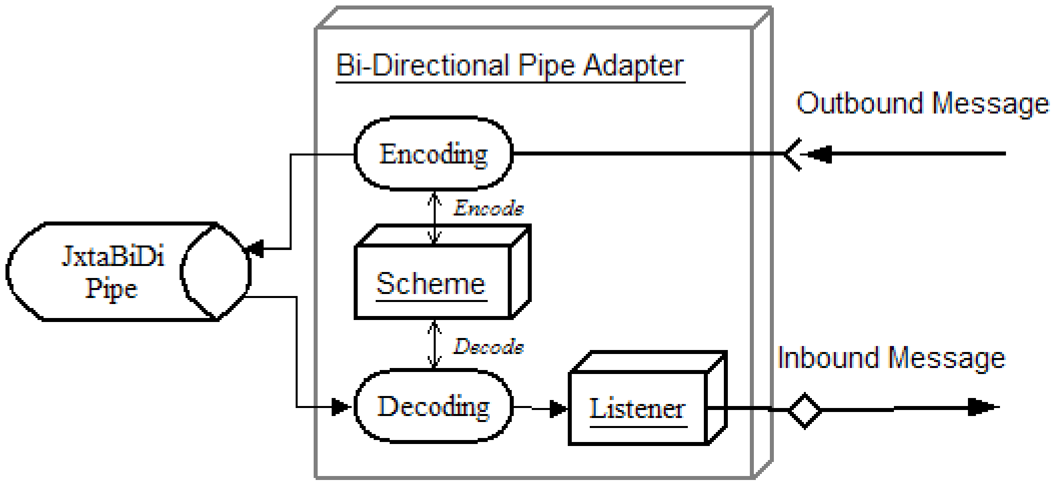


Figure 8: Bi-Directional Pipe Adapter



§ Testing

Procedure

In testing of the CBPA [Compression Bi-Directional Pipe Adapter], we used Dell D600 laptops with 512MB of RAM, a Pentium M processor running at 1.4 GHz [Dell], and Microsoft Windows XP®. We used JXTA Version 2.3.7 as the JXTA platform, running on Sun Microsystems’ Java [Gosling 1997] JDK Version 1.5.0 (build `1.5.0_06-b05`), which includes the native `gzip` compressor used in the CES.

For “local” testing, the computers were on the same 100BT Ethernet switch; for “remote” testing, the receiving computer was on a 100BT Ethernet switch connected to the Internet through a fire-walled ~25 MBps (~200 Mbps) connection, while the sending computer used a DSL connection with a maximum upload bandwidth of 640 Kbps.

Each *test* contains ten data sizes called trial *sets*. Each *trial set* contains 500 individual *trials* of the same data size, producing 5000 individual *trials* per *test*. The first 500 *trials* were discarded to ensure that the sender and receiver are in a *steady state* [Seigneur 2003]. Additionally, the highest and lowest outliers were discarded from each *set*, leaving 498 *trials*, per *trial set*.

Figure 9: Creating and Sending a Compressed Message

1. Create a CPA.
2. Access the CPA's attached CES.
3. Retrieve the CMEP from the CES.
4. Create a message element *A* using the CMEP.
5. Store an XML document in *A*.
6. Create a message element *B*.
7. Store an XML document in *B*.
8. User creates a JXTA `Message` object *M* and stores *A* and *B* in *M*.
9. The user then sends *M* through the CPA.

Figure 10: Example `employees` File.

```
<employees>
 <employee id="156201">
  <fname>Ryan</fname>
  <lname>McChomsky</lname>
  <age>36</age>
  <jobtitle>Cubical Farmer</jobtitle>
  <phone>977-2271</phone>
 </employee>
</employees>
```

To evaluate the performance of the CBPA, we used two JXTA Bi-Directional pipes. The first pipe, called the Communication pipe served two purposes. Firstly, this pipe transported *synchronization* messages used to ensure that both the sender and receiver of the test message were ready to process the trial, and was created using the standard bi-directional pipe instantiation procedure [Sun 2005]. Secondly, this pipe transported *uncompressed control* messages from the sender to the receiver, which provide a baseline send-time against which the send-time of compressed messages was compared. Control messages were sent *before* each compressed message. To encourage successful message exchange, the Communication pipe is *reliable*. The second pipe, called the Compressed pipe is fitted with a CBPA. *After* the control message is successfully sent to the receiver, the sender sends a compressible copy of the control message through the CBPA. A *single* test uses the same control and compressed pipes over its duration.

Trial Data Set

Each test set iterates over a set of seven pre-generated XML files that contain employee records for 1, 50, 100, 250, 500, 750 and 1000 fictitious employee records. Each employees file reaches a maximum depth of three from the root element to the deepest nested node within an employee record. Figure 10 shows an example employees document with one employee.

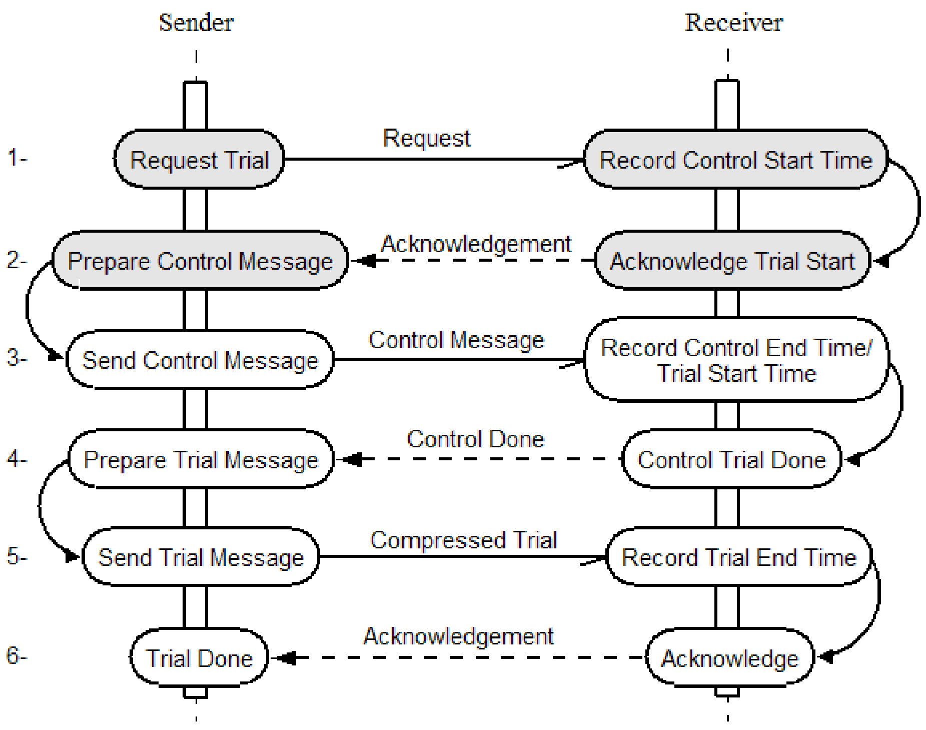
Note that while the individual employee records in each employees document cannot be said to be random (e.g. first and last names are reused, though not both at the same time), no two employee records are identical.

Test Protocol

To ensure consistency, each trial relies on the same protocol, illustrated in Figure 11. The protocol uses a handshaking [DOC 1996] process (grey activities in diagram) to ensure readiness of the sender and receiver. The sender requests a trial session with the receiver and the receiver records the receipt time with a 10 millisecond precision [Arnold 2005] using `System.currentTimeMillis()`.

Following the handshaking process, we begin *bidirectional benchmarking* (row 3 in Figure 11) to record the transit-time required for the *control* message compared to the transit-time of the *compressed* message. To avoid clock synchronization issues between the sender and receiver, bidirectional bandwidth

Figure 11: Trial Protocol



benchmarking uses a message-acknowledgement pair to measure message transit-times. The sender sends the *control* message containing the uncompressed payload message. When the receiver receives the *control* message, it records the receipt time of the *control* message and sends an acknowledgement. Next, the sender sends the *compressed* message. When the receiver receives the *compressed* message, it decompresses the compressed content, records the system time, and acknowledges the end of the trial. Note that we have concerned ourselves with measuring the time of sending a pre-serialized XML document and have not taken into account the time required for the XML serialization process. By the same token, we have not measured the time required to reparse the uncompressed XML content on the receiving side.

After each trial, the receiver compares the transit times for both the *control* and *compressed* messages, where the difference indicates the performance of the trial.

§ Results and Interpretation

Each *test* performed draws from files described in Section "Trial Data Set". Each consecutive *trial set* tests an XML document containing more employees. In this section each of the 498 trials of each trial set are averaged, and the average trial time is compared to the control time. The resulting value is the percentage increase or decrease in performance between the *control* and the *compressed trial*. Table 2, Table 3 and Table 4 show the sizes of the uncompressed test data compared to the size of the test data compressed with `gzip` and TREECHOP, XMLPPM, respectively. As expected, these tables indicate substantial space reductions when compression is applied to the employee data, resulting in between 24.8 - 88.6%, 15.0 - 89.4%, and 40.9 - 94.1% reduction in size compared to the original, for `gzip`, TREECHOP and XMLPPM, respectively.

Table 2: GZIP: Employee XML Compression Ratio.

XML Data Set	**Original Size (Bytes)**	**Compressed Size (Bytes)**	**% Reduction**
1	254	191	24.8%

XML Data Set	Original Size (Bytes)	Compressed Size (Bytes)	% Reduction
2	8720	1532	82.4%
3	17290	2591	85.0%
4	43208	5524	87.2%
5	86370	10253	88.1%
6	129384	14942	88.5%
7	172491	19589	88.6%

Table 3: TREECHOP: Employee XML Compression Ratio.

XML Data Set	Original Size (Bytes)	Compressed Size (Bytes)	% Reduction
1	254	216	15.0%
2	8720	1509	82.7%
3	17290	2510	85.5%
4	43208	5270	87.8%
5	86370	9662	88.8%
6	129384	13996	89.2%
7	172491	18322	89.4%

Table 4: XMLPPM: Employee XML Compression Ratio.

XML Data Set	Original Size(Bytes)	Compressed Size (Bytes)	% Reduction
1	254	150	40.9%
2	8720	1097	87.4%
3	17290	1688	90.2%
4	43208	3210	92.6%
5	86370	5588	93.5%
6	129384	7913	93.9%
7	172491	10198	94.1%

We recorded the absolute and percent decrease in AMTT [Average Message Transit-Time], (recorded in milliseconds) and compared it to the AMTT resulting when the same message is sent through a CPA which uses TREECHOP, `gzip` and XMLPPM to compress the same message. We performed two kinds of tests: (1) using a DSL connection and (2) using a Fast Ethernet LAN. We will first discuss the results from the DSL tests.

DSL

Experiments using DSL, shown in Tables 5, 6 and 7, determined that `gzip` resulted in 0.25 – 82.2% decrease in AMTT. Although both XML-aware compressors demonstrated an increase in the AMTT for the smallest message size (due to their increased compression time-overhead), for all other message sizes they achieved a marked decrease in AMTT of 31.46 – 68.67%, and 16.18 – 65.54% for TREECHOP and XMLPPM, respectively. These results very favorably support the hypothesis that compression can significantly decrease message send-times and, thus, increase application performance and decrease network traffic levels. Table 5 and Figure 12 illustrate the results of apply `gzip` compression during the send process while Table 6 and Figure 13 show the results obtained from using TREECHOP. Finally, Table 7 and Figure 14 illustrate the results obtained when using XMLPPM as a compressor.

Figure 12: `gzip` AMTT DSL

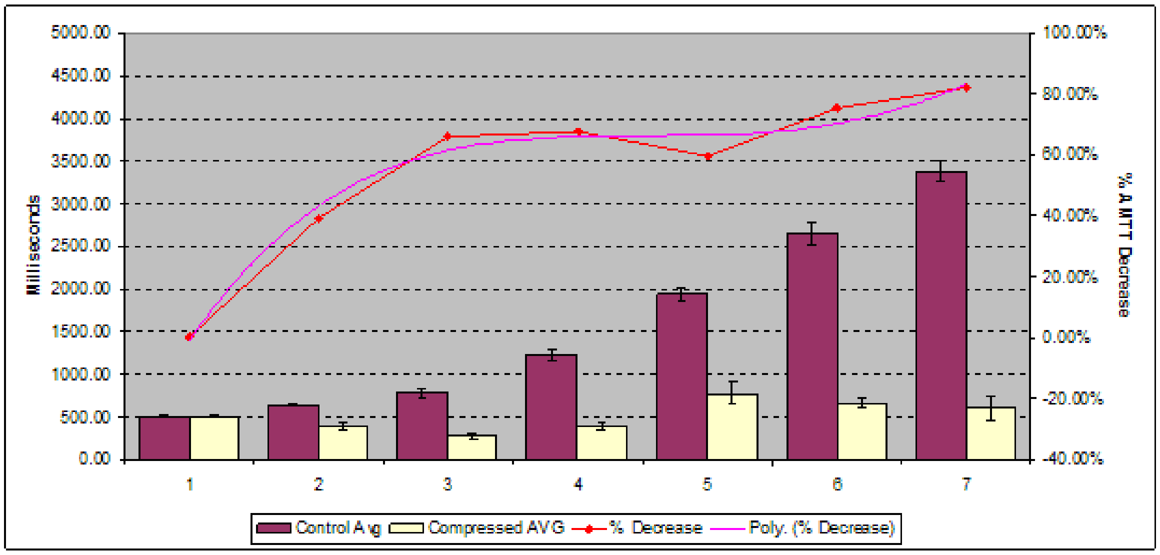


Table 5: `gzip` AMTT (ms) Results (DSL)

Set	Control (ms)	Compressed (ms)	Decrease (ms)	% Decrease
1	504.65	503.41	1.24	0.25%
2	643.73	390.03	253.69	39.41%
3	786.18	267.74	518.43	65.94%
4	1221.79	394.74	827.05	67.69%
5	1932.57	778.35	1154.23	59.72%
6	2647.72	653.04	1994.68	75.34%
7	3373.92	601.87	2772.05	82.16%

Table 6: TREECHOP AMTT (ms) Results (DSL)

Set	Control (ms)	Compressed (ms)	Decrease (ms)	% Decrease
1	504.79	516.91	-12.12	-2.40%
2	642.90	440.66	202.24	31.46%
3	784.69	347.05	437.64	55.77%
4	1214.18	547.09	667.08	54.94%
5	1927.39	1053.87	873.51	45.32%
6	2635.50	1025.99	1609.52	61.07%
7	3352.59	1050.36	2302.23	68.67%

Figure 13: TREECHOP AMTT DSL

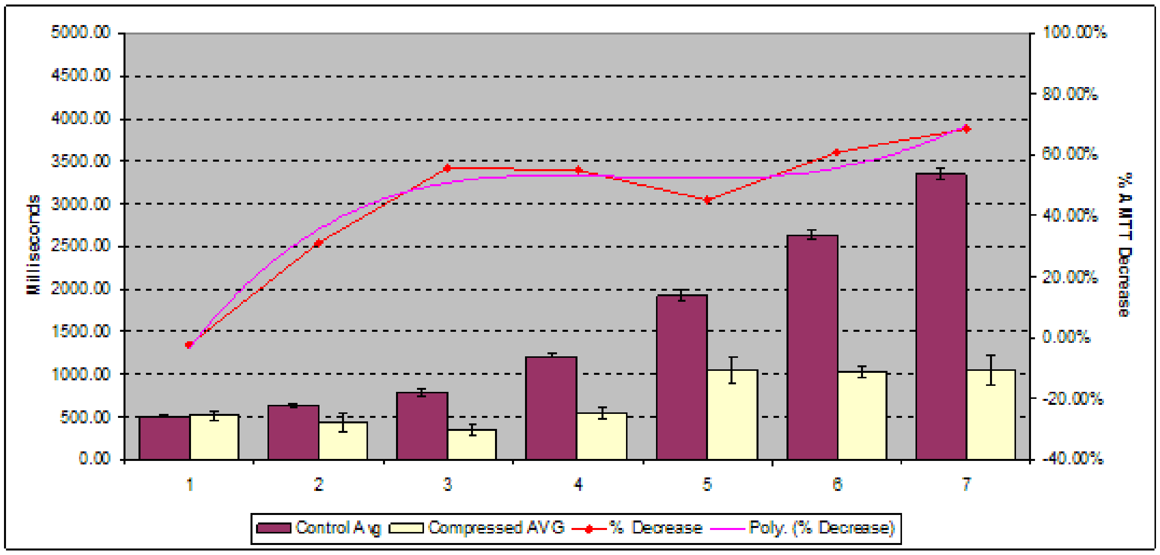


Table 7: XMLPPM AMTT (ms) Results (DSL)

Set	Control (ms)	Compressed (ms)	Decrease (ms)	% Decrease
1	506.70	609.73	-103.02	-20.33%
2	643.28	539.20	104.08	16.18%
3	789.34	455.48	333.87	42.30%
4	1225.70	639.18	586.52	47.85%
5	1947.50	1129.63	817.87	42.00%
6	2665.44	1207.68	1457.76	54.69%
7	3811.68	1313.58	2498.10	65.54%

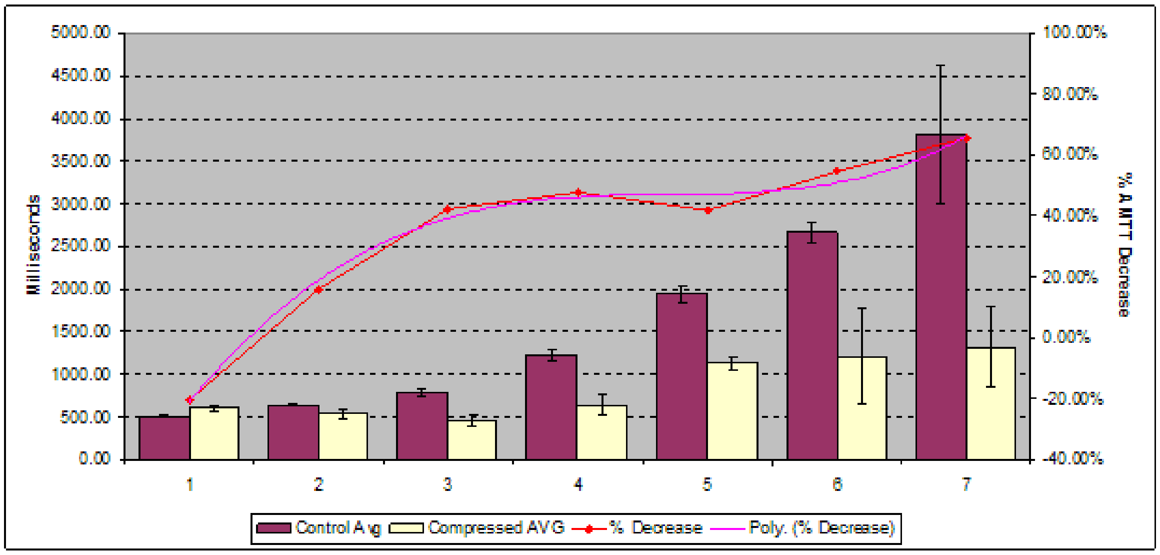
The resulting message transit-time reductions are summarized in Figure 15. These results indicate that because of its speed advantage `gzip` proved to be the most effective compressor in this application. XMLPPM resulted in the lowest transit-time decrease, despite its significant compression advantage in Table 4.

Analysis

The compression space-advantage of TREECHOP and XMLPPM is partially negated by the fact that, although they are more effective compressors, they both take longer to compress XML than `gzip`. This trend would continue until the compression time overhead becomes smaller than the time required to send the message over a slow connection. At that point, XML-aware compressors would achieve increasingly smaller compressed-AMTT values than `gzip`. XMLPPM resulted in a substantial compression advantage over TREECHOP; however, this advantage did not translate into smaller AMTT. Thus, the advantage of using XMLPPM over TREECHOP would only become apparent as network bandwidth decreased.

`Gzip` demonstrated an 18.24 – 110.26% lower send-time than TREECHOP and 25.36 – 143.59% lower send-time than XMLPPM. For many applications in low-traffic and high-bandwidth environments, `gzip` compression would be a prudent choice due to its compression speed, which translates to smaller send-times. In general, `gzip` would produce decreasing performance as the network became congested, or as overall bandwidth decreased.

Figure 14: XMLPPM AMTT DSL



The significant performance advantage exhibited by `gzip` was not investigated in-depth. However, there would seem to be two primary reason's leading to this advantage. Firstly, in in the Sun Microsystems' JDK, `gzip` is implemented using native-platform code rather than pure Java. Secondly, `gzip` does not incur the significant overhead of parsing the XML content of the messages. XMLPPM was implemented in C++ and executed as native code, but its architecture takes advantage of the structure of the XML document to achieve compression, so its performance is constrained. Similarly, TREECHOP is a pure-Java application which also takes advantage of the structure of XML.

To summarize, *all* DSL trials achieved a significant decrease in the AMTT for all but the smallest message sizes. `Gzip` compression achieved the largest decrease, however, in applications where *query-ability* is a required feature, TREECHOP would be the logical choice since it provides a greater AMTT reduction than XMLPPM and its compressed format is query-able.

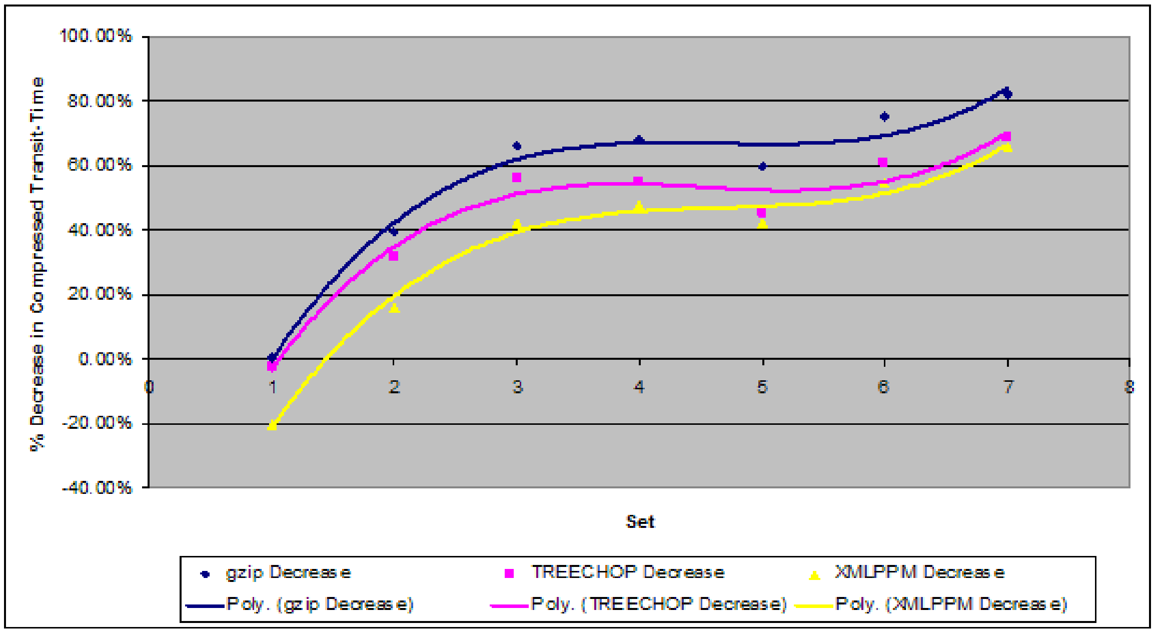
Ethernet

The results for the Fast Ethernet LAN show that the send-time *increased* between 0.8% and 3.58% for `gzip`, 2.02% and 72.08% for TREECHOP and 21.47% and 135.70% for XMLPPM, depending on the size of data. These results were expected, because of the speed and low traffic conditions of this network. The CBPA *did* achieve a marked reduction in the message size, therefore, as the traffic on the LAN increases, the use of the CBPA would result in a higher *message throughput* at high traffic levels; a hypothesis partially confirmed by the results obtained from the DSL connection tests. These results have implications on the scalability of applications in a high-traffic LAN environment, which is an important consideration.

As one would expect, the speed and connection quality of a 100BT Ethernet connection greatly reduces the efficacy of compressing XML data before sending. Therefore, our results show an all-around worsening of transit times, as depicted in Figure 16. The main saving-grace of these results is that as network congestion increased on the Ethernet network, the efficacy of using a CPA would increase in a similar manner.

In Figure 16 *negative decrease* values indicate a *increase* in send time. For example, a -140.0% decrease indicates that the message send time *increased* by 140%.

Figure 15: Comparison of DSL AMTT % Decreased Transit Time



§ Conclusions

This paper introduced a method for selective compression of JXTA messages. The compression and decompression processes are transparent to the client, who simply needs to create message elements that are to be compressed using the CMEP attached to the CES. Therefore, from the user perspective there is very little change to the messaging paradigm, which improves the ability of designers to integrate this paradigm with pre-existing and future applications.

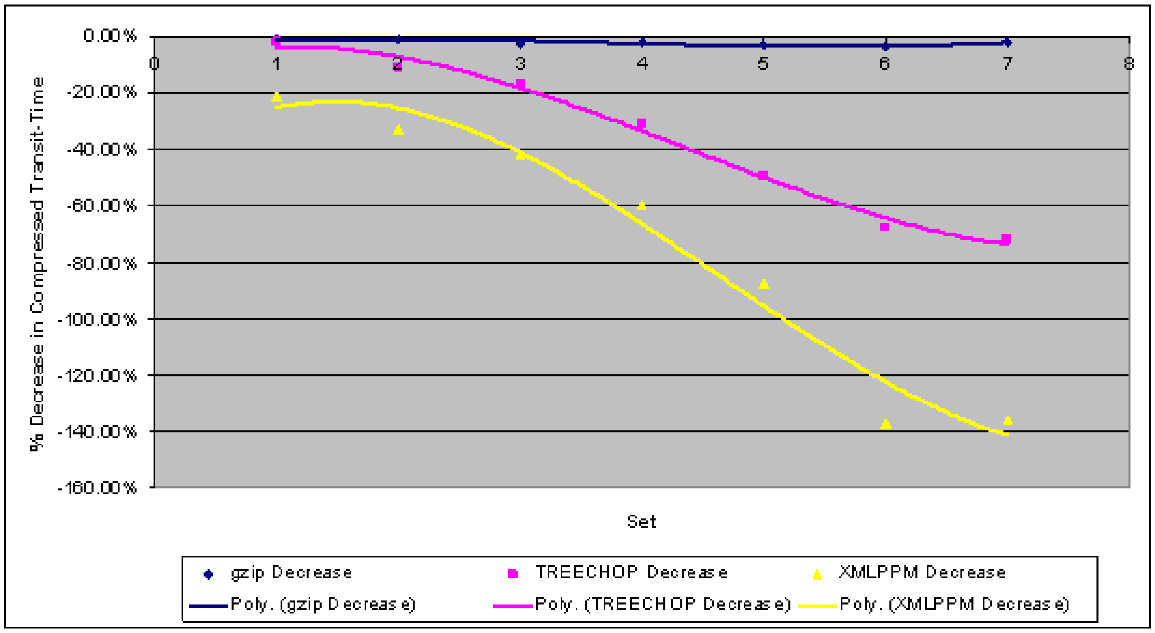
Tests of the efficacy of the CBPA on a DSL connection with an upload bandwidth of 640 Kbps indicate a -2.40 – 68.67% decrease in message send-time for TREECHOP, -20.33 – 65.54% decrease for XMLPPM and a 0.25 – 82.6% decrease for `gzip`. Our experiments also showed the decreased size of a message's data, by between 24.8 – 88.6%, 15.0 – 89.4% and 40.9 – 94.1% for `gzip`, TREECHOP and XMLPPM, respectively.

It should be noted that although `gzip` performed better in these tests, TREECHOP does offer the significant advantage of query-ability of the compressed XML, a feature that `gzip` does not allow. For applications relying on this capability, the logical choice would be to use TREECHOP. In applications where peers are connected to an Ethernet network, the observed decrease in message size using a CBPA would reduce traffic significantly, allowing the application to scale more effectively. In general, compression would improve the overall performance of the distributed application because high-bandwidth connections are not necessarily prevalent in an application with geographically disperse JXTA peers. Thus, the results support our hypothesis that XML data compression techniques can increase the performance of sending messages over JXTA pipes.

Future work into this area would investigate the efficacy of compression over HTTP connections, firewall traversal, relay peers and possibly modem connections. This would result in a more complete understanding as to message-size above which the CPA provides gains.

Additionally, the proposed architecture is ideally suited for applications that use encryption; the implementation allows selective encryption of the contents of messages and so the overhead of using a secure pipe as illustrated in [Seigneur 2003] could be offset by only encrypting the select data in messages. Finally, since compression decreases network traffic, the effect of compression on message delivery failure-rates over non-reliable pipes would be an important area of study.

Figure 16: Comparison of LAN AMTT % Decreased Transit Time



Bibliography

[Antoniu 2005] Antoniu, G., Hatcher, P., Jan, M., and Noblet, D. A. 2005. Performance Evaluation of JXTA Communication Layers. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05)* 1, 251-258.

[Arnold 2005] Arnold, M. and Grove, D. 2005. Collecting and exploiting high accuracy call graph profiles in virtual machines. In *Proceedings of the 2005 International Symposium on Code Generation and Optimization (CGO '05)* 00, 51 - 62.

[Boden 1995] Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., and Su, W.-K. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro 15*, 1, 29 - 36.

[Bray 2006] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. 2006. Extensible markup language (xml) 1.0 (fourth edition). W3c recommendation, World Wide Web Consortium (W3C). September. Accessed: November 4, 2006.

[Chawathe 2006] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. 2003. Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, New York, NY, USA, 407-418.

[Cheney 2001] Cheney, J. 2001. Compressing xml with multiplexed hierarchical ppm models. In *Proceedings of the Data Compression Conference (DCC '01)*, 0163.

[CollabNet] CollabNet. 2006. JXTA Platform Project. Accessed: April 16, 2007 from http://www.jxta.org.

[Dell] Dell. Technical specifications: Dell latitude c600/c500 user's guide. Online: http://support.dell.com/support/edocs/systems/latc600/en/ug/specs.htm. Accessed: April 15, 2007.

[Dierks 2006] Dierks, T. and Rescorla, E. 2006. The transport layer security (tls) protocol: Version 1.1. Standards Track RFC4346, The Internet Engineering Task Force. April. Accessed: April 15, 2007.

[DOC 1996] National Communications System, T. and Division, S. 1996. TELECOMMUNICATIONS: GLOSSARY OF TELECOMMUNICATION TERMS, Federal Standard 1037-C ed. General Services Administration, Information Technology Service, National Telecommunications and Information Administration, U.S. Department of Commerce, Boulder Colorado. Accessed: April 15, 2007 from http://glossary.its.bldrdoc.gov/fs-1037/.

[Duigou 2007] Duigou, M. and Dengler, T. 2007. JXTA v2.0 Protocols Specification. Sun Microsystems, Inc. and The Internet Society, https://jxta-spec.dev.java.net/JXTAProtocols.pdf. Accessed: July 05, 2007.

[Freed 1996] Freed, N., Borenstein, N., Moore, K., Klensin, J., and Postel, J. 1996. Multipurpose internet mail extensions (mime) (parts 1-5). DRAFT-STANDARD 2045,2046,2047,2048,2049, The Internet Engineering Task Force: Network Working Group, http://tools.ietf.org/html/rfc2045,rfc2046,rfc2047,rfc2048,rfc2049. November.

[Gailley 1991] Gailley, J.-L. and Adler, M. 1991. `gzip`. *The `gzip` Homepage*, Accessed: April 16, 2007.

[Gong 2002] Gong, L. 2002. Project JXTA: A technology overview. Tech. Rep. 01nov02, Sun Microsystems, Inc., http://www.jxta.org/. October. Accessed: April 15, 2007.

[Gosling 1997] Gosling, J. and McGilton, H. 1997. The Java language environment. White paper, Sun Microsystems, Inc., http://java.sun.com/docs/white/langenv/.

[Gudgin 2005] Gudgin, M., Mendelsohn, N., Nottingham, M., and Ruellan, H. 2005. Soap message transmission optimization mechanism. W3C Recommendation REC-soap12-mtom-20050125,World Wide Web Consortium (W3C), http://www.w3.org/TR/soap12-mtom/. January.

[JXTA] CollabNet. Project JXTA. Accessed: April 16, 2007.

[Leighton 2005] Leighton, G. 2005. Two New Approaches for Compressing XML. M.Sc. thesis, Jodrey School of Computer Science, Acadia University.

[Leighton 2005a] Leighton, G., Müldner, T., and Diamond, J. 2005a. Treechop: a tree-based query-able compressor for xml. In *Proceedings of the Ninth Canadian Workshop on Information Theory (CWIT 2005)*. 115-118.

[Leighton 2005b] Leighton, G., Müldner, T., and Diamond, J. 2005b. Treechop: A tree-based query-able compressor for xml. Tech. Rep. TR-2005-005, Acadia University, Jodrey School of Computer Science, Jodrey School Of Computer Science Acadia University Wolfville, NS B4P 2R6 - Canada. Aug. 2005.

[Müldner 2005] Müldner, T., Leighton, G., and Diamond, J. 2005. Using xml compression for www communication. In *Proceedings of the International Association for Developement of the Information Society (IADIS) International Conference WWW/Internet 2005(ICWI 2005)*. IADIS, Lisbon, Portugal, 459-466.

[Seigneur 2003] Seigneur, J.-M., Biegel, G., and Jensen, C. D. 2003. P2P with JXTA-Java pipes. In *PPPJ '03: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*. Computer Science Press, Inc., New York, NY, USA, 207-212.

[Sen 2004] Sen, S. and Wang, J. 2004. Analyzing Peer-to-Peer Traffic Across Large Networks. *IEEE/ACM Transactions on Networking 12*, 2, 219-232.

[Sun 2003] Sun. 2003. Java Native Interface Specification. Specification 6.0, Sun Microsystems, Inc. Accessed: April 16, 2007 from http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html.

[Sun 2005] Sun Microsystems, Inc. 2005. JXTA v2.3.x: Java Programmers Guide, 2.3.x ed. Sun Microsystems, Inc. Accessed: April 16, 2007 from http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf.

[Sun JXTA] Sun Microsystems, Inc. JXTA Technology Sun Microsystems, Inc. Accessed: April 16, 2007 from http://www.sun.com/software/jxta/

[Tanenbaum 2002] Tanenbaum, A. S. and van Steen, M. 2002. *Distributed Systems: Principles and Paradigms*, 1 ed. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, Chapter 1, 2-3.

[Tolani 2002] Tolani, P. M. and Haritsa, J. R. 2002. XGRIND: A Query-Friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 02)*. ICDE 00, 0225.

The Authors

Brian Demmings
Jodrey School of Computer Science, Acadia University, Wolfville, Canada
034776d@acadiau.ca

Brian Demmings is a Masters in Science in Computer Science candidate at Acadia University in Wolfville, Nova Scotia. His current area of research are filtering compressed XML and storage systems, however, his interests extend into XML Compression and P2P-based systems.

Tomasz Müldner
Jodrey School of Computer Science, Acadia University, Wolfville, Canada
tomasz.muldner@acadiau.ca

Tomasz Müldner is a professor of computer science at Acadia University in Nova Scotia, one of Canada's top undergraduate universities. He has received numerous teaching awards, including the prestigious Acadia University Alumni Excellence in Teaching Award in 1996. He is the author of over seventy papers and four books, including 'C++ Programming with Design Patterns Revealed" and 'C for Java Programmers". Dr. Müldner received his Ph.D. in mathematics from the Polish Academy of Science in Warsaw, Poland in 1975. His current research includes XML compression and encryption, P2P systems and algorithm explanation.

Gregory Leighton
Department of Computer Science, University of Calgary, Calgary, Canada
gleighto@cpsc.ucalgary.ca

Gregory Leighton is a Ph.D. student at the University of Calgary. His current research interests include XML data management and the Semantic Web.

Andrew Young
Jodrey School of Computer Science, Acadia University, Wolfville, Canada
056061y@acadiau.ca

Andrew Young was a form Bachelor of Computer Science with Honours student at Acadia University. Having graduated in 2006, Andrew is currently working in industry.

Extreme Markup Languages 2007®
Montréal, Québec, August 7-10, 2007
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.