Beyond the LAN: Techniques from Networked Games for Improving Groupware Performance

Jeff Dyck, Carl Gutwin, and David Pinelle
University of Saskatchewan
{jeff.dyck, carl.gutwin, david.pinelle} @usask.ca

ABSTRACT

The goal of real-time distributed groupware is to support synchronous work at a distance, but if these systems are to succeed, they must find ways to deal with real-world network issues more effectively. One rich area that can provide network techniques for groupware is network gaming: network games have more than a decade of experience building collaborative applications that perform well on the Internet. The techniques used by games have not traditionally been made public, but several game networking libraries have recently been released as open source, providing the opportunity to learn how games have achieved their network performance. We examined five game libraries to find networking techniques that could benefit groupware; this paper presents the most valuable concepts that we found. The techniques deal with limited bandwidth, reliability, and latency, and include both known techniques and principles that have not been seen before in the groupware community. By adopting these techniques, real-time groupware can dramatically improve network performance on the real-world Internet.

Categories and Subject Descriptors

H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—Computer-supported cooperative work.

General Terms

Performance, Design, Reliability.

Keywords

Networking, Quality of Service, Network Games, CSCW, Groupware, TNL, Raknet, Zoidcom, Enet, Zig.

1. INTRODUCTION

The goal of real-time distributed groupware is to support synchronous shared work at a distance. In order to achieve this goal, groupware must perform well on real-world wide-area networks like the Internet; but although many systems succeed in the research lab, network performance becomes a major problem when they move beyond the LAN and into the real world. On a local area network, it is easy for any networking infrastructure to perform well, since bandwidth is plentiful and packet loss is rare. On real world networks, however, bandwidth is limited and packet loss is common, and current approaches to networking for CSCW applications quickly run into severe difficulty with network delay. Delays cause problems for visual communication, coordination of actions, and anticipation, and generally reduce the richness and quality of real-time collaboration [10]. If real-time groupware is to succeed, it must find ways to reduce delay, and the most obvious place to start is in the infrastructure that groupware systems use to communicate over the network.

There are several disciplines that could be used as a source of ideas for improving groupware networking: for example, there is a great deal of research in video distribution, voice over IP, and distributed simulation. Although this research is valuable (see [5] for a survey), the data used in these domains often has very different characteristics and different quality of service (QoS) requirements than the messages used in groupware systems, and the networking techniques are as a result less applicable to groupware. There is, however, one area of previous work that has much in common with real-time groupware – in fact, that is a type of real-time groupware – and already has a proven record in efficient networking. That area is networked multiplayer games.

The network gaming industry has more than a decade of experience in delivering a high-quality multiplayer experience over the Internet, with millions of users and thousands of game titles. Games also have more in common with groupware than other types of distributed systems: they send short, frequent messages that are generated from human interaction with the system, and they send several different types of messages with different requirements for reliability and latency. A reasonable starting point for improving groupware networking, therefore, is to determine how games send information, and evaluate the applicability of these techniques for groupware. Learning how games deal with networking, however, is difficult since game companies do not generally publicize their techniques, since source code is usually proprietary, and since capturing network packets violates license agreements. This has limited what can be discovered from games, and previous surveys (e.g., [16]) have primarily been based on academic research papers; as a result, many of the novel aspects of game networking have remained undocumented in academic literature.

Recently, however, the source code for several game networking libraries has been released, providing an opportunity to determine exactly how games have achieved their network performance. These libraries provide a more comprehensive look at networking than do individual games, since the libraries bring together more techniques than are used in any single game, and since the techniques have been generalized to the point that they can be used for a variety of games in several genres. We inspected five different game networking libraries – TNL, Raknet, Zoidcom, Enet, and Zig – to find out how they achieve good network performance, and we discovered several networking techniques and principles that have not been considered before by groupware researchers. This paper presents these techniques and principles, and shows how they can be useful to other types of real-time distributed groupware. The techniques address the three critical issues that both games and groupware must deal with on the Internet, limited bandwidth, reliability, and latency, and are organized according to the problems they solve.

There are two contributions of our work. First, we have brought together a set of networking principles and techniques that have been extensively tested by the gaming industry. Even though some of the techniques in the list have been seen before in other network research, it is valuable to know which principles have stood the test of time and been proven valuable enough to retain a place in the networking libraries. Second, some of the variations of these techniques have not been seen before by the research community, and we describe these techniques in greater detail. These new variations of techniques are the result of two fundamental characteristics of both groupware and games – that messages are small, and that messages have diverse QoS requirements – characteristics that can lead to further research on techniques tailored for the needs of real-time distributed systems.

2. NETWORKING IN ACADEMIC GROUPWARE

Academic groupware that sends real-time awareness information is restricted to the LAN because of its approach to networking. Not all academic groupware is built in the way that this section describes, but it is the most common approach to networking in academic toolkits and applications that we have observed.

We refer to this method of networking as *event-driven TCP*. Using this method, the application's event model triggers events that result in information that needs to be sent over the network. This information is immediately packaged into a TCP packet and sent. Therefore, packet rate is governed by the application's event model. This works fine when events are rare and guaranteed delivery is required, like in a chat application, because the amount of information sent over the network is small and network performance does not affect usability. However, for interaction techniques that require updates to be sent frequently, such as telepointers, event-driven TCP does not work well when groupware applications are used over the Internet.

For example, GroupKit [9] uses event-driven TCP for sending its messages. Telepointer messages are generated and sent on mouse interrupts with one message in each packet, which typically produces packet rates that are between 30 and 60 telepointer updates per second. Each of these packets is sent via TCP/IP, which has a header size of 40 bytes, and each of the text-based telepointer messages has a size of approximately 50 bytes. This produces an upload data rate from moving a telepointer across the screen of around 32Kbps for every user that the messages need to be sent to. In a 4-user shared whiteboard session with a peer-to-peer unicast network architecture, this results in an upload bandwidth requirement of 128Kbps for each user, which is more than many common home Internet packages currently provide. When bandwidth is not sufficient to carry the telepointer messages, messages are queued in the sender's outgoing TCP buffer, and the motion at the receiver lags behind and slowly catches up between mouse movements.

Even for those users with sufficient bandwidth, performance may still be poor due to the reliability and bandwidth throttling mechanisms used by TCP. Using TCP, when packets are lost or delivered out of order, incoming packets are blocked at the receiver and no information is processed until messages are received in order. TCP resends lost messages whether they are still useful or not, and the receiver waits for these late messages before unsynchronized information can be processed. For example, in a shared whiteboard application, a lost telepointer update message means that all messages are blocked, including non-dependent ones like tool selections and chat messages, until the telepointer message arrives. Upon arrival, this telepointer update is late and does not represent the location of the sender as well as other blocked information, and other messages are all delayed while waiting for this late telepointer position. The result is that upon arrival, the ordered burst of messages is processed together faster than the display can display the result, producing an appearance of the telepointer jumping across the screen in an unnatural way. This behaviour has a negative impact on usability, making actions hard to coordinate and gestures hard to recognize, as well as causing telepointer motion to break down [10].

Although GroupKit is an older groupware toolkit, it still serves as a representative example of how many academic groupware toolkits can work. For example, based on documentation and on packet traces we performed on sample applications, Disciple [12], Java Shared Data Toolkit [3], Collabrary [2], MAUI [11], CoWord and CoPowerPoint [17], JAMM [1], Clock [8], and Habanero [4] all handle networking in a similar fashion. Some of these toolkits offer additional networking options (e.g. Java Shared Data Toolkit supports the lightweight reliable multicast protocol), however, the default is to use event-driven TCP.

3. OPEN SOURCE GAME LIBRARIES

The methodology used in this study was to examine open source game networking libraries to find networking techniques that can benefit groupware. We began by identifying all of the open source game networking libraries that are mature, recommended for use on game development web sites, and in use in existing games. We thoroughly read through their documentation, reverse engineered their designs, and inspected source code, noting any networking techniques that are used. The main tool we used for reverse engineering and source code inspection was Understand for C++ [15]. This process produced a list of the networking techniques used in each of the open source libraries. We then categorized the networking techniques according to the problems they solved and evaluated how effective they would be for groupware based on how well they solved groupware's latency, jitter, and bandwidth problems. This produced a list of the most important and useful techniques that appear in network games. We then identified which of the techniques were novel to groupware based on what had been published previously in the groupware academic community. Last, we analyzed what led to the development of these techniques.

3.1 Game networking libraries

The *Torque Network Library (TNL)* [7] is derived from the network code used in the multiplayer games Starsiege: Tribes and Tribes 2. Tribes is a first person shooter (FPS) game that supports up to 32 users. Unlike most FPS games of its time, Tribes was situated in an outdoor setting where many users could see each other at once, and the traffic filtering technique based on visibility that was used in indoor FPS games was not as effective, meaning that Tribes had to send unprecedented amounts of information over the 28.8 Kbps modems it was designed to work with. As a result, much network optimization was needed to accommodate its design. The Tribes network code was further evolved for its sequel, Tribes 2, which included improvements based on the lessons learned in Tribes. After Tribes 2 was completed, the

networking code was packaged by its developers into a general purpose, stand-alone library called TNL, which was offered commercially and has been used in many successful independent and commercially developed games.

Raknet [13] is a commercially developed game networking library that has been used in several commercial multiplayer game releases since 2002. It is frequently recommended by game developers on game development message boards, both for its ease of use and high performance. In 2004, the source code was released, and commercial licenses were offered for free.

Zoidcom [14] is a full-featured commercial game networking library that first appeared as a beta release in 2004 and is still in beta development as we write this paper. The full source code for the library is not available, but the C header files are included and it is well-documented, which reveal several performance-enhancing techniques that are relevant to this study. No commercial game releases that use Zoidcom are available yet, but it has been used in several independent game projects.

Enet [6] was developed as part of an open source first person shooter game called Cube, which was first released in 2002 and has been an active project since then. The Enet library is offered as a separate, stand-alone networking library and is freely available for unlimited open source and commercial use. It offers only low-level services, which include session management, network monitoring, reliable UDP transport, and flow control. Although it has fewer features than the other libraries presented here, Enet's design is well-considered and carefully tailored for the needs of games for the features it supports.

Zig [18] first appeared as an open source project in 2002 and has been an active project since then with regular releases. It is not yet in popular use and it has comparably fewer performance-enhancing features than the other game libraries in this study. It has been included here because of the unique compression and aggregation techniques it uses, which are described further below.

4. GAME NETWORKING TECHNIQUES

Game network libraries have been designed to minimize some of the effects of the most critical network problems that affect usability: limited bandwidth, packet loss, and latency. In this section, we present the most important methods for solving these three problems that we found in the game libraries.

4.1 Bandwidth conservation

Game libraries are designed both to minimize bandwidth usage and to cope gracefully when bandwidth is constrained. The bandwidth minimization techniques used in the game libraries fall under four main categories: encoding and compression, rate and flow control, aggregation, and priority scheduling.

4.1.1 Encoding and compression

The single most important technique for reducing bandwidth is to use minimal representations for the traffic being sent. Game networking libraries encourage this by making it easy for application programmers to efficiently encode information, by making use of dynamically built string lookup tables, and by compressing strings automatically.

Minimum bit-length encodings: Raknet, Zoidcom, and Zig use a similar approach to encoding primitives, where they enable game programmers to easily encode their network traffic efficiently. They provide a bit encoder that makes writing primitives to a bit stream a single method call. This approach results in the application programmers sending only the values as primitives to the network – no parameter names are sent and primitives are never sent inefficiently as strings. TNL improves the efficiency of this process by allowing the application programmer to specify the number of bits to use to encode each primitive. For example, an integer with a maximum value of 5 can represented using 3 bits in TNL, while the other libraries would write a full byte since a byte is the smallest primitive they support. The TNL method can achieve optimal bit-length representations, but requires the application programmer to calculate the number of bits that should be used to represent each value. An example of compressing a payload for TNL is shown in Figure 1.

```
bool b = true;
U32 i = 5;
float f = 0.123;
stream->writeFlag(b);
stream->writeFloat(f, 7);
stream->writeInt(i, 3);
```

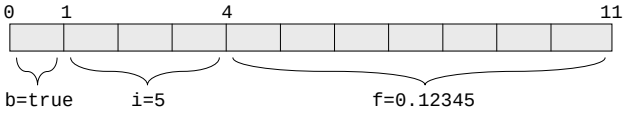

Figure 1: TNL includes methods for writing each primitive to a bit stream, where the first parameter is the value and the second is the number of bits to use to represent the value. The above code would represent the int, float, and boolean as 11 bits total. The 7-bit float value loses precision and is read as 0.118 at the receiver.

String lookup tables: In games, strings are the most costly data type to send, as each character requires a full byte to encode. Therefore, a mechanism for encoding strings that are sent repeatedly as a numeric id can be an effective optimization. However, to do this effectively, we must know which strings will be sent by the application, and this has dependencies on the game, runtime parameters such as usernames and user-configured chat hotkeys, and tasks that are performed within the game.

TNL uses a string lookup table that is generated dynamically at runtime. Applications add strings that are expected to be repeated to the lookup table, they send the string table additions to the other clients, and then they send these strings as a numeric id from that point on. A string can be a single word or a phrase. This dictionary-based approach is well-suited to games since they often send the same strings repeatedly in a session. An example of adding a string to the dictionary is shown in Figure 2.

```
static StringTableEntry returnString(
                "%e0 returned the %e1 flag.");
```

Figure 2: String tables are used to encode commonly occurring strings at runtime. In this example, %e0 and %e1 are both names of players, which vary based on the application instance. Once added to the string table, strings can be sent as numeric string IDs, as shown in figure 3.

Registered remote procedure calls (RPCs): RPCs are a signal to call a method remotely. The method signature and parameters for the method must be sent over the network. An efficient encoding for RPCs is to use an id for the method, which implied the parameter structure, and followed by a minimum bit-length representation for the parameter values.

In TNL, RPCs function in this way. The signatures for RPCs are registered with other clients during runtime as a dictionary entry

with a unique numeric ID. The signature for an RPC includes the procedure name as well as its parameter list as minimum bit-length representations. When sending an RPC, the procedure is encoded efficiently as an ID and a set of minimum bit-length parameter values. An example is shown in Figure 3.

Lossless compression: Game libraries use several approaches to applying lossless compression within individual messages, strings, and packets. The techniques we observed vary greatly, from per-string Huffman encodings to lossless compression applied to entire packets. There seems to be no standard compression method that all gaming libraries agree upon.

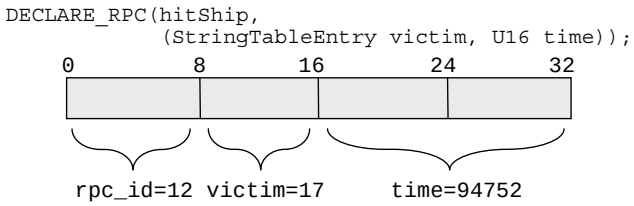

Figure 3: A sample of encoding an RPC using an approach similar to TNL's. The RPC method's id *rpc_id* is encoded as a 8-bit numeric id, and parameters are encoded using minimum bit-length representations. Here, the string *victim* is a numeric 8-bit id for a string in the string lookup table, and the parameter *time* is a 16-bit representation of the system time.

TNL compresses strings that are not inserted into its string table using Huffman coding. Since strings sent by games are often short, Huffman coding can produce a less efficient representation due to the amount of character repetition being small in short strings. The TNL Huffman coder is aware of this, and although it encodes all strings, it only sends the encoded version if it is shorter than the non-encoded version, adding a 1-bit flag to the beginning of the string payload to identify if it is compressed.

The Raknet String compressor functions similarly to TNL's – it applies Huffman coding on a per-string basis. One optimization in Raknet is that it maintains frequency charts for the occurrences of characters in Strings, and the Huffman tree can be rebuilt at runtime and sent to other clients based on the current frequency charts to ensure that the encoding fits the application. This is useful since the character frequencies in strings sent by games may vary based on the application type, runtime parameters, and the task. Raknet's encoding algorithm also combines dynamic Huffman encoding with LZ-encoding, and so its approach is best described as dynamic LZH compression.

Zig compresses entire packet payloads rather than individual strings. This can be advantageous since there may be more character pattern repetition in the full packet and efficiencies can be greater than when compressing only single strings. The compressor Zig uses is the Bzip2 library. Since Bzip2 can produce a longer result than the original when there are few repeating patterns, Zig only accepts the compressed value if it is smaller than the original. Zig adds a further enhancement where the programmer can specify the smallest packet payload to compress, which skips the compression process on small packets that are less likely to benefit from compression. Zig also keeps track of compression ratios, which the programmer can use to adjust the payload size threshold.

4.1.2 Rate and flow control

Exceeding bandwidth limits causes severe usability problems. To prevent this from happening, game networking libraries make use of three main techniques: bandwidth monitoring, static rate control, and adaptive flow control.

Bandwidth monitoring: Different games may require different methods for reacting to limited bandwidth, and in some cases, the decision for how to react is better left to the application programmer. To enable the application programmer to more easily react to changes, all networking libraries include methods for monitoring the amount of bandwidth used. Information available includes the current amount of incoming and outgoing bandwidth available, based on what has been sent over the past second, ping times, loss rates, packet window sizes, and outgoing queue sizes. No two libraries provide the exact same information, but each provides sufficient network information to enable the programmer to make well-informed adaptive decisions. TNL further simplifies the task of adapting to the network by offering virtual methods specifically for reacting to network resources that the programmer can override with their own adaptive logic.

Network rate control: Sometimes, game programmers know what minimum send rates are acceptable for their games. TNL allows the programmer to apply a rate control policy that maintains specified minimum and maximum send and receive rates. The fixed policy uses a credit system where not sending information earns the sender up to one second worth of send rate credit. The credit can then be used when there is a burst of information to be sent at once. The TNL default is to use this fixed policy with a 96 ms delay between packets (~10 packets/second).

Object rate control: TNL, Raknet, Zig, and Zoidcom support object replication, and the rate control technique used for objects is to let the application programmer specify a maximum (and minimum with Zoidcom) update rate for each replicated object. The update rate adaptively changes according to the ranges specified for each replicated object and the network conditions.

Adaptive flow control: One method for not exceeding the amount of available bandwidth is to monitor and adapt to the amount of packets currently in the network, called the window size. This requires acknowledgements to be sent so that the sender knows that a packet is out of the window. This works well for reliable information since the acknowledgements are being sent anyway. However, games send much of their traffic unreliably, so they have to use alternate mechanisms for controlling flow since there are no acknowledgements sent.

TNL's adaptive flow control policy uses an adaptive window size to control its send rate. When packets are received, it increases its window size up to a preconfigured maximum, and when packets are lost, the send window is decreased. The send timer then checks that there is room in the send window and only sends if there is room to send another packet. Since much of the game traffic is unreliable the sender needs feedback on its loss rates, so TNL periodically sends ACKs and NACKs to the sender solely for the purpose of adapting the window size. This is best characterized as a receiver-driven flow control mechanism.

Enet uses an adaptive flow technique, but the approach is quite different than the one used in TNL. The main difference is that Enet does not acknowledge any non-reliable messages, as this adds network overhead. Rather, the window size applies only to

reliable packets. Unreliable packets have a probability of being dropped, which increases as the window size grows. Therefore, the Enet policy is to drop unreliable packets in order to make room to send reliable ones, and to make use of the reliable transmissions for adapting the flow control rather than using bandwidth to send acknowledgements for unreliable information. Raknet's flow control policy works similarly to Enet's, though it does not use probability to determine whether to send unreliable messages. Instead, it suppresses all unreliable messages when the window is full and sends them otherwise.

4.1.3 Aggregation

Messages sent by games are often small, and so multiple messages can often be aggregated together into a single packet. This saves bandwidth consumed by packet headers, as well as reduces the resources required to process packets along the way. Aggregation works by combining messages that are queued to be sent until the maximum packet size is reached, all of the messages in the queue are sent, or the timer for sending a packet is reached. The aggregation approaches described below appear in all libraries except Enet, which does not support aggregation.

Send queues: Outgoing messages are written to an outgoing message queue, and packets are then generated based on the queue contents. The packet is filled with messages up to the maximum transfer unit (MTU) size and sent, potentially containing many aggregated messages.

Frames: Object replication works by creating replicated objects that extend a generic replicated type offered by the networking library. Updates to the replicated objects occur in frames, which combine updates for the shared data structure to send over the network. In TNL, Raknet, and Zoidcom, the application programmer can specify a maximum (and in Zoidcom's case, a minimum as well) update rate for each replicated object. The frames consist of a subset of the replicated objects that is determined by the frequency preferences. The generated frames are written to the send buffer and aggregated with other outgoing RPC traffic. In Zig, object replication is less automated, requiring the application programmer to define exactly what is included in each frame. Only one frame can be defined per application and so the entire set of replicated object updates has to be aggregated by the application programmer into a single frame. This approach requires more effort from the application developer, but it also is more flexible than an automated approach.

4.1.4 Priority

Sometimes, the number of messages in the outgoing message queue exceeds what can be aggregated into a packet. This happens when there are large, dense bursts of messages, when the data rate is low due to limited bandwidth, and it is particularly common when there are many users in the system. One mechanism for degrading gracefully is to mark messages with a priority that indicates the order in which they should be selected for inclusion in packets. This way, the most latency-sensitive messages are sent promptly and less latency-sensitive messages are sent later or dropped from the send queue if their information becomes stale. By using priority queues, bandwidth can be limited while allowing the game to function as best as it possibly can. We observed several different mechanisms for supporting priority.

Numerically assigned and automated: TNL and Zoidcom allow the programmer to set a numeric priority for each replicated object. The sending mechanism then sends individual object updates from highest to lowest priority until the bandwidth available is used. Low priority unreliable messages that cannot be sent are dropped, and low priority reliable messages either wait to be sent or are updated with more recent information.

Numerically assigned but manual: Raknet's approach is to provide infrastructure to enable application programmers to handle their own priority scheduling. Raknet supports different priority levels for information, but does not dictate how the priorities are handled. Instead, Raknet provides several abstract methods that can optionally be implemented by the application programmer to define how the application handles the priorities.

Reliable messages first: Enet uses a policy where reliable messages have priority over unreliable messages. The unreliable messages are dropped according to an adaptive probability. When bandwidth is sufficient, all unreliable messages are sent. However, when bandwidth becomes constrained, the probability of dropping unreliable messages increases adaptively until an equilibrium state is reached.

RPCs or frames first: Zig allows the priority of RPCs to be set to one of two policies. The first policy forces RPCs to be sent out before any additional frames, even if a frame needs to be dropped in order to send the RPC. The second policy waits until the RPC can be aggregated with a game frame in the same packet. Frames do not have priorities.

Deliver at all costs: TNL also offers a "quickest delivery" policy, which assigns priority higher than any other message and keeps sending this message until it is known to have arrived. This policy is described further in section 4.3.2 below.

4.1.5 Bandwidth conservation summary

The techniques reported in this section are all aimed at coping with limited bandwidth. In particular, games minimize bandwidth with encoding and compression techniques, they maximize throughput without exceeding bandwidth limits using adaptive rate and flow control, they aggregate messages into packets to reduce transport overhead, and they use priority scheduling to help to degrade gracefully when bandwidth becomes constrained. These are all techniques that could benefit real-time groupware.

4.2 Low-cost reliability and ordering

Games need to send traffic with variable levels of reliability and ordering. Since games are real-time applications, they cannot simply send all information as reliable ordered since this level of service results in poor performance, as described in section two above. Instead, games make use of two main techniques to achieve reliability - they offer several different combinations of ordering and reliable delivery, and they manage it at a message level rather than at a packet level.

4.2.1 Several levels of reliability

Some game messages require reliable delivery, while some messages do not, and some information needs to arrive in order, while other information does not, and there are various combinations of each. All of the game libraries we examined offer several different QoS options for delivery and ordering. A total of five distinct reliability and ordering policies appeared in the networking libraries. For implementation details, see Raknet [R], as it is the only library that offers all five policies.

Reliable ordered protocols are implemented over UDP by each of the network libraries. The reliable UDP implementations follow the design of TCP, but have some key differences as well. For example, all reliable messages are replied to with acknowledgements, similar to TCP. However, they use more responsive flow control algorithms that share logic with unreliable traffic, and in some cases, they use more elaborate ordering algorithms that are better suited to the needs of games. *Reliable unordered* messages are guaranteed to arrive, but are processed in the order that they are received. This is a useful policy for sending discrete events that are independent of other messages, such as a spaceship being hit by a bullet. The *reliable sequenced* policy drops all late arriving reliable information at the receiver, and also drops packets from the resend queue at the sender when a later packet is acknowledged. *Unreliable unordered* messages are never resent, and the unordered designation simply means that messages are processed in the order in which they arrive. *Unreliable sequenced* messages are not resent, and out-of-order arrivals are discarded at the receiver rather than processed.

The most interesting of these policies is reliable sequenced. Using this policy, only the most recent update to a stream of reliable sequence messages is reliable. The advantage of this policy is that it offers some of the latency and bandwidth advantages of unreliable traffic, but ensures that the most recent updates in a stream arrive. This is a very useful policy for the often bursty streams in games and groupware since this traffic benefits from making the most recent update reliable, but previous updates are unimportant and stale. For example, this policy is well-suited to telepointer or avatar movement messages.

4.2.2 Message-level reliability

Most real-time distributed media provides reliability support using a packet protocol. This is fine when most messages have the same reliability requirements (e.g. VoIP) or when messages are large (e.g. file transfers). However, when messages are small, frequent, and have diverse reliability requirements, it is better to implement reliability at a message level than at a packet level.

All libraries but TNL implement a packet level protocol. This requires all messages in the packet to be treated equally, and performance suffers as a result. As a simple example, consider a packet from a first person shooter game that contains movement message that uses an unreliable sequenced delivery policy and a weapon fire message that uses a reliable unordered delivery policy. Since one of the messages requires reliable delivery, packet-level reliability would require the system to send this packet using a reliable protocol. If the packet was lost, the packet-level protocol would resend the entire packet rather than just the weapon fire message. This example is shown in Figure 4a.

TNL implements message-level reliability, which enables a single packet to carry many messages with various levels of reliability. This is implemented by adding a lightweight reliability header to every message, and there is no reliability portion in the packet header. As an example, consider the same scenario as above where a packet is lost containing an unreliable movement message and a reliable ordered weapon fire message. The movement message could simply be dropped since delivery is not required, and the weapon fire message could be aggregated into the next packet to be sent, which eliminates the need to resend the lost packet. This can produce a tremendous efficiency gain in low bandwidth lossy conditions when network performance becomes critical. This is shown in Figure 4b.

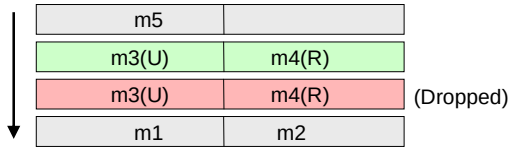

Figure 4a: Using packet-level reliability, when the second packet containing messages 3 (unreliable) and 4 (reliable) is dropped, the entire packet needs to be retransmitted. Here, message 3 is resent even though it is not reliable message, which is inefficient.

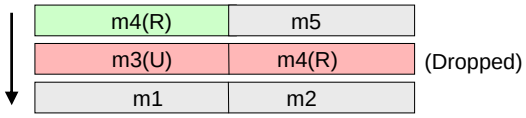

Figure 4b: Message-level reliability allows message 3 to be dropped, and aggregates message 4 into the next outgoing packet, which is more efficient than sending the entire dropped packet.

4.3 Minimizing latency

The priority policies, flow and rate control techniques, and efficient encodings described above partially address latency problems. However, three additional causes of latency remain. The most critical latency problem occurs when ordered messages are lost, which causes other ordered messages to be blocked at the receiver while waiting for the missing message to arrive. A second source of latency occurs when there is not enough bandwidth and the outgoing message queue gets backed up. Last, sometimes time-critical messages need to be sent, and they have to wait in the outgoing queue or get lost the first time and have to wait further to be resent. Games have techniques for reducing latency in all three of these scenarios.

4.3.1 Multiple ordered streams

A significant source of latency comes from having to wait for reliable ordered messages that are lost or late to arrive at the receiver. When this happens, all subsequent ordered messages are blocked at the receiver while waiting for the lost message to arrive. Games partially address this problem by offering several unordered policy options, but still, some information must be ordered and the latency problem can still be significant.

Because of the many message types sent in games and their diverse QoS requirements, it is common to have independently ordered messages. For example, ordered chat messages and ordered firing messages do not need to be ordered together as the order of firing and chat are independent. In this case, a lost chat message should not block firing messages.

To avoid this unnecessary latency, Raknet provides up to 32 independently ordered streams. Each stream is ordered relative only to messages on the same stream. Streams are identified using a channel number, which is specified as a parameter of each message sent by the sender. The channel number is encoded in a message-level header, so although reliability is controlled at a packet level in Raknet, ordering is at a message level. This allows messages that are ordered on separate streams to be aggregated together in a single packet.

4.3.2 The current state data policy

When there is not enough bandwidth, the outgoing send queue can become backed up. Additionally, reliable messages can remain in a resend queue until an acknowledgement is received. While queued, updates to these messages can become available and the queued information can become stale.

The reliable sequenced policy used in Raknet partially addresses the problem of staleness in the reliable queue. Rather than resending reliable data that is no longer current, it drops the message from the resend queue when it receives an acknowledgement that a more recent update arrived at the receiver. However, this policy does not remove stale information when acknowledgements are not received or when the queue is backed up due to low bandwidth.

To optimize for this situation, TNL adds a QoS level that they call "current state data", which ensures that the most recent update to information marked with this QoS flag is sent. Before information is sent, a check is performed to ensure that there is no updated value available. If there is an updated value, the queued information is dropped and replaced with the update. This approach ensures that only the most recent information is ever sent. However, it is important to note that the source of the queued information must be known, so this approach lends itself better to data replication tasks than it does to sending RPCs.

4.3.3 The quickest delivery policy

Some information sent in games needs to arrive before all other information, either because it is highly latency-sensitive (e.g. a hit in a first person shooter game), or because subsequent messages have a dependency on a piece of information (e.g. a new string table entry). Although it is possible to achieve this using logic at the application level, it can make the game programmer's job much easier to implement this at the network library level.

To ensure that highly latency-sensitive information arrives at its destination as quickly as possible, TNL includes a policy that it calls "quickest delivery". It works by including a message in every outgoing packet until it receives an acknowledgement that confirms its arrival at its destination. This guarantees the soonest possible delivery of a message since in the event of a packet loss or a delayed packet, the message is delivered with the next packet that is not lost or delayed instead of having to wait to decide to resend due to loss and without the need to wait for a late packet. This policy trades off bandwidth efficiency for minimized latency since it sends information redundantly, but the penalty is usually not a large one since game messages are small. However, due to this inefficiency, the programmer must take care to not overuse this policy. An example of the quickest delivery policy is shown in Figure 5.

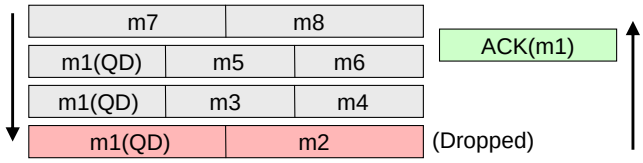

Figure 5: The quickest delivery policy sends a message with every packet until it is acknowledged, guaranteeing that it arrives with the next packet the receiver receives.

5. SHARED WHITEBOARD SCENARIO

The techniques used in games are directly applicable to real-time groupware. As an example, we consider applying the techniques described above to a shared whiteboard application. This section describes steps to applying the types of techniques and presents a brief analysis of the result of applying them.

5.1 Shared whiteboard network design

Whiteboard message types: We assume that the shared whiteboard uses eight message types: chat, telepointer, tool selection, grab, drag, drop, annotation, and session information.

QoS for each message: Chat, annotation, and session information messages should all be reliable and ordered, but do not require synchronization with any other information streams, so they are all sent over their own independent channels. These message types are not very latency sensitive, and so priority can be set low, as they can be sent later in favor of more latency sensitive information. Telepointers are highly latency sensitive, and do not require reliable or ordered delivery, as late information is not displayed in this design. Since telepointers are not dependent on any other messages, they are sent along their own ordering channel. The message types related to drawing operations need to be ordered with one another, as the effect at the receiver will be incorrect if they are processed out of order. The most latency sensitive drawing operations are grab and drop, as these operation lock the object for modification, and so a quickest delivery (QD) policy is applied to these message types. Drag operations, like telepointers, do not need to be reliable, although they do need to be ordered with the other drawing operations, so drag methods are given a sequenced ordering policy and high priority. Tool selections are required to be ordered within the drawing channel, and their priority should be high so that the user's avatar can promptly reflect tool changes. Annotations are designed to update in real-time, so individual messages are individual characters. In order to preserve the metaphor of typing, they are given a medium priority so that latency is reduced. These QoS requirements are shown in table 1 below.

	Reliable	Ordering	Channel	Priority
Chat	Yes	Ordered	0	Low
Tele	No	Sequenced	1	High
Tool	Yes	Ordered	2	High
Grab	Yes	Ordered	2	QD
Drag	No	Sequenced	2	High
Drop	Yes	Ordered	2	QD
Annotation	Yes	Yes	3	Medium
Session	Yes	Yes	4	Low

Table 1: QoS requirements for individual message types in a shared whiteboard.

Efficient message encoding: Chat, session information, tool selection, grab, and drop are all discrete events and are therefore best modeled as RPCs. For each of these, we register an RPC with bit lengths for parameters specified where appropriate. For example, [x.y] coordinates should be encoded such that their limits do not exceed the maximum workspace dimensions, tool selections should be encoded so that the tool parameter does not exceed the total number of tools available to the system, etc. Telepointers and drag operations are modeled as replicated data, and the parameters are encoded using minimum bit lengths.

String compression: The only strings sent by the system are those from chat messages and from pasting text strings to annotations. It

is reasonable to assume that the names of users will be frequently typed in chat, and so usernames are added to the string table when users join a session. Other strings will be compressed using adaptive Huffman encoding, where the characters frequencies are tracked and the Huffman tree is rebuilt and sent out periodically.

Adaptive rates: Telepointers and drag operations are streaming types and are assigned a minimum update frequency of 0 updates per second and a maximum frequency of 30 updates per second. The rates will adapt according to the state of the network, maintaining the maximum rate when network resources are plentiful and decreasing when resources become constrained. Since telepointer updates and drag messages will comprise most of the messages sent by the system, this adaptive frequency range is all that is necessary to allow the system to degrade gracefully when network resources are constrained.

No stale information: Telepointer and drag operations are given TNL's current state data policy, which ensures that only the most up-to-date values are sent. This is possible because the telepointer and drag messages are modeled as replicated data rather than as RPCs. This policy will both reduce latency and traffic.

5.2 Networking design implications

Less bandwidth usage: Since discrete operations are uncommon compared to telepointers, and drag operations and telepointers use the same amount of bandwidth, the maximum bandwidth for a shared whiteboard can be reasonably approximated by considering only that used by telepointers. Using the network design described above, the bandwidth for sending telepointers would be a tiny fraction of what the event-driven TCP model used. Under ideal network conditions, the above approach would consume just over 11Kbps per connected client (UDP/IP header: 28 bytes; custom packet protocol: 12 bytes; message protocol: 3 bytes; telepointer payload: 5 bytes; send rate: 30 messages per second, 1 per packet). This is 1/3 less bandwidth than the version GroupKit uses described in section 2 above.

However, when resources become constrained, the flow controller will begin to aggregate telepointers, which results in a substantial bandwidth saving. For example, aggregating 3 telepointers into each packet would result in a bandwidth usage of 5Kbps per client due to the reduction in packet headers that need to be sent. Additionally, as bandwidth becomes so constrained that it can no longer support 30 updates per second, the telepointer rate can drop, reducing the required bandwidth to suit the conditions.

More timely information: Telepointer and drag operations will remain highly responsive, even if network conditions are lossy. The high priority ensures that they will be sent out whenever there is enough bandwidth to support them before other operations that are more latency tolerant. In the event of packet loss, telepointers will not be blocked while waiting for other, reliable information to be sent, and the most recent telepointer positions will be used. In the event of a burst loss or temporary network congestion, the most recent telepointer positions will be sent as soon as the traffic can get through.

Grab and drop messages will always be sent ahead of any other types of information, so users will always know as soon as possible when another user has picked up or let go of an object in the workspace. This will enable faster turn-taking and fewer conflicts over objects. By sequencing and ordering all drawing operations, the operations will appear to work similarly to how they work on the sender's machine, regardless of network effects.

Degrades gracefully: When bandwidth is constrained, the network library will send packets less frequently and aggregate more messages into each packet. This adds a small amount of latency, but maintains smoothness. In extremely constrained conditions, the send rate of telepointer and drag messages will be reduced, which will reduce the smoothness and/or accuracy of the streams, but the application will continue to function, as these messages are traded to ensure that the more critical messages get through.

Better overall usability: The user experienced will be greatly improved over an event-driven TCP model. Under ideal network conditions, this network design will perform the same as TCP-based implementations. However, as network conditions become worse, this networking design will continue to support a highly usable shared whiteboard, promising low latency, up-to-date telepointer and drag positions, and sustainability down to comparatively very low amounts of bandwidth.

6. DISCUSSION

This study of networking techniques used in multiplayer games produced two important results: it identified a set of networking techniques that are known to work over the Internet, and it identified some ideas for future research.

6.1 Lessons for practitioners

This paper presents a large number of networking techniques that are ready to use for building real-time distributed groupware. The techniques are presented at a high level rather than in detail, but the details are all found in the open source libraries for anyone needing more information. These techniques are particularly useful because they are known to be effective from years of experience in games and the game networking libraries serve as high quality examples of how to build these techniques. The other contribution to practitioners is that the principles that drive these techniques has led to a set of networking guidelines that are valuable to developers: limit bandwidth use, use appropriate reliability and ordering policies, and degrade gracefully. Additionally, by examining game networking libraries and describing their techniques, this paper shows that these libraries are candidates for use for building groupware applications.

6.1.1 Limit bandwidth use

Games use a variety of techniques for reducing the required bandwidth. The reason for this is that minimizing traffic is critical to performance. Some of the important bandwidth minimization principles that game networking libraries use are:

Enable hand-tuning: Make it easy for application programmers to efficiently encode their messages.

Avoid sending strings: Send primitives as minimum bit-length primitives, encode RPCs numerically, and use tables to encode strings as primitives. Avoid sending strings whenever possible.

Aggregate messages: When bandwidth is insufficient to send messages as they are generated, delay them slightly so that several can be aggregated into each packet. The small amount of added latency is a reasonable tradeoff for the increased efficiency.

Compress strings: It is useful to always attempt to compress strings using Huffman encoding or other lossless techniques. If the result is smaller, send it, and if not, send the original.

Imply parameters: Known bit-lengths of headers, ids, and values can imply parameter names. Never send these names, only values.

Avoid sending stale information: Take care to ensure that every message that is sent is useful to the receiver, and replace outgoing information with current values if updates are available.

6.1.2 Degrade gracefully

Applications should be prepared to cope adaptively with lower levels of network service. Some principles that appear in game networking libraries are:

Use adaptive flow control: Flow control using an adaptive window size is an effective technique for delivering timely information when bandwidth is sufficient and for driving aggregation, priority scheduling, and rate control policies when bandwidth is constrained.

Determine frequency ranges: By using minimum and maximum update frequencies, the application can degrade its traffic flows appropriately to cope with limited network resources.

Set priorities: Not all messages have the same level of importance, and close attention to assigning priority to messages is important when bandwidth is constrained.

Provide information to higher layers: Applications need to know when network conditions are poor, as there is a great deal that an application can also do to adapt to these conditions at runtime.

6.1.3 Use appropriate reliability and ordering

A major source of latency is due to blocking incoming messages that are out of order. This scenario needs to be avoided as often as possible using the following principles:

Do not order unnecessarily: Whenever possible, avoid ordering altogether. Just because a message needs to be delivered reliably does not imply that it has to be ordered as well.

Order independently: Games and groupware send a wide variety of different messages types, and ordered messages can be grouped into independently ordered streams.

Use sequenced policies: Sequenced policies can be particularly appropriate for interactive applications because they keep messages ordered, avoid blocking, and do not send or resend stale information. Unreliable traffic can be cheaply ordered with reliable information using an unreliable sequenced policy. Likewise, reliable sequenced policies are also useful because they do not block unnecessarily.

6.1.4 Use game networking libraries

The reason that network gaming libraries have become common is that it is not trivial to build good network code for games, and the same holds true for groupware. The poor performance seen in academic groupware applications is mostly the result of not devoting particular attention to network designs and optimizations rather than not having the capability to do so. An effective approach to building groupware that performs better over the Internet without the cumbersome task of writing efficient networking code is to start using existing game networking libraries. Games and groupware have many of the same requirements, and game network libraries meet the needs of groupware applications more closely than any other network implementations that are available. Additionally, they are robust and well-tested through real world use, and are likely to be more efficient to integrate and use than it is to develop a less efficient, less robust approach from scratch.

Eventually, groupware may have its own networking libraries that are better-suited to the needs of groupware. Until then, game networking libraries represent the best low-effort option for developers who want their applications to be used on the Internet.

6.2 Areas for future work

6.2.1 Improvements to techniques

Many of the techniques presented here offer opportunities for improvements. In particular, design aspects of some techniques vary among libraries, showing that determining the best method is not trivial and more work is required. Also, some of the techniques require considerable effort from the application programmer, and this effort can likely be reduced by automating some of these functions. Areas for future work include:

Automating encoding: Game libraries have made it easier to hand-tailor message encodings, but hand-tailoring still demands considerable attention from the application programmer. New methods that use the same principles and can approach the same level of efficiency but reduce the load on developers are needed. The techniques used in games can guide the development of easier to use techniques. Some ideas include adaptively determining minimum bit-length encodings for primitives, dynamically building string tables based on frequencies, and simpler programming interfaces for encoding shared information.

Better string compression: Strings in games and groupware are short, frequent, and most of the redundancy is among strings in separate messages rather than within individual strings. However, most lossless techniques for compressing strings assume that the strings are long and that the redundancy is contained within the string. Raknet's approach of dynamically generating new Huffman trees based on observed runtime character frequencies is a good example of a compression technique tailored to suit the characteristics of groupware, but there are certainly more opportunities available for more efficiently compressing strings.

Adaptive window sizes for groupware: Game networking libraries use a variety of techniques for adaptively controlling window sizes. It is unclear which of these techniques is best, and it seems to depend on the characteristics of the groupware traffic. Further work is needed to determine how to most effectively control adaptive window sizes in groupware, paying close attention to the bursty nature of interactive traffic, variable traffic patterns among applications, and diverse requirements for reliability among message types. In particular, how to best address the problem of controlling window size in unreliable traffic should be addressed.

Better aggregation policies: Aggregation in game networking libraries is driven by the window size only, but since aggregation adds latency, it should also be a function of the latency requirements of the information being sent. For example, aggregation should be avoided when low latency is required, and aggregation should be enabled when the added latency does not impact usability. QoS requirements and the network window size should both be considered by an effective aggregation policy.

Specialized delivery policies: TNL's quickest delivery and Raknet's sequenced ordering are examples of specialized delivery policies that are well-suited to the needs of games and groupware. There are undoubtedly more scenarios that are unique to

groupware traffic that could benefit from having policies tailored for the scenario. At a minimum, policies that address quality of experience issues beyond timeliness such as smoothness and accuracy are required. Additionally, policies should address the more complex aspects of collaboration such as degree of interest, focus and nimbus, and closely-coupled tasks would be beneficial.

6.2.2 Lessons for developing new techniques

Some of the techniques used in games take advantage of the characteristics of groupware traffic. These same characteristics can be applied to drive new groupware networking techniques. The characteristics of groupware traffic that can lead to new techniques are:

Streaming awareness information characteristics: Real-time groupware traffic consists mostly of streaming awareness information. This traffic is often bursty, messages are small, and messages have high frequencies. Game networking libraries have made use of these characteristics to enhance their aggregation policies, compression and encoding techniques, and scheduling techniques. Making use of these same characteristics can drive future techniques and optimizations to existing techniques.

Messages have diverse QoS requirements: Several of the techniques used in games are the direct result of observing the QoS requirements of game message types. The diverse QoS requirements of groupware can lead to further efficiency gains as the requirements are better understood and as policies and techniques are developed to take advantage of the requirements.

7. RELATED WORK

There is an abundance of work from various domains that is relevant to the techniques presented here. Many of the techniques used in games are familiar, although they have been tailored specifically to the needs of games.

A recent survey [5] of application layer networking techniques for groupware presents the most comprehensive collection of techniques from both groupware and from related domains such as multimedia, IP telephony, and distributed systems. Although some related techniques are mentioned, none of the techniques in the survey are the same as any of the techniques presented here.

Smed et al has surveyed techniques used in network games specifically [16]. This report reviewed published work from military simulations, networked virtual environments, and networked games. The survey describes aggregation and compression as useful techniques, but the specifics of the techniques are different from those we observed in games.

8. CONCLUSION

This paper presents the first analysis of game networking based on source code and documentation from real game networking libraries. The techniques and principles presented here are the result of significant real-world experience from the gaming industry in delivering a quality experience to users over the Internet. These techniques and principles are directly applicable to groupware, and applying them will drastically improve the performance of groupware when used under constrained network conditions. This work also has produced new directions for groupware networking research that are based on the current state-of-the-art in game networking.

Groupware aims to enable collaboration among people who are located all over the world. To do this, we must find ways of coping with the limitations of today's Internet. Network games have been successfully providing rich, real-time, interactive experiences to groups of people located all over the world for over a decade. By bringing the techniques that games use to groupware, we can further promote collaboration among individuals everywhere.

REFERENCES

[1] Begole, J., Rosson, M., Shaffer, C. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. Trans. Comput.-Hum. Interact. 6, 2 (Jun. 1999), Pages 95 - 132. 1999.

[2] Boyle, M., Greenberg, S. GroupLab Collabrary: A Toolkit for Multimedia Groupware. CSCW 2002 Workshop on Network Services for Groupware. 2002.

[3] Burridge, R. Java Shared Data Toolkit User Guide. Sun Microsystems, JavaSoft Division. Available from https://jsdt.dev.java.net. 2004.

[4] Chabert, A., Grossman, E., Jackson, L., Pietrowicz, S., & Seguin, C. Java object-sharing in Habanero. Communications of the ACM, 41(6), 69-76. 1998.

[5] Dyck, J. A Survey of Application Layer Networking Techniques for Real-time Distributed Groupware. University Of Saskatchewan Interaction Lab Tech Report, TR-2006-01. Available from http://hci.usask.ca. 2006.

[6] Enet. Enet Features and Architecture. Available at: http://enet.cubik.org/Features.html. 2003.

[7] GarageGames. Torque Network Library Design Fundamentals. TNL 1.5.0, 23 Feb 2005. Available at: http://opentnl.sourceforge.net/doxydocs/fundamentals.html

[8] Graham, T.C.N., Urnes, T., Nejabi, R. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. Proc. UIST 1996.

[9] Greenberg, S. and Roseman, M. Groupware Toolkits for Synchronous Work. Computer-Supported Cooperative Work (Trends in Software 7), Chapter 6, p135-168, John Wiley & Sons Ltd, ISBN 0471 96736 X. 258pp. 1999.

[10] Gutwin, C. Effects of Network Delay on Group Work in Shared Workspaces. Proc. ECSCW 2001.

[11] Hill, J., and Gutwin, C. The MAUI Toolkit: Groupware Widgets for Group Awareness. Computer-Supported Cooperative Work, 13 (5-6), 539-571. 2004.

[12] Marsic, I. Real-Time Collaboration in Heterogeneous Computing Environments. Proc. ITCC 2000, p222-227.

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

[14] Ruppel, J. Zoidcom 0.6.2 Manual. Available at: http://www.zoidcom.com. 2005.

[15] Scientific Toolworks, Inc. Understand for C++ User Guide and Reference Manual. Available at www.scitools.com. 2005.

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

[17] Xia, S., Sun, D., Sun, C., Chen, D., Shen, H. Leveraging single-user applications for multi-user collaboration: the CoWord approach. Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work, Nov 6-10, Chicago, IL USA.

[18] Zig. Zig 1.4.0 (Beta). Downloadable from http://zige.sourceforge.net. 2005.