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

Jeff Dyck[1], Carl Gutwin[1], T. C. Nicholas Graham[2], and David Pinelle[3]

[1]Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada

[2]Department of Computer Science
Queen's University
Kingston, ON, Canada

[3]Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, NV, USA

jeff.dyck@usask.ca, gutwin@cs.usask.ca, graham@cs.queensu.ca, pinelle@cs.unlv.edu

ABSTRACT

Networked games can provide groupware developers with important lessons in how to deal with real-world networking issues such as latency, limited bandwidth and packet loss. Games have similar demands and characteristics to groupware, but unlike the applications studied by academics, games have provided production-quality real-time interaction for many years. 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 achieve network performance. We examined five game libraries to find networking techniques that could benefit groupware; this paper presents the concepts most valuable to groupware developers, including techniques to deal with limited bandwidth, reliability, and latency. Some of the techniques have been previously reported in the networking literature; therefore, the contribution of this paper is to survey which techniques have been shown to work, over several years, and then to link these techniques to quality requirements specific to groupware. By adopting these techniques, groupware designers 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, QoS, Network Games, Groupware Performance.

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. Although many systems succeed in the research lab, network performance becomes a major problem when they move beyond the LAN and into real-world deployment. 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 the Internet, however, bandwidth is limited and packet loss is common, and current approaches to networking for CSCW applications quickly run into severe difficulty. Networking infrastructures have been improving, with the widespread presence of high-speed connections to the home. However, the increase of wireless and mobile platforms means that it is as important as ever for groupware applications to operate effectively under limited networking conditions. Poor networks cause problems for visual communication, coordination and anticipation of actions, and generally reduce the richness and quality of real-time collaboration [11,12]. Effective use of limited networks involves tradeoffs – e.g., between jitter and feedthrough time – and if real-time groupware is to succeed, it must find ways to manage these tradeoffs and optimize limited network resources.

There are several disciplines that could be used as a source of ideas for improving groupware networking (see [4] for a survey): for example, there is a great deal of research into techniques for improving video distribution, voice over IP, file transfer, distributed simulations, and large-scale collaborative virtual environments. It is not a given, however, that the techniques that have been introduced in these research communities will be valuable for real-time groupware. Often, the data used in these domains has very different characteristics and different quality of service (QoS) requirements than the messages used in groupware systems. Networking algorithms for Internet-based applications are designed to optimize application-specific performance attributes – for example, minimizing page load time when web browsing or maximizing throughput when serving video or sound. In contrast, the performance of real-time groupware is typically measured in terms of feedback and feedthrough times, and are characterized by workloads involving frequent, small bursts of information. Most networking algorithms have not been designed with the needs of groupware in mind, and are not necessarily applicable to this kind of distributed system.

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 that 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 to evaluate the applicability of these techniques.

In this paper, we present what is to our knowledge the first review and classification of the networking algorithms used in multiplayer computer games, and discuss the conditions under which these algorithms are applicable to the development of real-time groupware. Learning how games deal with networking is difficult since game companies do not generally publicize their techniques. Even previous surveys (e.g., [20]) have primarily been based on academic research papers. As a result, many of the novel aspects of game networking have remained, until now, undocumented in academic literature. Our review was possible due to the release of the source code for several game networking libraries. Our results are based on a comprehensive study of these libraries – TNL, Raknet, Zoidcom, Enet, and Zig – to find out how they achieve good network performance. Our classification reveals several networking techniques and principles that have not been considered before by groupware researchers, 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.

Our methodology in exploring the techniques was to delve into the source of the libraries themselves and reverse-engineer the precise designs of the mechanisms from the code. We then synthesized the specific implementations into general techniques used across the libraries. Finally, we classified the techniques according to the problems they address, allowing us to link the techniques to specific issues in groupware development.

The networking techniques that we report are not themselves new; our contribution is their classification and evaluation as techniques for improving real-time groupware. Game libraries represent an implicit evaluation of networking techniques: the techniques that we present here are in the game libraries because they have been found to work over several iterations of individual systems. In addition, this implicit evaluation has been carried out with systems whose communication style, message characteristics, and QoS requirements are very similar to the types of real-time groupware that the CSCW community is interested in building.

This paper makes three contributions:

- We collect, synthesize, and present a set of networking techniques that have demonstrated effectiveness in improving the network performance of real-time distributed groupware – techniques which few groupware designers or members of the CSCW community are familiar with;
- We demonstrate how networking techniques have addressed the fundamental characteristics of real-time groupware: small messages, bursty traffic, and diverse QoS requirements. These examples can be used as starting points for tailoring other techniques to the needs of real-time distributed systems;
- We identify general principles that underlie the networking approach of these libraries (limiting bandwidth, degrading gracefully, and using appropriate reliability and ordering) that have not previously been discussed in the groupware literature, and that can be an important part of a framework for improving groupware performance more generally.

In the remainder of the paper, we describe the state of the art in networking in academic groupware applications, then present the techniques and underlying principles drawn from the five libraries under study, and finally show through a detailed example how these techniques can be applied to groupware.

2. NETWORKING IN GROUPWARE

Real-time distributed groupware communicates by sending messages over a network. These messages indicate the actions and changes made by the people using the system, and can include awareness messages (e.g., telepointer motion), explicit communication (e.g., chat, voice), operations and transactions on the data model, and session events. Awareness messages are the most common (e.g., since telepointers move frequently), but other messages types are often more important (e.g., creation or deletion operations). The way that the underlying groupware system composes, organizes, and sends these various message types has a dramatic impact on network performance and on the user's experience of the collaborative activity.

One particular approach to networking can be seen in many of the groupware systems and toolkits that have appeared in the CSCW community (such as Groupkit [19], Disciple [17], Java Shared Data Toolkit [2], Collabrary [1], MAUI [15], CoWord and CoPowerPoint [21], Clock [7], and Habanero [3]). These 'academic groupware' systems all use what we call *event-driven TCP*, an approach that works reasonably well on a LAN, but which is completely unsuitable for use on the Internet.

In event-driven TCP, applications trigger events (e.g., keyboard, mouse, or transaction events) that result in messages to be sent over the network. Each message is immediately put into a TCP packet and sent; therefore, packet rate is governed by the application's event model. Event-driven TCP is relatively simple to implement, and performs well when events are rare and guaranteed delivery is required (e.g., in a chat application). Even for applications that send awareness messages (such as telepointer positions) much more frequently, event-driven TCP can work on a high-bandwidth LAN; however, the approach does not work well when groupware applications are used over the Internet.

As an example, consider a system that sends text-based or object-based telepointer update messages using event-driven TCP (e.g., Groupkit, MAUI, or Collaborary). Messages will be generated and sent on mouse interrupts, which produces packet rates between 30 and 60 updates per second. Each message is approximately 50 bytes, and is sent in a separate TCP/IP packet, which adds an additional 40-byte header. This produces an upload data rate around 32Kbps for every user that the messages need to be sent to. In a four-user shared whiteboard session with a peer-to-peer unicast network architecture, this results in an upload bandwidth requirement of 128Kbps, which is more than many 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 begins to lag behind the source. Even when there is sufficient bandwidth, however, event-driven TCP causes performance problems because of its reliability mechanism: when TCP packets are lost, incoming packets are blocked at the receiver until the lost packet is retransmitted. TCP resends lost messages regardless of whether they are still useful to the application, and the receiver cannot process other messages (even if they are unrelated to the lost packet) until the resent message arrives. In a shared whiteboard application, for example, a lost telepointer update message means that all messages are blocked, including tool selection and chat messages, until the telepointer message arrives. Upon arrival, the waiting messages are processed all at once, causing the telepointer to jump across the screen in an unnatural way. These network problems make it difficult for collaborators to follow telepointer motion, to coordinate actions, and to recognize gestures [12].

Although not all toolkits work in exactly this way, and although some of these systems offer other networking options (e.g. JSDT supports the lightweight reliable multicast protocol), the default in most cases is to use event-driven TCP.

More advanced networking algorithms trade off several metrics that capture aspects of users' experience [6]. *Feedback time* is the time from a user performing an action to seeing the results of that action. *Feedthrough time* is the time from a user performing an action to other users' seeing its consequences. *Jitter* represents variance in feedback and feedthrough time. *Fidelity* represents the degree to which different participants' views agree. These metrics often trade off against each other. In broadcast video, for example, higher feedthrough time is often acceptable in order to reduce jitter, whereas in teleconferencing, higher jitter is acceptable in order to gain immediacy of interaction. Groupware applications often sacrifice short-term fidelity in order to improve feedback time [21]. By examining networking libraries, we have been able to determine which algorithms are effective in optimizing one or more of these quality attributes.

3. STUDY METHODS

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, are recommended for use on game development web sites, and are in use in existing games. We read through all of the library 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++ (www.scitools.com). 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 solve, and evaluated how effective they would be for groupware based on how well they deal with 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 CSCW community. Finally, we analyzed what game and traffic factors led to the development of these techniques. The libraries that we studied were TNL, Raknet, Zoidcom, Enet, and Zig.

3.1 Game networking libraries

TNL (opentnl.sourceforge.net) 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 visibility-based traffic filtering technique used in indoor FPS games could not be used. This meant that Tribes had to send unprecedented amounts of information over the 28.8 Kbps modems it was designed to work with, and as a result, much network optimization was needed to accommodate its design. The Tribes network code was further improved for its sequel, Tribes 2, and after Tribes 2 was completed, the networking code was packaged by its developers into a stand-alone library called TNL, which was offered commercially and has been used in many successful independent and commercially developed games.

Raknet (www.rakkarsoft.com) 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 with free commercial licenses.

Zoidcom (www.zoidcom.com) is a full-featured commercial game networking library that first appeared as a beta release in 2004 and is still in development. The full source code for the library is not available, but the C header files are included and it is well-documented, revealing several performance-enhancing techniques that are relevant to this study. No commercial games using Zoidcom have been released, but it has been used in several independent game projects.

Enet (enet.cubik.org) 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 (zige.sourceforge.net) 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.

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.

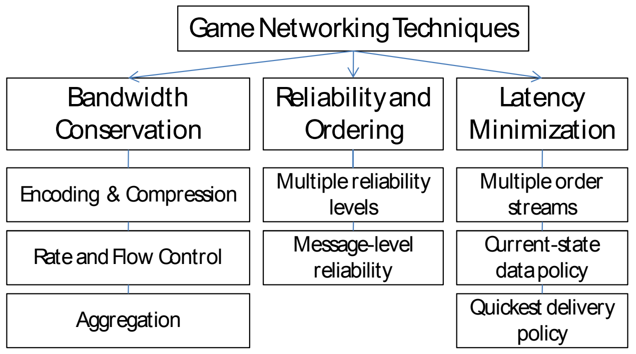


Figure 1. Summary of game networking techniques

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 reducing message size. Game networking libraries provide mechanisms for efficient encoding of information, provide lookup tables for common data, and provide compression techniques for other types of data.

Minimum bit-length encodings. Several libraries (e.g., Raknet, Zoidcom, and Zig) provide functions for sending primitive values with the smallest possible number of bits. This means that primitives are never sent as strings, that only the primitives are sent across the network (rather than the field names), and that each value uses the most efficient representation.

TNL further improves this process by allowing the programmer to specify the number of bits to use (see Figure 2). For example, an integer with a range of 0-5 can be sent using 3 bits in TNL; other libraries would require 8 bits, since a byte is the smallest representation they support. The TNL method can achieve optimal bit-length representations, but requires that the application programmer calculate the number of bits needed for each value.

String lookup tables. Strings are a costly data type to send since each character requires a full byte. Therefore, a mechanism for encoding frequent strings as a numeric id can be an effective optimization. In order to do this, the system must know which strings will be sent by the application, which depends on runtime parameters such as usernames, user-configured chat hotkeys, and tasks that are performed within the game.

TNL uses a string lookup table that is generated dynamically at runtime. Any strings that are likely to be repeated are added to the lookup table; these additions are communicated to other clients, and then the string can be sent as a table entry. This dictionary-based approach is well-suited to games since they often send the same strings repeatedly.

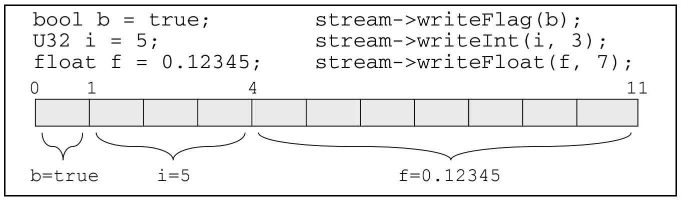
RPC lookup tables. Remote procedure calls (RPCs) are a signal to other clients to execute a method or function. This requires that the function's signature and parameter list be sent over the network, sometimes repeatedly. TNL uses the idea of a lookup table (as discussed above) to encode RPCs more efficiently. Information about an RPC is registered with other clients at runtime as a dictionary entry with a unique numeric ID. RPCs can then be sent as an ID (that allows lookup of signature and parameter information) followed by minimum bit-length representation for the parameter values (see Figure 3).

Lossless compression. For any message that cannot be reduced using the above techniques, game libraries apply one of several lossless compression schemes. The type of compression, and the objects on which it is applied, vary widely across the libraries.

Two common approaches involve compressing strings with Huffman coding, and compressing entire packet payloads. TNL and Raknet take the former approach: they apply Huffman coding on a per-string basis, with a length check to ensure that the technique is only used when it is likely to perform well.

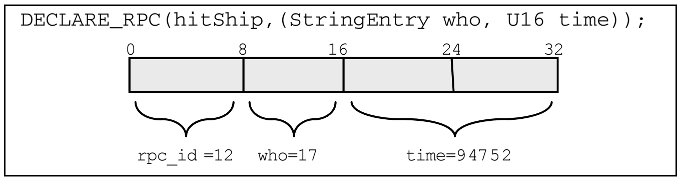
One optimization introduced by Raknet is maintaining frequency charts for the occurrences of characters in strings, allowing the Huffman tree to be rebuilt at any time, which ensures that the encoding fits the application. This is useful since the character frequencies in strings may vary based on the application type, runtime parameters, and the players' current tasks.

The second approach, seen in Zig, is to compress entire packet payloads rather than individual strings. This can provide better performance than per-string compression, since there may be more repetition in a full packet than in a single string. Zig uses the BZip2 algorithm, again employing a check to make sure that the technique is only applied when it will actually result in a shorter message. Zig also allows programmers to specify the smallest packet payload to compress, which avoids testing compression performance on small packets.



```
bool b = true;        stream->writeFlag(b);
U32 i = 5;            stream->writeInt(i, 3);
float f = 0.12345;    stream->writeFloat(f, 7);
```

Figure 2: Sending values with minimal bit lengths in TNL



```
DECLARE_RPC(hitShip,(StringEntry who, U16 time));
```

Figure 3. RPCs as sent by TNL. The ID is encoded as 8 bits; the first parameter is encoded using a string lookup, and the second parameter is encoded using a minimal representation

Each of these techniques can help in groupware by reducing latency in cases where bandwidth is constrained. This can in turn help with feedback and feedthrough time. In deployments where bandwidth is freely available, these techniques can actually harm performance, as the processing overhead of reducing message size can outweigh the saved transmission time.

4.1.2 Rate and flow control

Exceeding bandwidth limits causes severe usability problems [14]. To prevent this from happening, game networking libraries make use of three main techniques: bandwidth monitoring, static rate control, and adaptive flow control. These techniques allow developers to trade off between fidelity versus feedthrough time, by eliding updates when bandwidth is scarce.

Bandwidth monitoring. Game libraries provide methods for monitoring the amount of bandwidth that is being used, so that the system can make decisions about when to change to lower-traffic communication strategies. All libraries can report information such as the current incoming and outgoing available bandwidth (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 build in well-informed adaptation 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. In some cases, game programmers know beforehand what minimum send rates are acceptable for their games. TNL allows the programmer to set 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 96ms packet interval (~10 packets/second).

Object rate control. Libraries that support object replication (TNL, Raknet, Zig, and Zoidcom), use a separate rate control technique for objects. The library lets the application programmer specify a maximum (and minimum with Zoidcom) update rate for each replicated object. This allows objects with different characteristics to have different update rates; in addition, each object's rate adapts to current network conditions, varying within the ranges specified by the programmer.

Adaptive flow control. Flow control is a method for limiting bandwidth by monitoring and adapting to the number of packets currently in the network (called the window size). This requires that acknowledgements be sent to inform the sender that a packet is out of the window. This works well for reliable information since acknowledgements are being sent anyway; however, games send much of their traffic unreliably, so they must use alternate mechanisms for controlling flow. Two main approaches are seen:

- *Receiver-driven control*. TNL uses an adaptive window size to control 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 solely for the purpose of adapting the window size.
- *Probability-based control*. Enet and Raknet do not send any extra acknowledgment, as this adds traffic. Rather, the window size (based on acknowledgments for reliable packets) is used as an indication of how many unreliable packets must be dropped. Enet uses a 'drop probability' that increases as the window size grows, whereas Raknet simply drops all unreliable messages when the window is full. In both cases, unreliable messages are held back to make room for higher-priority reliable messages.

4.1.3 Aggregation

Messages sent by games are often small, and so multiple messages can often be aggregated into a single packet. This saves space consumed by packet headers, and reduces the resources required to process packets along the way. Aggregation works by filling packets from an outgoing send queue, and messages are aggregated until one of three conditions is met: the maximum packet size is reached, all of the messages in the queue are sent, or the timer signal for sending a packet is received. This policy can increase latency, since messages are delayed until one of the conditions is met – but can actually improve feedthrough time when bandwidth is limited. All of the libraries we studied except Enet support aggregation, and allow the technique to be applied to both regular messages and object updates:

Send queues. Outgoing messages are written to a send queue, and packets are then filled 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. Updates for replicated objects occur in frames, which aggregate data for several objects. The frames consist of a subset of the replicated objects (based on object send rates as described above); each frame is written to the send buffer and aggregated with other outgoing traffic. Most libraries automatically handle framing, but Zig requires the application programmer to define exactly what is included in each frame. This approach requires more effort from the developer, but is also more flexible.

4.1.4 Priority

The number of messages in the send queue often exceeds what can be aggregated into a single packet. This can happen when there are bursts of messages, when there is limited bandwidth, or when there are many users in the system. One way that game libraries deal with these situations is to mark messages with a priority that indicates the order in which they should be taken from the send queue. This means that latency-sensitive messages are sent earlier, and that latency-tolerant messages are sent later (or even dropped from the send queue if their information becomes stale). Priority queues provide programmers with a high-level mechanism for specifying the relative temporal importance of different messages; e.g., jitter-sensitive message streams such as voice data may benefit from higher priority. 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 bandwidth limits are reached – at which point low priority unreliable messages are dropped, and low-priority reliable messages must wait.

Numerically assigned but manual. Raknet's approach allows 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 messages of different priority.

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 messages are sent; as available bandwidth decreases, the probability of dropping unreliable messages increases until an equilibrium state is reached.

RPCs or frames first. Zig provides two priority levels for RPCs. When priority is high, RPCs are sent before any additional frames, even if a frame needs to be dropped in order to send the RPC. Otherwise, Zig 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 gives a message top priority – it is sent in every packet until an acknowledgment is received (see Section 4.3.3).

4.2 Low-cost reliability and ordering

As described above, there are severe performance penalties for using protocols such as TCP that provide guaranteed message delivery and ordering. However, game messages have variable requirements for reliability and ordering, and game libraries exploit this variability by providing several mechanisms that provide different levels of service. Games make use of two main techniques – they offer several different combinations of ordering and reliability, and they manage these at a message level rather than at a packet level.

4.2.1 Several levels of reliability

Different game messages require different combinations of reliability and ordering. In general, highly reliable, totally ordered messaging provides high fidelity – the receiver experiences the same operations as the sender. Relaxing reliability and ordering weakens fidelity, but brings improved feedback time. The game libraries we examined provide five distinct QoS options for delivery and ordering:

- *Reliable ordered* protocols are implemented over UDP by all of the libraries. The reliable UDP implementations follow the design of TCP, but have some key differences. 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 independent discrete events, 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 sender's resend queue 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.

The most interesting of these policies is the 'reliable sequenced' approach. Using this policy, only the last update to a stream of messages is sent reliably. This policy offers some of the latency and bandwidth advantages of unreliable delivery, but ensures that the final update will arrive. This is useful for the bursts of awareness messages that occur in games and groupware: for example, a telepointer's final position is the most important of a sequence of 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 that all messages in the packet be treated equally, and performance can suffer as a result. For example, consider a packet that contains two messages: a movement message with an 'unreliable sequenced' delivery policy, and a weapon-fire message with a 'reliable unordered' policy. Since one of the messages requires reliable delivery, packet-level reliability requires that the system 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.

In contrast, TNL implements message-level reliability. This is implemented by adding a lightweight reliability header to every message, and removing the reliability portion of the packet header. When the packet is lost in the scenario given above, message-level reliability means that the movement message will simply be dropped (since delivery is not required), and that the fire message will be resent with the next packet. This approach is much more efficient in low bandwidth and lossy conditions.

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 backs up. Last, time-critical messages may be delayed by waiting in the send queue or through packet loss. This kind of latency negatively impacts both feedthrough time and jitter. The game libraries we studied have techniques for reducing latency in all 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; when this happens, all subsequent ordered messages are blocked at the receiver. 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.

Since there are many message types each with different QoS requirements, it is common to have independently ordered messages. For example, ordered chat messages and ordered firing messages do not need to share an ordering, since the order of firing and chat are independent. In this case, a lost chat message should not block firing messages.

Game libraries provide a mechanism for independent ordering; for example, Raknet provides 32 independent streams that are 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 even though 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 into a single packet.

4.3.2 '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. In some cases, the information in these messages can become stale, and therefore will be useless when sent to the receiver.

The 'reliable sequenced' policy used in Raknet partially addresses the problem of staleness in the reliable queue. Rather than resending reliable data that is stale, it drops the message from the resend queue when it receives an acknowledgement that a more recent update has arrived at the receiver.

However, this policy does not deal with stale information in the regular send queue. To handle this situation, TNL adds a QoS level that they call 'current state data', which ensures that only the most recent update is sent. Before a message is sent, a check is performed to ensure that there is no updated value available. If there is an updated value, the queued message 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 'Quickest delivery' policy

Some messages need to arrive before all other information, either because they are highly latency-sensitive (e.g., a hit in a first-person shooter game), or because subsequent messages are dependent on the earlier message (e.g., a new string table entry).

To support this type of delivery, TNL includes a 'quickest delivery' policy that works by including a message in every outgoing packet until an acknowledgement is received (Figure 4). This guarantees the soonest possible delivery of a message – since in the event of a packet loss or a delayed packet, the message always appears in the next packet as well. This policy trades off bandwidth efficiency for minimized latency, since it sends information redundantly, but the penalty is usually not a large one since most game messages are small. However, this inefficiency means that the programmer must not overuse the policy.

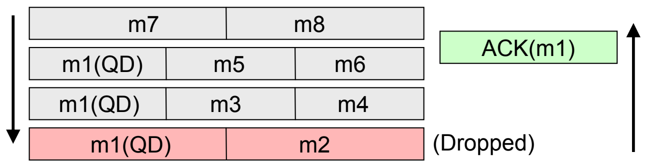


Figure 4: Quickest delivery policy sends a message with every packet until it is acknowledged, guaranteeing first arrival.

5. SHARED WHITEBOARD SCENARIO

The techniques used in game libraries are directly applicable to real-time groupware. As an example, we consider applying the techniques described above to a shared whiteboard application. Through this example, we show how these techniques address the different issues considered in Section 4: bandwidth optimization, low-cost reliability and ordering, and latency minimization. This section describes how the techniques can be applied, 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 type. There are different QoS requirements for each of the message types (summarized in Table 1).

- 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 have low requirements on feedthrough time, and so they can be given a low priority, which will allow them to be delayed in favor of more latency-sensitive information.
- Telepointers need low feedthrough time, but do not need reliable or ordered delivery. Since telepointers are not dependent on any other messages, they are sent on their own ordering channel.
- The message types related to drawing operations need to be ordered with one another, since the effects 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 operations 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 must 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 be updated in real time, so each character is sent as it is typed. In order to provide feedthrough that preserves the sense of typing, these messages are given a medium priority to reduce latency.

Table 1: QoS requirements for whiteboard message types.

	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

Efficient message encoding. Chat, session information, tool selection, grabs, and drops are all discrete events and are therefore best modeled as RPCs. For each of these, we register an RPC and specify bit lengths for parameters where appropriate. For example, (x,y) coordinates should be encoded such that their limits do not exceed the maximum workspace dimensions, and tool selections should be encoded so that the tool parameter does not exceed the total number of tools available to the system. 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 character 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 are 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 Implications for performance

Using the techniques described above in the implementation of the shared whiteboard will result in several improvements to the application's performance on the real-world Internet.

Reduced bandwidth. 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 small fraction of what is used by event-driven TCP. Under ideal network conditions, our 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 message per packet). This is one-third the bandwidth used by the event-driven TCP example described in Section 2.

The bandwidth differences are more dramatic when resources become constrained – in these situations, the flow controller will begin to aggregate telepointers, resulting in a substantial bandwidth saving. For example, aggregating 3 telepointers into each packet would decrease bandwidth usage to 5Kbps per client, through the reduction in packet headers. In addition, when 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.

Better handling of time-sensitive information. Telepointer and drag operations will remain highly responsive, even in lossy network conditions. Their high priority ensures that they will be sent out (when there is enough bandwidth) before other operations that are more latency tolerant. In the event of packet loss, telepointers will not be blocked waiting for reliable information to be sent, and the most recent telepointer positions will be used. In the event of burst loss or temporary network congestion, the most recent telepointer positions will be sent as soon as possible.

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.

Graceful degradation. When bandwidth is constrained, the system will send packets less frequently and aggregate more messages into each packet. This adds a small amount of latency, but maintains smoothness. In extreme conditions, the send rate of telepointer and drag messages will be reduced, reducing the smoothness and or accuracy of the streams; but the application will continue to function, since the awareness messages will be held back whenever more critical messages are sent.

Better overall usability. The user experience will be greatly improved compared with an event-driven TCP model. Under ideal network conditions, this network design will perform the same as TCP-based implementations. However, as network conditions worsen, our networking design will continue to support a highly usable shared whiteboard, with low latency, up-to-date telepointer and drag positions, and sustainable interaction, even in situations of extremely low bandwidth.

6. DISCUSSION

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, but the concepts are reasonably simple, and implementation details can be found in the code of the libraries. The techniques we discuss above are known to be effective through years of evaluation in games, and the game networking libraries can serve as high-quality examples of how to build these techniques.

In the remainder of the paper, we discuss some of the underlying principles that can be used to guide the design of groupware network infrastructures more generally, suggest that groupware developers start using game network libraries, and outline several areas for future work.

6.1 Underlying principles

Our examination of game networking techniques has identified several underlying principles that can be used more generally to guide the design of real-time distributed groupware. Our experience suggests that groupware developers should be thinking about limiting bandwidth, using appropriate reliability and ordering policies, and providing mechanisms for graceful degradation of service when resources are limited.

6.1.1 Limit bandwidth use

In environments where bandwidth is limited, feedthrough time and jitter become major problems for groupware usability. These problems are becoming increasingly important as groupware is used in mobile settings with low capacity networks. Game libraries use a variety of techniques for reducing bandwidth requirements, based on the following principles:

- *Use efficient representations*: send primitives as minimum bit-length primitives, encode RPCs numerically, imply field and parameter names through order, use tables to encode strings, and avoid sending strings whenever possible.
- *Aggregate messages*: when bandwidth is insufficient to send messages as they are generated, aggregate several into each packet. The small amount of added latency is a reasonable tradeoff for the increased efficiency.
- *Compress*: It is useful to always attempt to compress strings or string-heavy payloads using a lossless techniques; if the result is smaller, send it, and if not, send the original.
- *Do not send stale information*: ensure that every message will be 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. Principles from the game network libraries allow application programmers to choose between degrading jitter, fidelity and feedthrough time:

- *Use adaptive flow control*: flow control with an adaptive window size allows timely delivery when bandwidth is sufficient, and supports aggregation, priority scheduling, and rate control policies when bandwidth is constrained.
- *Determine frequency ranges*: minimum and maximum update frequencies allow the application to reduce traffic flows appropriately to cope with limited network resources.
- *Set priorities*: not all messages have the same level of importance, and careful message prioritization can have large effects on usability when bandwidth is constrained.

- *Provide network information to the application*: application programmers can make good decisions about how to deal with poor network conditions, and so network monitoring information should be supplied by the toolkit layer.

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*: avoid ordering whenever possible – reliability requirements do not always imply that ordering is also required.
- *Order independently:* both games and groupware send a wide variety of 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.

6.2 Use game networking libraries

The reason that network gaming libraries have become common is that it is difficult to build good network code, and the same holds true for groupware. The prevalence of simplistic networking models like event-driven TCP is partly a result of the difficulty of designing and implementing for performance. Therefore, one way to build better-performing groupware, without requiring that groupware developers begin writing efficient networking code, is to start using existing game libraries for groupware. 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.

6.3 Areas for future work

6.3.1 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 customize message encodings, but hand-tailoring still demands considerable attention from the programmer. New methods are needed that use the same principles but reduce the load on developers. 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. Work has begun on this aspect [10], but more could be done.

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 opportunities available for more efficient inter-message compression of 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 best 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, further work should be done on the problem of controlling window sizes for mixed-reliability traffic.

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 that address the more complex aspects of collaboration such as degree of interest, focus and nimbus, and closely-coupled tasks would be beneficial.

6.3.2 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 new groupware networking techniques. The characteristics that can lead to new techniques are:

Streaming awareness information. Real-time groupware traffic consists mostly of awareness information. These messages are bursty, small, and frequent. 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 messages. 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 a great deal of research from a variety of domains that is relevant to the techniques, the principles, and the issues presented here. First, many of the basic ideas for the techniques presented here have originally appeared in research from the multimedia, IP telephony, distributed systems, and collaborative virtual environments research communities. A recent survey of application layer networking techniques for groupware presents an overview of techniques from these areas [4]. As mentioned earlier, these approaches are not always applicable to the specific needs and data that is seen in groupware, but they provide starting points for groupware-specific techniques.

Second, other surveys of games and game networking present additional techniques that can be valuable. In particular, Smed and colleagues [20] have reviewed a variety of published academic work in military simulations, collaborative virtual environments, and networked games. These surveys introduce other techniques such as dead-reckoning that have been used in games and distributed simulations, and also mention some of the network techniques seen here (aggregation and compression, although the details are different to those of the game libraries).

Finally, a core of groupware researchers is also looking at performance issues, including performance aspects of groupware distribution architectures [14,16], scalability [13], QoS [8,9], generalized compression [10], and interface techniques for hiding delay [11,12]. This work offers a valuable complement to the present study; although the performance models are often based on abstractions of the real network environment, and thus do not have sufficient detail to capture the issues addressed in our study, prior research provides a high-level context of toolkits and distribution architectures in which our techniques can be used.

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 real-time distributed groupware, and applying them will dramatically 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 many years. By bringing game network techniques and principles to groupware, we can both improve groupware performance and make groupware applications feasible across a much wider range of network situations and conditions.

REFERENCES

[1] Boyle, M., and Greenberg, S., Rapidly Prototyping Multimedia Groupware. *Proc. Conference on Distributed Multimedia Systems (DMS'05)*, Banff, Canada, 2005.

[2] Burridge, R. *Java Shared Data Toolkit User Guide*. Sun Microsystems, 2004. Available from jsdt.dev.java.net.

[3] Chabert, A., Grossman, E., Jackson, L., Pietrowicz, S., and Seguin, C., Java Object Sharing in Habanero. *CACM*, 41(6), 69-76, 1998.

[4] Dyck, J. *A Survey of Application-Layer Networking Techniques for Real-time Distributed Groupware*. Technical Report HCI-TR-06-06, University of Saskatchewan, available at hci.usask.ca.

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

[6] Fletcher, R.D.S., Graham, T.C.N. and Wolfe, C., Plug-replaceable Consistency Maintenance for Multiplayer Games, *Proc. NetGames 2006,* 34-37.

[7] Graham, T.C.N., Urnes, T., and Nejabi, R. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. *Proc. UIST 1996*, 1-10.

[8] Graham, T.C.N., Phillips, W.G. and Wolfe, C., Quality Analysis of Distribution Architectures for Synchronous Groupware, *Proc. CollaborateCom,* 2006, 1-9.

[9] Greenhalgh, C., Benford, S., and Reynard, G., A QoS architecture for collaborative virtual environments, *Proc. ACM Multimedia 1999*, 121-130.

[10] Gutwin, C., Fedak, C., Watson, M., Bell, T., and Dyck, J., Improving Network Efficiency in Real-Time Groupware with General Message Compression, *Proc. CSCW 2006*, 119-128.

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

[12] Gutwin, C. Effects of Network Delay on Group Work in Shared Workspaces. *Proc. ECSCW 2001*, 299-318.

[13] Hall, R., Mathur, A., Jahanian, F., Prakash, A., Rasmussen, C., Corona: A Communication Service for Scalable, Reliable Group Collaboration Systems, *Proc. CSCW 1996*, 140-149.

[14] Hayne, S. and Pendergast, M., Experiences with Object-Oriented Group Support Software Development, *IBM Systems Journal*, 34(1), 1995, 96-120.

[15] Hill, J., Gutwin, C., The MAUI Toolkit: Groupware Widgets for Group Awareness. *CSCW*, 13 (5-6), 539-571, 2004.

[16] Junuzovic, S, and Dewan, P., Response times in N-user replicated, centralized, and proximity-based hybrid collaboration architectures, *Proc. CSCW 2006*, 129-138.

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

[18] Reynard, G., Benford, S., Greenhalgh, C., and Heath, C., Awareness driven video quality of service in collaborative virtual environments, *Proc. CHI 1998*, 464-471.

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

[20] Smed, J., Kaukoranta, K., and Hakonen, H. Aspects of Networking in Multiplayer Computer Games, *The Electronic Library*, 20(2), 2002, 87–97.

[21] Xia, S., Sun, D., Sun, C., Chen, D., Shen, H., Leveraging single-user applications for multi-user collaboration: the CoWord approach. *Proc. CSCW 2004*, 162-171.