Improving Network Efficiency in Real-Time Groupware with General Message Compression

Carl Gutwin, Christopher Fedak, Mark Watson, Jeff Dyck, and Tim Bell*

Computer Science Department
University of Saskatchewan
110 Science Place, Saskatoon, Canada

*Computer Science Department
University of Canterbury
Private Bag 4800, Christchurch, New Zealand

carl.gutwin, chris.fedak, mark.watson, jeff.dyck @usask.ca; tim.bell@canterbury.ac.nz

ABSTRACT
Groupware communicates by sending messages across the network, and groupware programmers use a variety of formats for these messages, such as XML, plain text, or serialized objects. Although these formats have many advantages, they are often so verbose that they overload the system's network resources. Groupware programmers could improve efficiency by using more compact formats, but this efficiency comes at the cost of increased complexity, reduced convenience, and reduced readability. In this paper we propose an alternate approach for improving efficiency – an automatic compression system that transparently minimizes verbose formats. Our general message compressor – GMC – automatically finds and removes redundancy in message streams, without any knowledge of the contents or structure of the message, and without any need for the programmer to change the way they work. In tests with realistic message traces, GMC reduced text messages to 20% of their original size, XML messages to 8% of the original, and serialized objects to 9%. Although not as compact as a hand-coded representation, GMC provides most of the compression benefits with almost none of the work – it allows groupware programmers to use convenient message formats without compromising transport efficiency.

Categories and Subject Descriptors
H.5.3 [Group and Organization Interfaces]: *Computer-supported cooperative work*; E.4 [Coding and Information Theory]: *Data compaction and compression*.

General Terms
Algorithms, Design, Experimentation, Human Factors.

Keywords
Groupware performance, message compression, network delay.

1. INTRODUCTION
Groupware systems share information – such as data updates, remote commands, lock requests, or user events – by sending messages. Groupware programmers use a variety of formats for these messages, such as XML, plain text, or serialized objects. Each format has advantages: some are easy to construct (e.g., serialized objects), others fit well with existing tools (e.g., XML messages with XML parsers), and others are easy for humans to read (e.g., plain text). The drawback to all of these message formats, however, is that they are verbose – that is, they use a large amount of space to represent a small amount of information. For example, the text telepointer message in Figure 1 uses 85 bytes to send a timestamp, a client ID and a new pointer location. Serialized objects are even worse – a simple telepointer event object can require 267 bytes in serialized form.

```
(timestamp: 803488132 sender_id: 12 session_id:
9357)(telepointer x: 1138.0 y: 601.0)
```

Figure 1. A text telepointer message

Inefficient message representations are a major problem because messages must be sent across the network, and current groupware systems regularly run out of network bandwidth. If a system tries to send more data than it has bandwidth, messages will pile up, and latency will increase to the point where many functions of the system (such as telepointers, locks, and shared data structures) are unusable. There are several situations that can result in low available bandwidth for a groupware system: for example, low-data-rate networks (wireless or dialup), high-bandwidth networks that already have a large amount of traffic, or broadband connections with asymmetric upload/download rates. In these situations, groupware can exceed bandwidth with its own messages, choking its own communication channel.

Even though network capacities are increasing in general, situations of low available bandwidth are not going to disappear in the foreseeable future. As a result, it is important to reduce the amount of data sent by the groupware system. There are several methods for reducing data rate (e.g., send messages less frequently) – but the inefficiency of message formats is an obvious place to start. In many groupware systems, the majority of the data sent between clients is message structure or syntax, rather than critical information. If messages could be represented more efficiently, groupware applications would need less bandwidth, and so would be usable in more network situations (or, alternatively, would be able to send messages more often).

Groupware programmers can make their messages far more efficient by designing compact representations – for example, by using only the minimum number of bits necessary to represent numbers, and by hard-coding the order of fields and parameter lists. In fact, most networked games already do this, and it is clear that they have greatly reduced their bandwidth requirements as a result. However, there are problems with asking groupware programmers to use efficient message representations: compact formats are much more difficult to design and build, are less flexible, and provide none of the benefits described above (such as readability, convenience, or interoperability).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSCW'06, November 4–8, 2006, Banff, Alberta, Canada.
Copyright 2006 ACM 1-59593-249-6/06/0011...$5.00

In this paper we propose an alternate approach: rather than asking groupware programmers to design minimal representations, we have built a groupware message compressor that minimizes message representations automatically. GMC is designed specifically to compress the message structure and syntax that is useful for programmers, but inefficient at the transport level: it does this by building a dictionary of sequences that are repeated across a set of messages, and replaces those sequences with short lookup codes. GMC can also further compress message data using a Ziv-Lempel algorithm and Huffman coding. Furthermore, GMC works with all networking designs, including priority scheduling algorithms, aggregation policies, and network protocols, including unreliable transport protocols such as UDP.

In tests with realistic message traces, GMC dramatically reduced message sizes: text messages to 20% of their original size, XML messages to 8%, and serialized objects to 9%. GMC performs substantially better than per-message Ziv-Lempel compression, which compresses XML to 57% but does not compress text messages or serialized objects at all. In addition, compression with GMC is fast (less than 2ms per message in our test setup), and the system is fully under application control, so that the compressor can fit into an overall application-level QoS scheme.

The main contribution of GMC is that it allows groupware programmers to use representations that are convenient, without sacrificing transport efficiency. GMC is easy to understand and implement, and can be ported to any groupware system.

In the next sections we review the foundations underlying the work, consider where repetition occurs in a groupware message stream, introduce GMC and its compression techniques, report on the performance of the system, and discuss ways that GMC or similar compressors can be used in real-time groupware systems.

2. BACKGROUND

This research is based on four foundations: groupware messaging, groupware network performance, data compression, and techniques for compressing network data.

2.1 Groupware Messaging

Distributed groupware systems have to send information to one another: information about state changes, requests for data, commands, and notification of user events. This communication happens through *messages* – parcels of information sent over the network that encapsulate one update, request, command, or notification [17]. Groupware systems use messages regardless of whether the underlying system uses distributed data structures, remote procedure calls, or a notification server, and regardless of whether the model layers are centralized or replicated (since even centralized systems must distribute information about user actions and view changes).

There are several possible types of message, depending on the application. Examples include model-layer updates, telepointers, streaming multimedia such as voice or video, system-level control and feedback messages, text chat, and session-management messages. Different message types have different characteristics, but two main groups can be considered [5]: *transactions* concern longer-term changes to the system, such as modifications to data structures or lock requests, and are usually infrequent; *streaming messages* are much more frequent, and provide information about the transient state of a user's activity or communication. An important further distinction within the category of streaming messages is between awareness messages and multimedia. Since video and VoIP transmissions are usually already compressed, and are usually handled with established protocols such as RTP [21], we are more concerned here with awareness messages (e.g., telepointers, avatar movements, intermediate locations of objects during drag operations, or changes to view locations). Streaming awareness messages have different QoS requirements than do transactions: in particular, they do not all have to arrive, but they do need low latency. This means that they are better sent using UDP – which is faster but non-guaranteed – than TCP [5].

Both transactions and awareness messages are usually small enough to be sent whole, and are routed to other clients depending on the system's distribution architecture. In centralized-communication systems, each client connects only to the server; in a peer-to-peer architecture, each client makes a connection to every other client [17]. The distribution architecture plays a major role in a groupware system's overall data rate, since routing determines whether each messages is sent once (to the server) or multiple times (to each client, assuming no native multicast).

We are concerned primarily with real-time distributed groupware systems that send awareness messages and transactions. This covers several types of real-world groupware: shared workspaces, shared editors, collaborative virtual environments, and networked games. We do not focus on asynchronous awareness servers or media spaces since awareness systems generally have minimal data volumes, and since multimedia systems generally use well-established protocols for sending information (such as RTP).

Although some effort has been made to establish standard protocols for groupware (e.g., [12]), these are not yet widespread, and in practice, groupware programmers must determine what to send and how to send it. A groupware message must encode several pieces of information, both data and metadata:

- the sender of the message (client ID)
- the message ID (for loss detection and ordering)
- the application ID (since several groupware applications may be sharing the same network port to get through firewalls)
- a timestamp (required to control playback from a buffer)
- the message type (e.g., telepointer move, model update)
- field names for each parameter (e.g., x position, y position)
- data values for each field

This information can be represented in several ways. Different representations have different advantages, but usually at the cost of message size. However, the strengths of text, XML, and objects are enough that most groupware systems have used these formats: for example, text strings are used by GroupKit [19] and TeamRooms [18]; XML by Disciple [14], and serialized objects by JSDT (jsdt.dev.java.net/) and JAMM [2].

2.2 Groupware Network Performance

The performance of real-time distributed groupware over real-world wide-area networks has been frequently criticized (e.g., [4,10,25]). These performance problems are primarily due to network issues: latency, which is the time required for information to travel between locations; jitter, which is variance in latency; loss, which results from network packets not arriving at their destination; and insufficient bandwidth. These problems are common in today's wide-area networks, and although networking advances are aiming to reduce these problems, the problems will

be present for some time to come. In the meantime, groupware applications must attempt to deal with these issues themselves.

Network delay has been shown to have serious effects on users. It can cause difficulties in coordinating collaborative actions [25,16], in predicting others' intentions [8], and in interpreting gestural communication [9]. The overall effect is that collaboration breaks down – groups tend to decouple their collaboration and work more independently. When latency becomes extreme (as happens when systems exceed their network bandwidth), the distributed parts of the application appear to grind to a halt – telepointers freeze, locks are never granted, and changes are never propagated.

Delivering network performance in real-time groupware is a difficult task due to the diverse performance needs of the applications, and the situational factors that affect performance requirements [7]. There are many different genres of groupware (including games, whiteboards, conferencing systems, shared editors, virtual classrooms, and collaborative virtual environments), and each application can have multiple interaction techniques with diverse quality of service requirements. The applications also need to keep model layer data synchronized and these requirements vary based on the needs of the application. In addition, performance requirements are affected by situational factors such as proximity to other users, the level of coupling and dependence between actions, and the level of awareness that a user wishes to maintain of their collaborators.

The diversity of requirements in groupware means that no one technique can solve all performance problems. However, bandwidth restrictions are one of the most critical causes of latency in distributed systems, and the size and efficiency of groupware messages will play a major role in any attempt to improve groupware performance.

2.3 Data Compression

Compression means making a piece of data smaller by finding a more compact way to represent it. There are two main types of data compression – lossless, which guarantees that the decompressed data is exactly the same as the source, and lossy, which accepts some reduction in the quality of a picture or signal in order to achieve higher compression. For groupware messages, we are almost exclusively interested in lossless methods, since all of the parts of the message are required in order to interpret it correctly. Lossy methods can only be used to compress certain types of data elements within the message (such as pictures). There are many lossless compression techniques. These are generally categorized into statistical modeling and dictionary methods (see [3] for more detail on these methods).

Statistical modeling. Statistical modeling schemes represent the symbols of an alphabet using variable-length bit sequences, and use shorter sequences for more likely symbols. For example, standard ASCII characters are represented with 8-bit codes, but in many cases some characters are more likely than others; if the more frequent characters had shorter codes, many messages could be shortened. Huffman coding [11] and arithmetic coding [3] are the most common methods for generating optimal bit codes. Sometimes ad-hoc approximations to these are used, such as coding the 15 most common values in 4 bits.

A more powerful version of statistical modeling takes account of the context in which a symbol appears in order to more accurately determine its probability; for example, the likelihood of an 'e' in English text becomes much higher if one knows that the preceding two characters were 't' and 'h'. The more accurate probability distributions improve coding efficiency. The most advanced context-modeling technique is the PPM family [3], which is usually coupled with an arithmetic encoding scheme. PPM techniques usually achieve the highest compression, but are slower than other techniques. A related method is the BWT transform, which permutes the text using sorting so that characters are ordered by their context. This approximates PPM, but is much more efficient to compute, and is used in the popular utility `bzip2`. It has the disadvantage that data must be compressed in reasonably large blocks, so is unsuitable for an interactive system.

Dictionary compression. Dictionary techniques replace sequences of several symbols in the source message with indexes into a dictionary. For example, the string "moveTelepointer" could be replaced in a message by a dictionary index (e.g. '1') as long as both the sender and the receiver are using the same dictionary. The entries in the dictionary can either be determined beforehand, or built up adaptively as messages are read. The most well-known adaptive dictionary encoders are the Ziv-Lempel family (e.g., LZ77 [26] and LZ78 [27]) which are seen in common tools such as `gzip` and `deflate`. Some Ziv-Lempel techniques work without a separate dictionary, by replacing a sequence with a pointer to a previous occurrence of that sequence in the message itself. Thus the text is its own dictionary, and adaptation happens naturally as the nature of the text changes.

2.4 Network Compression

Most data compression techniques are designed for use in compressing files rather than a sequence of individual messages sent out across a network. In the case of a stream of messages, additional issues must be taken into consideration. One main concern is reliability – some transport protocols (e.g., UDP) that are commonly used to send groupware messages are not guaranteed to arrive. These protocols are essential because they provide a much better fit to the QoS requirement of streaming awareness messages than reliable protocols such as TCP [5]; therefore, the message compression technique must be able to deal with the loss of some parts of the overall data stream. As a result, the Ziv-Lempel techniques that refer to previous parts of the stream (e.g., LZ78 [27]) cannot be used for compressing the entire stream as if it were one file (these techniques can still be used to compress individual messages, however). In the network setting, a variety of compression techniques have been used:

- *Hardware compression*. Some hardware manufacturers have attempted to address the problem of bandwidth limits with network compression modules. These devices compress individual outgoing packets and then transport the resulting message over a connection to another such device, which decompresses it.
- *Packet compression.* There are a number of schemes for compressing IP packets. These techniques use intra-message compression (i.e., each packet has its own dictionary) or probabilistic algorithms for compression [24]. For example, HP packet-by-packet compression [22] uses per-packet dictionaries, run-length encoding, and a reduced code set. In one evaluation, this method compressed general network packets to approximately 52% of original size. Although these techniques have been used to improve performance for

some types of network connections (e.g., PPP), they are still not commonly seen. One reason is that they are generally not under application control – that is, they require additional computation time, and in some cases, this additional time is unacceptable to the application.

- *Delta compression*. Another manner in which network traffic can be compressed is via delta compression (also called delta encoding). Here, after an initial state is established, messages that are transmitted contain only changes since the last update. This is the technique used in screen-sharing systems such as VNC [1]. Note that in most cases, delta compression has strict reliability requirements, since the delta values are relative to a previously-sent message.
- *Tailored compression schemes*. Some systems can have their bandwidth use dramatically reduced if there is knowledge about how the data in the messages will be used (or not used). A good example of this is Compressed X (www.vigor.nu/dxpc/), a protocol for sending X-Windows messages more efficiently. Compressed X does several things to reduce message size: it removes all unused information from the message, recodes common remote procedure calls (RPCs) using shortcuts, and recodes the data of these RPCs with minimal bit-length representations.
- *Game techniques.* Multiplayer networked games represent a highly evolved category of groupware. As such, game programmers have had to address the problem of compressing numerous messages sent over lossy connections [23]. Games are strongly oriented towards using minimal representations, but they achieve this in different ways. First, some game networking libraries provide programming abstractions that let the programmer specify representation. For example, the Torque Network Library [6] provides wrappers on all 'writeToStream' messages that allow the programmer to specify the number of bits to use. Second, some game libraries provide standard message structures that the system knows how to compress. For example, Raknet [20] provides standard object types that programmers must use if they want their messages to be compressed. Third, all games also employ dictionaries for remote procedure calls and commonly-used strings.
- *Binary formats for XML.* ASN.1 (Abstract Syntax Notation One) is a standard for the binary encoding of tree formatted data. As such, it has been suggested as a less verbose encoding method for XML [13]. This method requires that the application programmer define an XML schema for the messages, and then convert each message to a binary form. The degree of compression is dependent upon how carefully the designer constrains the schema [15].

Although each of these techniques is effective in some circumstances, none are entirely appropriate for groupware compression. In the case of IP-packet and hardware compression, the techniques rely on within-message redundancy, which as will be discussed below, offers relatively small savings for groupware messages. Furthermore, hardware compression requires devices at either end of the data link, which limits the generality of the method. Delta compression is inappropriate because of its reliability requirements; in practice (e.g., with systems like VNC), this presupposes a TCP-style lossless transmission protocol. The remaining techniques (custom formats, game formats, and binary XML formats) are applicable to groupware, but require a non-trivial degree of work on the part of the programmers and designers. Much like creating minimal length encodings for messages, these techniques make the process of developing and modifying the application far more complex, and require that programmers build their systems in particular ways.

Given that existing techniques are not perfectly suited to the requirements of groupware, we next look in more detail at groupware messages, and consider where savings could be found from a groupware-specific compression scheme.

3. SOURCES OF INEFFICIENCY IN GROUPWARE MESSAGES

There are three main sources of redundancy in groupware messages that can be exploited for compression: repetition of sequences within a single message; repetition across several messages, and inefficient encoding of the symbols of the message.

3.1 Repetition within a single message

There are often repeated sequences inside groupware messages, and these repetitions can be compressed using a dictionary scheme. This is the approach taken by some existing packet compressors. However, most groupware messages are relatively short, and dictionary compression is less effective with short source documents because there is less opportunity for sequences to be repeated.

For example, the example moveObject message in Figure 2, generated as a user drags an object across the screen in a shared whiteboard, shows the sequences that can be replaced by pointers to earlier occurrences of the text. Most of the repeated sequences are short, and (in this case) none are repeated more than twice. Only a small reduction is therefore possible by compressing the message as an individual document.

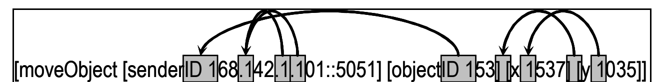

Figure 2. Within-message repetition.

3.2 Repetition between messages

In contrast, there is often considerably more repetition from one message to the next, as long as messages are of the same type. Figure 3 shows two example moveObject messages. Several elements of the two messages are identical: the message type indicator, the sender and object IDs, parts of the data elements, and the field-delimiter syntax. As can be seen in the figure, all but a few bytes of the message exactly are the same.

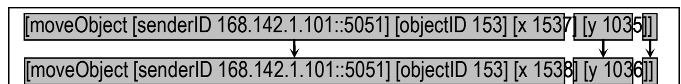

Figure 3. Between-message repetition.

This repetition of certain message patterns is the basis of the GMC template compression scheme described below. The amount of compression that is possible with this approach depends on the amount of self-similarity in the messages. Most groupware systems use several message types, and other types of messages (e.g., telepointer or chat messages) will show much less similarity to the object move message in Figure 3. For each message type, however, there will be considerable self-similarity, and in many systems, there are only a small number of message types.

Furthermore, at least some types are likely to be seen often (e.g., awareness messages such as object and telepointer moves).

3.3 Inefficient encoding

Groupware systems often encode the elements of a message inefficiently. For example:

- *Encoding of numbers*. String representations of numeric values takes more space than binary representations.
- *Field widths for numeric data types*. Some fields use more space than is needed: for example, screen coordinates are often represented as four-byte integers, even though they have a much smaller possible range.
- *Symbol frequency*. The symbols (e.g., the characters) used in groupware messages do not occur with the same frequencies, but are often represented using the same number of bits. As described above, schemes such as Huffman coding give the more frequent symbols shorter codes.
- *Structure as text*. Field names and message types are often represented as text strings; these could be represented as an enumeration that would use considerably less space.

How small is an efficient encoding? As an example, Table 1 compares the text-encoded message of Figure 3 with a customized minimal format that represents values in binary, and uses minimal-width fields for the possible data ranges.

Table 1. Sizes of text and minimal representations. The ‘padding’ field is needed to reach a byte boundary.

Field	Range	Text format	Minimal format
Message type	0..31	12 bytes	5 bits
Sender ID	0..63	31 bytes	6 bits
Object ID	0..1023	15 bytes	10 bits
X	0..2047	9 bytes	11 bits
Y	0..2047	9 bytes	11 bits
Padding			5 bits
Total		608 bits (76 bytes)	48 bits (6 bytes)

In the next section, we introduce GMC, a compressor that performs both between-message and within-message compression.

4. GMC

The goal of GMC is to automatically and quickly obtain most of the compression that is possible with a hand-coded representation, but without requiring any extra work on the part of the application programmer. GMC has two parts: a template compressor that finds repeated structure and syntax elements between messages, and a within-message compressor that uses Ziv-Lempel to reduce the size of large, data-intensive messages.

4.1 Between-message (template) compression

The goal of between-message compression is to remove sequences that are the same across several messages in a stream. The scheme works by treating one message as a template and then comparing subsequent messages to the template to determine which sequences are repeated. Any repeated sequences are assigned a code and entered into a dictionary; these codes can then be used to replace the sequence in any new message. There are three parts to the scheme: creation of a new template, compression of a message, and announcement of new templates. These are described below, and general algorithms are given in Figure 4.

Function **createTemplate** (message *m*)

- create new template *T*:
  - build a suffix trie *S* from message *m*
  - create a lookup table *L*
- compress the previous *n* messages with *T* and determine the overall compression ratio
- if the ratio with *T* is better than the previous ratio:
  - if there are fewer than the maximum number of templates:
    - announce *T* as a new template
  - else
    - find the least efficient existing template *Q*
    - announce *T* as a replacement for *Q*

Function **compressMessage** (message *m*)

- for each template *T*:
  - *remainder* = *m*
  - *result(T)* = Ø
  - while *remainder* is not empty:
    - find the longest prefix *p* of remainder in template *T*
    - if there is a prefix:
      - if *p* is in the lookup table:
        - add the key for *p* to *result(T)*
      - else
        - add *p* to the lookup table and add the new key to *result(T)*
      - remove *p* from *remainder*
    - if there is no prefix of *remainder* in *T*:
      - find the sequence of characters *s* that is not in *T*
      - add the escape code and *s* to *result(T)*
      - remove *s* from *remainder*
  - if *T* has a Huffman code:
  - encode *result(T)* with the code set
  - record the compression ratio *r(T)* = size(*result*) / size(*m*)
- if no *r(T)* is lower than the desired compression ratio:
  - create a new template from message *m*
- return the best result and the number of the associated template

Figure 4. Algorithms for creating templates and compressing messages using templates.

Template creation. A new template is created for a message by constructing three elements:

- a suffix trie from the bytes of the message, which allows for fast determination of longest-substring matches in the compression phase;
- a lookup dictionary (initially empty), which will be used to store repeated sequences as they are found during the compression phase;
- a Huffman frequency table, which keeps track of which dictionary entries are used most often; after a number of messages have gone by, the frequencies are used to create a Huffman code set for the template.

Message compression. Compression of a new message involves finding the longest substrings of the current message that are also in the template. Whenever a substring is found, we replace it in the original message with the corresponding dictionary key. If we encounter bytes that are not in the template, we add them directly to the output as an escape sequence. A compressed message therefore consists of lookup codes and escape sequences.

Announcing new templates and dictionary entries. Whenever GMC creates a new template, it must send the template to other clients in the groupware system. Similarly, whenever the compressor adds an entry to the dictionary, it must tell others to add the entry to their dictionaries as well. We call these notifications *announcements*, and we assume that they are sent reliably (since messages cannot be decompressed correctly until the announcements arrive). The number of announcements depends on the characteristics of the message stream itself, but the expected number of templates is low – in the evaluation below, a stream of about 13,000 messages resulted in seven templates (of about 700 bytes each) and 781 dictionary entries (6 bytes each).

There are a few message types for which templates are not useful – in particular, one-of-a-kind messages, such as those that send model-layer data to late entrants. In these cases, it would be useless to have the other clients build a template: therefore, we do not tell others to build the template until the second message of a type comes in. That is, GMC only announces the new template when it is used for a new message (showing that there is at least one other message that is similar to the template).

4.1.1 Balancing compression ratio and time

The algorithm used in GMC has two variables that can be used to tune the compression: the amount of time needed to compress a message, and the desired compression ratio. GMC attempts to automatically find a reasonable balance: with each message, it gradually lowers the target compression ratio until compression time exceeds a threshold (currently 2ms).

A third variable in GMC is the number of templates created by the scheme. We assume that GMC must work on a heterogeneous stream of messages, and so must create as many templates as there are self-similar message types in the stream, in order to achieve good compression. GMC does this automatically, by creating new templates whenever the compression rate for a new message drops below a threshold. GMC maintains a set of templates, and compresses a message using the template that gives the best result. Multiple templates means that each message must record the template that was used to compress it, information that is placed in the message header (see below).

GMC's template compressor is completely unaware of the structure or meaning of the messages that it processes. This means that it will work with any message type, and also means that it is naturally adaptive to changes in the message stream. That is, if one type of message gives way to another, GMC will simply create a new template for the new message type.

4.2 Within-message compression

The template compression scheme described above removes repeated sequences that appear in every message in the stream. Although this removes a main source of redundancy, there are still cases where the data elements of the message contain repeated information (e.g., the text part of a chat message). Since template compression does not affect elements that are unique to a message, data-intensive messages can be compressed further.

GMC identifies data-intensive messages by keeping track of the size of the escape sequences; when this value exceeds a threshold (currently 100 bytes), GMC runs a standard zlib algorithm (deflate) on the output of the template compressor. Deflate uses a mixture of dictionary compression and Huffman encoding to compress the input. The resulting message size is compared with the input: in cases where the algorithm actually increases the size of the message, no compression is carried out.

Within-message compression is only valuable in a few cases (e.g., when participants send files to each other, or when the system sends the model layer to a late entrant). Nevertheless, it is inexpensive to include the functionality in GMC, since zlib is built into many class libraries (we use java.util.zlib), and can make a substantial difference for large messages.

4.3 A compressed-message protocol

A compressed message must be sent with information about how to decode it. We assume that GMC has already sent the template and dictionary announcements needed to decompress the message; however, we must send additional information with each message: namely, whether zlib compression is used, whether template compression is used, and if so, which template.

We designed a simple message protocol for GMC messages that contains this information (see Table 2). This protocol could easily be integrated into a more complete groupware message protocol, but for illustration, we have developed it to stand alone. The protocol uses a variable-length header that can be up to 16 bits: one bit to state whether deflate is used or not; one bit to state whether template compression is used or not; and if template compression is used, 14 bits to state the template number. This allows for 16,384 templates across the entire system (e.g., 1,638 clients who each announce ten message templates). Note however that this number of templates would only be used for decompression: for compressing, each client only uses the small number of templates that they themselves have generated.

Table 2. Message protocol for GMC-compressed messages

Header			Payload
Deflate?	Template?	Template no.	Compressed message
1 bit	1 bit	14 bits	Variable length

4.4 An example of GMC in code

Figure 5 shows an example of the code that would be written to use GMC in a Java groupware system. We assume that there are generic Sender and Receiver objects that handle network communication. As can be seen from the example, using GMC requires only a few additional calls; in addition, if GMC were to be integrated with the network layer, it could be made completely transparent to the programmer.

```
Sender:
GMC gmc = new GMC();
String message = "(timestamp: 716626384 sender_id: 12
   session_id: 9357)(telepointer x: 308.0 y: 78.0)";
Sender.send(gmc.encodeString(message));
Receiver:
GMC gmc = new GMC();
String incoming = (String) gmc.decodeString(Receiver.receive());
System.out.println(incoming);
```

Figure 5. Compressing and decompressing a text message

5. EVALUATION OF GMC

We tested our implementation of GMC by comparing its performance with other compression schemes on three realistic

message traces. The goals of the evaluation were to compare compression ratios and computation time, and to determine how GMC compared both to hand-coded representations and to other low-effort compression techniques.

5.1 Dataset

We used a GroupKit message trace, recorded from a drawing program, as the basis for an evaluation dataset. The trace has 12,850 messages gathered over 47 minutes, and contains seven message types: telepointer position, object create, object resize, object move, text chat, text box edit, and late-entrant model-update messages. The relative frequencies of each type are shown in Table 3. Note that object-resize messages are common because this function is used in creating a new object (i.e., objects are created on mouse down, then resized as the mouse is dragged).

The messages were used to generate new traces in three different formats: text, XML, and Java objects. Examples of each format are shown in Figure 6. The overall compressability of the streams was tested by running gzip on the text and XML traces as files: text was reduced to 10.4% of its original size, and XML to 4.2%.

Table 3. Number and frequency of message types.

Message type	# in trace	% of total
Telepointer position	10593	82.4%
Object create	97	0.75%
Object move	265	2.1%
Object resize	958	7.5%
Text chat	20	0.15%
Text box edit	916	7.1%
Model update	1	<0.01%

Text:

```
(timestamp: 716626384 sender_id: 12 session_id:
9357)(telepointer x: 308.0 y: 78.0)
```

XML:

```
<group_event>
    <timestamp> 716626384 </timestamp>
    <sender_id> 12 </sender_id>
    <session_id> 9357 </session_id>
    <event_content>
        <telepointer_event>
            <x> 308.0 </x>
            <y> 78.0 </y>
        </telepointer_event>
    </event_content>
</group_event>
```

Objects (class definitions):

```
abstract class AbstractGroupEvent {
    int sessionID, senderID;
    Long timestamp;
}
class TelepointerEvent extends AbstractGroupEvent{
    int x, y;
}
```

Figure 6. Examples of the three test representations.

5.2 Compression techniques

We compared GMC to several other techniques: two types of hand-coded representation, a within-message zlib compressor, and an XML-specific representation called ASN.1.

- *GMC* operated as described above, in 'fully automatic' mode, with template compression and within-message compression (deflate). The desired compression ratio started at 30%, and was reduced for each message until compression time exceeded 2ms per message. Two bytes were added to each message for the message header (see Section 4.3).
- *Hand-coded* is a custom binary representation that uses no syntax or delimiters, converts all field names and message type names to enumerated types, and represents all data values in a minimum number of bits. Table 1 above shows an example of this type of representation. No padding was counted in the experiment.
- *Primitives* is similar to the hand-coded representation described above, but we assume that all elements of the message are represented by the shortest primitive type (e.g., byte, short, integer) rather than minimal bit values.
- *Deflate* applies a Ziv-Lempel technique (deflate) to each message individually. The technique is from the standard Java library (java.util.zlib) and is set to 'balanced' mode, which produced the best compression in pilot tests.
- *ASN.1* is a standard for the representation of tree data that has been proposed as a method reducing the verbosity of XML messages. The schema for our messages was not highly constrained, to simulate a groupware programmer who is not focusing on compression. We encoded the messages using the Unaligned-PER ASN.1 scheme [13].

5.3 Methods and Results

We built a test application in Java that would carry out compression and decompression for each technique and for each of the three message traces (text, XML, objects), and record original size, compressed size, compression time, and decompression time. The trials were run on a Macintosh PowerPC G4 system running at 1.4 GHz. Table 4 shows the results of these tests, and also shows each technique's savings compared to the original, and compared to the best technique (i.e., hand-coded).

Figures 7 and 8 show the compression ratios and times for the different techniques (all decompression times were less than 1ms for all techniques). GMC created seven templates for the text trace, three for the XML trace, and four for the object trace.

Table 4. Results of compression tests with text, XML, and serialized-object message traces.

Technique	Compress time (ms) per message	Size (bytes) per message	Percent of Original	Percent of Hand-coded
Text				
Original	0.0	88.4	100%	920%
Hand-coded	≈0.0	9.6	11%	100%
Hand-primitive	≈0.0	12.2	14%	127%
GMC	1.0	17.7	20%	181%
Deflate	2.9	86.8	98%	903%
XML				
Original	0.0	224.5	100%	2455%
Hand-coded	≈0.0	9.1	4%	100%
Hand-primitive	≈0.0	12.2	5%	127%
GMC	1.1	19.7	8%	199%
Deflate	2.4	134.4	57%	1398%
ASN.1	0.5	47.2	20%	491%
Serialized Objects				
Original	0.0	277.5	100%	2887%
Hand-coded	≈0.0	9.6	3%	100%
Hand-primitive	≈0.0	12.2	4%	127%
GMC	2.8	25.3	9%	263%
Deflate	12.3	231.6	83%	2410%

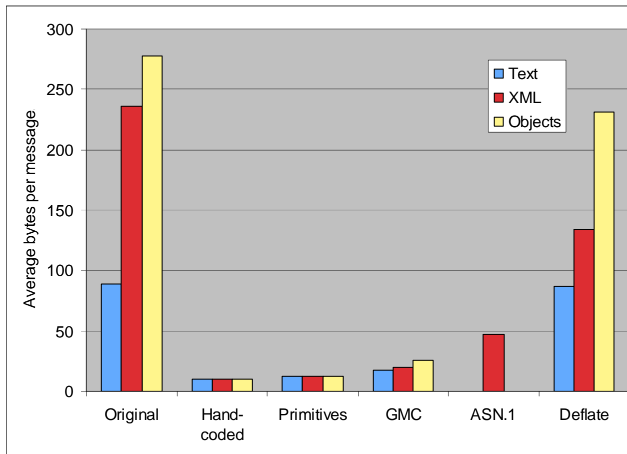

Figure 7. Message sizes (payload) for all compression techniques (for GMC, size includes all overheads).

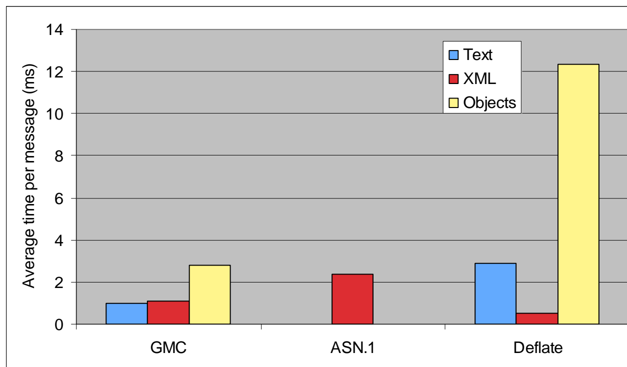

Figure 8. Average compression time per message, for all techniques (note that for the original trace, compression time is zero; for the hand-coded representations, it is near zero).

5.3.1 Effects on actual bandwidth

The evaluation shows that compression provides a substantial saving compared with the original messages. However, the results include only the groupware messages themselves; if the messages were to be sent across an actual network, they would have to be sent inside packets, which would incur additional cost for the packet headers. We do not include packet overheads in our calculations, however, because there are different ways of sending the messages that would result in different amounts of overhead (and which have nothing to do with the compression techniques).

The most basic method for transmission is to send one message per UDP packet, which adds 28 bytes (20 for the IP header, 8 for the UDP header). This is often larger than the compressed message itself (see Table 4), and so this method would dramatically reduce any difference between the compression techniques. However, there are other ways to send messages that avoid this overhead. In particular, we assume that several groupware messages will be aggregated into a single packet, and that the payload portion of a packet will be much larger than the header when it is sent. For example, if packets contain 10 messages of 25 bytes each, header overhead is reduced to 11%. In this case, the total bandwidth requirement for sending 30 messages per second (for the object stream) is reduced from 6819 bytes/sec to 834 bytes/sec, enough savings to allow for a voice link, a higher-quality video signal, or a higher message frequency.

5.3.2 GMC overhead in the evaluation

Overhead in GMC consists of three types of announcements: new templates, additions to an existing template's dictionary, and frequency lists for templates that are Huffman-encoded. The data sizes for all overheads are included in the results above, but are reported separately here in Table 5.

Table 5. Details of GMC announcement overhead

Data Type	Templates	Dictionary Additions	Frequency Lists
Text	7 (718 bytes)	781 (6 bytes each)	5 (1K bytes each)
XML	4 (1077 bytes)	534 (6 bytes each)	3 (1K bytes each)

Note that we do not include packet-header overhead in these results, for the same reason as given above – that there are several ways that announcements could be sent, and packetization could be different across these different schemes. For example, announcements can be sent reliably (without requiring a separate TCP channel) if new announcements are included at the front of every outgoing aggregated UDP packet, until the receiver sends an acknowledgement.

6. DISCUSSION

In this section we summarize our main results, and consider several issues that arise from this research: whether programmers should use GMC or hand-coded representations, how the GMC results generalize to other groupware traffic, possible improvements to GMC, and how GMC works with other network infrastructure in a groupware system.

6.1 Summary of results

Our experiments show that GMC provides substantial improvements both over uncompressed messages and over 'low programmer effort' methods like ASN.1 and deflate (Ziv-Lempel). Compared to the original text stream, the GMC-compressed messages were 20% of the original size, and XML and serialized objects were even better (8% and 9% of original). In contrast, the Ziv-Lempel algorithm was not particularly effective on text or serialized objects, and only reduced the XML messages to 57% of original size. Last, the ASN.1 encoding reduced XML to 20% of original size, but was still more than twice the size of GMC.

Furthermore, GMC was extremely fast in comparison to other techniques: with our test setup, GMC used between 1.0 and 2.8 milliseconds to compress messages; in contrast, deflate used between 2.9 and 12.3 ms per message.

However, GMC produces messages that are larger than hand-coded representations (either minimal-bit or minimal-primitive representations). GMC-compressed messages were approximately twice the size of the hand-coded version for all three message traces. Thus, although GMC represents a dramatic saving over uncompressed messages, there is still room for additional compression if the application programmer wishes to work for it.

6.2 GMC vs. hand-coded representations

Given the superior compression of the hand-coded representation, it is worth asking whether groupware programmers should simply

use these minimal formats. There are three reasons why programmers should consider GMC. First, the amount of effort required to design and maintain the hand-coded format is substantial; in contrast, GMC requires almost no effort. Second, even though GMC messages are twice the size of hand-coded messages, the actual effect on bandwidth is relatively small (ignoring packet overhead): moving from GMC to a hand-coded representation would change the bandwidth requirements (for 30 messages/sec) by only a small amount. Third, there are ways that the compression of GMC can be improved (see below), which will further reduce the difference between the two techniques.

There are, however, situations where the hand-coded representation should be chosen over GMC. In particular, if a very large number of messages is sent out per second (e.g., in a peer-to-peer system with many clients), then the bandwidth savings of a minimal representation will multiply. Although these situations may happen, they are not likely, particularly given the popularity of the centralized-message-server architecture that limits upload bandwidth requirements for groupware clients.

Even when extreme compression is required, it may not be required immediately in the development process; therefore, GMC is valuable for prototyping groupware systems and testing them in real-world conditions, even if the programmer wishes to eventually move to a hand-coded representation.

Last, in some cases there may be tools other than GMC that can provide a balance between programmer effort and compression. Game libraries offer compression rates that are near hand-coded levels, and if the programmer is willing to conform to the abstractions dictated by the library, they can achieve good results without doing the custom representation themselves. Similarly, if a programmer wishes to carefully constrain the schema of an XML message type, good compression can be achieved with the ASN.1 format. In these cases, however, there is still a considerable amount of effort required – game libraries generally provide only low-level support, and constraining an XML schema for ASN.1 still requires that programmers calculate much of the representation themselves. No method other than GMC offers high performance with minimal effort.

6.3 Generalizing the results

The main strength of GMC is in reducing between-message repetition, and so GMC will work well in any groupware system that uses message types that have some characteristics in common with those seen in our experiments. In addition, the more verbose the message structure, the better GMC will work – so if a groupware programmer defines a less verbose text representation, the savings from GMC will be reduced (although likely still worth doing). Finally, some groupware systems will send more data-intensive messages (e.g., pictures or sounds), and GMC will be less effective on these messages. It will still compress both the structure and the content of the messages, but its performance will move towards that of a basic zlib implementation.

6.4 Possible improvements to GMC

There are several possible improvements that could be made to various parts of the GMC scheme. Here, we consider three: composite templates, recoding of numbers, and aggregation.

Composite templates

The structural elements of two similar messages are exactly the same (see Figure 3), but GMC can only recognize contiguous repetition: that is, wherever there is a variable data element in a message, GMC must output two separate lookups: one for the repeated sequence before the data value, and one for the sequence after. If templates could have 'holes' in them for data values, then an entire message structure could be saved in a single template, saving several bytes in lookup codes.

One simple way to accomplish this is to build a second set of templates on the output of the first pass. For example, if the system repeatedly sees the lookup-code sequence L1 <esc> L2 <esc> L3 in the output of the compressor, it could form a new template (L4), placing the two escape sequences at the end. Although this method would only save a few bytes per message, the messages are already so small (17-25 bytes) that a few bytes could make a fairly large relative improvement.

Recoding numbers from text to binary

GMC currently treats all elements of text and XML messages as strings; but encoding numbers as strings is inefficient. Encoding numbers in binary representations can save a few additional bytes. This could work as follows: when an escape sequence is written out (i.e., a sequence of characters not in a template), GMC could check whether the sequence is a number. If so, a different escape code is used to indicate a number, and a binary representation is written out instead of the characters. Table 6 shows how different number ranges are represented, and the space savings that can be realized. Note that this requires that one of the codes in the 256-entry dictionary be reserved for the additional escape (this is not a difficult requirement, since GMC almost never uses all the dictionary entries).

Table 6. Savings from recoding numbers from text to binary.

Number range	Text bytes	Binary bytes	Average saving
-127..+127	1-3	1	1.13 bytes/number
±128.. ±32767	3-5	2	2.67 bytes/number
±32768..±8388607	5-7	3	4.40 bytes/number

Aggregation

It is possible for several messages to be grouped together into a single packet; if this is done, then there are additional opportunities to compress the aggregated message. In particular, Ziv-Lempel compression techniques are likely to be more effective, since the 'message' is now longer. Aggregation can happen in situations where the data-gathering rate is higher than the send rate: for example, telepointer positions are collected at a rate of 30/second, but are only sent at a rate of 5/second. This may be done for a variety of reasons (e.g., to maintain a high-resolution telepointer playback even in a low-send-rate situation), and would require that GMC coordinate its activities with other elements in a network layer (e.g., the aggregator).

6.5 GMC as part of network infrastructure

Although GMC can work as a stand-alone module (and the reference implementation is designed in this way), it is best thought of as one part in a larger network layer. A more comprehensive network infrastructure for a groupware system should deal with a wide variety of quality of service issues, and should couple the use of compression to monitoring of available bandwidth, rate control schemes, and latency tolerance.

Under the control of a network layer, GMC would no longer run in automatic mode; parameters such as allowable compression time, desired compression rate, and maximum number of templates to create could all be under the control of the application or the network infrastructure. Adjustments to these parameters would be made to achieve overall quality-of-service goals. In addition, having a network layer greatly simplifies GMC's announcement requirements, since we assume that a network layer would already be handling all communication, and could therefore deal with the reliability requirements of the announcements.

7. CONCLUSION

Messages sent by groupware systems can take many forms, and although different forms have different advantages, most common formats are extremely inefficient. Rather than force groupware programmers to build efficient representations, we designed GMC, a message compressor that automatically reduces a wide variety of formats without requiring knowledge of message structure or content. GMC requires almost no effort or attention from the application programmer, is fast, and dramatically compresses text, XML, and serialized objects. Our next steps with GMC are to implement the improvements discussed above, and then integrate the system into a full networking layer for groupware. The reference implementation of GMC is available at hci.usask.ca/research/compression.shtml.

8. ACKNOWLEDGMENTS

This research was carried out as part of the NECTAR research network, and is supported by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] AT&T Corp. *VNC - How it Works*, available at www.uk. research.att.com/archive/vnc/howitworks.html, 1999.

[2] Begole, J., Rosson, M., and Shaffer, C., Supporting Worker Independence in Collaboration Transparency, *Proc. ACM UIST 1998*, 133-142.

[3] Bell, T., Cleary, J., and Witten, I., *Text Compression*. Prentice Hall, Englewood Cliffs, N.J., 1990.

[4] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J. Agu, E., Claypool, M. The effects of loss and latency on user performance in unreal tournament 2003, *Proc. ACM SIGCOMM 2004 workshops on NetGames '04*, 2004.

[5] Dyck, J., Gutwin, C., Subramanian, S., and Fedak, C. High-Performance Telepointers. *Proc. CSCW 2004*, 172-181.

[6] GarageGames. *Torque Network Library Design Fundamentals*, 2005. Available at: opentnl.sourceforge.net/doxydocs/fundamentals.html

[7] Gracanin, D., Zhou, Y., and DaSilva, L. Quality of Service for Networked Virtual Environments. *IEEE Communications Magazine,* April 2004.

[8] Gutwin, C. The Effects of Network Delays on Group Work in Real-Time Groupware. *Proc. ECSCW 2001*, 299-318.

[9] Gutwin, C., Penner, R. Improving Interpretation of Remote Gestures with Telepointer Traces, *Proc. CSCW 2002*, 49-57.

[10] Gutwin, C., Benford, S., Dyck, J., Fraser, M., Vaghi, I., and Greenhalgh, C. Revealing Delay in Collaborative Environments. *Proc. CHI 2004*, 503-510.

[11] Huffman, D., A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, vol. 40, 1962, 1098-1101.

[12] International Telecommunication Union (ITU), *Standard T.120 - Data Protocols for Multimedia Conferencing*, 1996.

[13] ISO/IEC 8824-1. Abstract Syntax Notation One: Specification of Basic Notation. www.itu.int/ITU-T/studygroups/com17/languages/X.680amd1.pdf, 2006.

[14] Marsic, I. Real-Time Collaboration in Heterogeneous Computing Environments. *Proc. ITCC 2000*, 222-227.

[15] OSS-Nokalva Inc., Alternative binary representations of the XML Information Set based on ASN.1, *Proc. W3C Workshop on Binary Interchange of XML Information Item Sets*, www.w3.org/2003/08/binary-interchange-workshop/32-OSS-Nokalva-Position-Paper-updated.pdf, 2006.

[16] Park, K., Kenyon, R. Effects of Network Characteristics on Human Performance in Collaborative Virtual Environments, *Proc. IEEE Virtual Reality 1999*, 104-111.

[17] Phillips, W.G. *Architectures for Synchronous Groupware*. Technical Report 1999-425. Department of Computing and Information Science, Queen's University, 1999.

[18] Roseman, M., and Greenberg, S., TeamRooms: Network Places for Collaboration, *Proc. ACM CSCW 1996*, 325-333.

[19] Roseman, M., and Greenberg, S., Building Real-Time Groupware with GroupKit, a Groupware Toolkit, *ACM ToCHI*, 3(1), 66-106, 1996.

[20] Rakkarsoft. *Raknet Manual*. Available at: http://www.rakkarsoft.com/raknet/manual/. 2004.

[21] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications*, Internet Engineering Task Force, Audio-Video Transport Working Group, Jan. 1996. RFC-1889.

[22] Seroussi, G. and Lempel, A., *Compression using Small Dictionaries with Applications to Network Packets*, United States Patent 5389922, 1995.

[23] Smed, J., Kaukoranta, K., and Hakonen, H. *A Review on Networking and Multiplayer Computer Games*. Technical Report 454, Turku Centre for Computer Science, 2002.

[24] Tye, C., and Fairhurt, G. A Review of IP Packet Compression Techniques. *PGNet 2003*, Liverpool, 2003.

[25] Vaghi, I., Greenhalgh, C., Benford, S. Coping with Inconsistency due to Network Delays in Collaborative Virtual Environments, *Proc. ACM VRST 1999*, 42-49.

[26] Ziv, J., and Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3), 1977, 337-343.

[27] Ziv, J., and Lempel, A. Compression of Individual Sequences via Variable Rate Encoding. *IEEE Transactions on Information Theory*, 24(5), 1978, 530-536.