ENSC 835: HIGH-PERFORMANCE NETWORKS

Simulation and Performance Analysis of Fully Meshed Network Games

Fall 2003

FINAL PROJECT

David Mikulec
http://www.sfu.ca/~dmikulec
dmikulec@sfu.ca

1. ABSTRACT

Multi-player network games are enjoyed by millions of computer gamers. This paper analyzes and simulates the performance of a TCP implementation of the peer-to-peer architecture to connect remote gamers. It shows that a simple application-layer protocol running over TCP has promise for being able to handle large peer-to-peer games – on the order of 10-20 peers over current ADSL connections – despite its relative simplicity compared to complex UDP-based protocols currently used by most games.

2. Introduction

In the 1990's gamers got their first taste of mainstream multi-player networked PC games, from first-person shooters such as Quake (id Software, 1996), to real-time strategy games including Starcraft (Blizzard Entertainment, 1997) to Massively Multiplayer Online Role Playing Games (MMORPG) like Ultima Online (Origin Systems, 1997). The early 21st century witnessed the launch of the Sony PlayStation2 network adapter kit and the Microsoft Xbox Live service, which widened the audience for networked games by millions of users. Recently the dominant video game maker Electronic Arts announced that all future sports games (as well as many other titles) would feature online play. Given the explosive growth of networked multi-player video games, a simulation and analysis of their network performance is of significant interest. This paper, being the result of a project assignment in a course on High-Performance Networking, will focus on the performance of those first-person shooters, sports games and, real-time strategy games that place the highest demands on network infrastructure in terms of bandwidth and delay requirements.

2.1. Architectural Models

The key design decision faced by networked game programmers is which network architecture to follow: client-server (e.g. Quake) or peer-to-peer (Age of Empires). Figure 1 illustrates these two models, with client-server on the left (each game client communicates solely with the central server), and peer-to-peer on the right (each game client communicates directly with all other peers). For the purposes of this paper, when the term peer-to-peer is used, it is implied that it is in the fully-meshed sense – that is each peer exchanges data with all other remote peers involved in the same game (as opposed to large peer-to-peer networks like Gnutella where thousands of peers maintain connections to only a few remote peers). This paper will focus on the peer-to-peer model, for reasons that will be outlined after a brief discussion of the pros and cons of each model in the following sections.

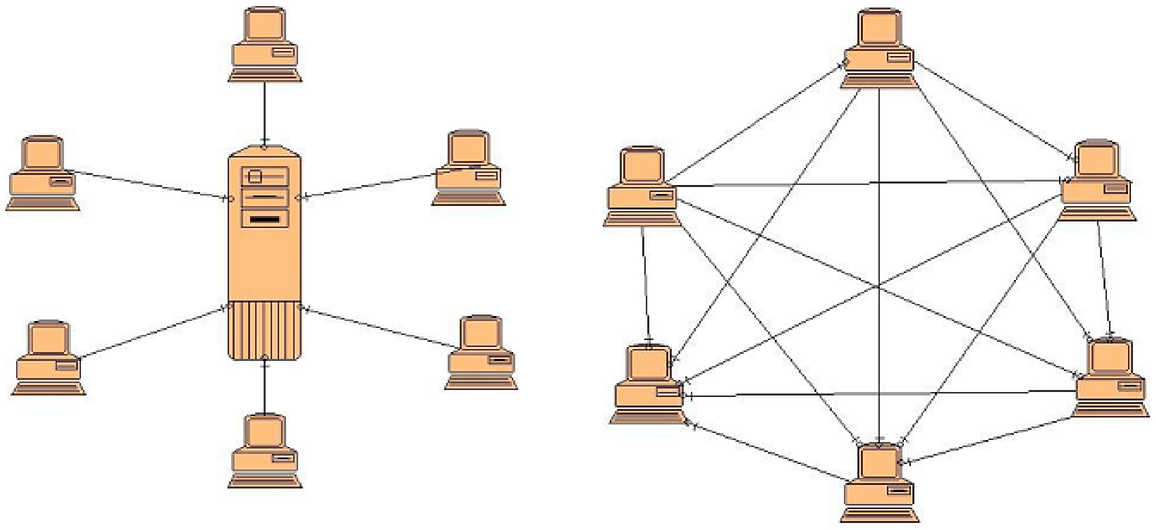

Figure 1. Client Server and Peer-to-Peer architectures

2.1.1. Client-Server

Client-server games have two main advantages over peer-to-peer games: first, they consume far less upstream bandwidth because clients only need to send one copy of information to the server as opposed to N remote clients, and conserve some downstream bandwidth (due to less per-packet overhead) as the server can aggregate information from all remote clients; second, one central server has the authoritative view of the "world", which allows the game to continue without interruption despite some dropped packets – compromising only quality of play for the client whose packets (incoming or outgoing) are dropped.

2.1.2. Peer-to-Peer

Peer-to-peer games, on the other hand, have distinct advantages due to the lack of a server: no separate server code base needs to be developed, maintained and distributed; there is no need to monitor or provide vast (and currently expensive) bandwidth for central servers (run by the developer or a 3rd party contractor – although low bandwidth servers are often provided to facilitate users to find others interested in playing the same game); there is no central point of failure, only individual peers can lose their connection (many games can continue with fewer players by eliminating the disconnected player entirely, or with the computer taking over play for that player); and most importantly the end-to-end delay is lower as it is simply based on the connection between the peers, rather than client-to-server-to-client. While the other factors mentioned are all desirable, low delay is the key because it directly affects the quality of play as there is a strong correlation between lower delay and better gameplay [5].

2.1.3. P-2-P as the Way of the Future

Both client-server and peer-to-peer architectures offer benefits at present, but in the future peer-to-peer networks are likely to dominate thanks to the laws of physics. In typical network games amongst users spread out across continental North America, the speed of light places a lower bound on end-to-end delay between any two computers on the order of 10ms (i.e. distances on the order of 3000km divided by the speed of light 300,000km/s; globally the lower bound is on the order of 100ms). From extensive personal experience with online games, broadband users with good connections currently enjoy delays on the order of 50ms to a broad range of hosts in North America; so there remains not even an order of magnitude of possible improvement in latency before physical limits are reached. Bandwidth on the other hand is available in much higher quantities than the typical 1 ~ 2 Mbps range most broadband users have today; if Moore's law holds we can expect users to have multi-megabit per second connections in the near future. Given that bandwidth is the most significant technical impediment to selecting a peer-to-peer architecture for large fully meshed network applications, it is likely that with increasingly high bandwidth and reliable connections, peer-to-peer networks will be the obvious choice in the future to maximize performance by reducing delay, hence they are the focus of this paper.

The purpose of this project is to investigate the performance of a general fully meshed peer-to-peer gaming network. While the network code for popular games can be quite detailed, and highly optimized for a specific game, this paper seeks not to focus on any particular game, but rather to gain an understanding of the issues common to many games. By focusing on the common networking aspects, the goal is to obtain results applicable to a broad range of games, and even to other collaborative networked applications with similar performance requirements that have yet to be thought of.

2.2. Choice of simulation tool: ns-2

The tool used in this project is the ns-2 version 2.26 network simulation tool, available freely from the Information Sciences Institute (ISI) at the University of Southern California. The tool interprets Tcl programs, allowing a user to set up topologies of nodes connected by links of varying characteristics, pass traffic between them, and track a multitude of network performance statistics. Through extensions to Tcl, namely OTcl and TclCL, it is also possible to link in C++ code, where detailed and intensive algorithms can be implemented. A visualization tool that normally accompanies ns-2, nam, allows the observation of the individual packets flowing through the network, link and queue statuses, and some simple annotations.

2.3. Roadmap

The following sections of this paper will trace the creation of an at first simple model of a fully meshed peer-to-peer gaming network that will be expanded upon to create a reasonably robust model which can be used to simulated the performance of a real network with varying network conditions.

The following section of this document will present a more detailed description of the challenges and requirements of fully meshed peer-to-peer game networks. Next, the steps taken to create an ns-2 model that simulates such a network will be documented. Subsequent sections will display results of various simulations to observe the performance of the model, an investigation into improvements to the model, followed by a discussion of the results, and finally conclusions and recommendations for future work.

3. Peer-to-Peer Gaming Networks

Peer-to-peer gaming networks are conceptually easy to understand – one instance of the application runs on each host, and each application must update all the other peer applications on a regular basis. The implementation of these networks is a challenging task when the performance requirements are stringent – which they are for networked games. Peer-to-peer network game requirements can be best divided amongst the three main factors commonly used to describe network performance: they require absolutely no loss, low delay, and the worse the actual delay is, the tighter the delay jitter requirement. To understand the roots of these requirements, and to specify them in more precise terms, we need to take a closer look at the typical architecture of a peer-to-peer game.

In peer-to-peer games no single host controls the authoritative view of the "world". A networked game can only function if each application has a consistent view of the relevant data – to be more precise, the game engines must have consistent state data, although what is actually rendered graphically on each user's screen may differ (e.g. graphical effects such as shading, and textures may be rendered differently due to differences in hardware). Without a central server, the applications have two basic methods available to them to remain synchronized: first, each peer could periodically send its entire state to each other peer – needless to say this is prohibitively expensive for any but the most simple of games; the second approach is the defacto solution, each peer must run the same simulation (which is just a complex finite state machine) – which requires identical inputs to produce identical outputs. Given this mechanism for keeping in synchronization, the following subsections consider its impact on loss, delay, and delay jitter, as well as the transport protocol selected to meet these requirements.

3.1. Loss in Peer-to-Peer Network Games

It clearly follows that if all clients need exactly the same input to their simulation, that no packets can be lost, otherwise the output of the simulation engines would quickly diverge due to differences in inputs, which is known as "desyncing", and the players would no longer be playing in the same game. Therefore, any dropped packets must be retransmitted until they are received.

3.2. Delay in Network Games

Realistic games are much like a TV show, the display must be refreshed at rates of roughly 30 frames per second or higher to appear smooth. The game engine generates these visual frames by rendering a visualization based on the state of the game – this state is updated by combining inputs from users with the previous state. As was noted previously, for these frames to match on all hosts, identical inputs must be applied to the game engine. Given current end-to-end network delays on the order of 50ms, let alone the time needed for state update and graphical processing, if the application were designed to wait until a frame was displayed before soliciting inputs for the next frame, it would be impossible to achieve a frame rate of even 20Hz. Cleary, a better approach than stop-and-wait is necessary to satisfy the expected video quality.

The widely deployed solution is to buffer user input until several frames in the future (for example, if the game allows six frames of delay, users would see the first frame rendered on the screen and enter their inputs, their inputs are sent to each remote peer while the game engine continues rendering the second, third, etc. frames, and only after a delay of six frames (i.e. the 2nd through 7th), are the user inputs applied to generate the 8th frame – if user inputs have not been received in time to generate the 8th frame, the peer freezes until it receives such data. Continuing the example, user inputs received after the second frame are applied to the 9th, inputs received after the 3rd are applied to the 10th, and so on), which frees the game simulation to proceed several frames while allowing time for all user inputs to reach all clients. The tighter the delay requirement, the more responsive the game feels to users as inputs are quickly applied to the simulation, the longer the delay requirement, the more "sluggish" the game feels. Any latency beyond the allowable value leads to the game freezing while it is starved of input; hence the delay requirements are very stringent for networked games. For typical real-time strategy games, which are not based entirely on reactions, user experience studies indicate delays on the order of 500ms are acceptable, while for reaction-based games such as first-person shooters and sports games, 100ms is a good time, and 200ms is considered adequate [3, 5].

Significant tradeoffs in terms of game responsiveness and requirements imposed on the network must be made by the game designers, based largely on the type of game, but also on their knowledge of the user base's typical connectivity, and perhaps input from the user community. Designers pick a fixed number of frames of delay between when input occurs and when it is used in the game simulation, which imposes a requirement on the network that it be able to deliver inputs from each peer to each other peer within that time frame. Any failure to meet this requirement results in an application being starved of input at which point it must stall (leading to the choppy "stop-and-go" behaviour typical of games playing with poor network connections).

3.3. The Effect of Jitter in Network Games

Networked game requirements on delay jitter are closely tied to the maximum allowable delay. In a network game where the maximum allowable delay is only slightly above the actual end-to-end delay, data inputs are arriving "just in time", and any jitter that interrupts this regular arrival will lead to frozen frames. Conversely, networks with much better end-to-end delay than required can tolerate a lot more jitter, so long as the sum of the average end-to-end delay, and the maximum additional delay due to jitter does not run over the maximum allowable delay, the game can continue without interruption. To summarize, the closer the average delay is to the maximum acceptable, the tighter the jitter requirements.

3.4. Transport protocol of choice: TCP

Up to this point, the theoretical requirements of a peer-to-peer networked game have been discussed; it is now necessary to consider the practical aspects of the implementation. To an application level programmer, there are two choices available for transport protocols: Transmission Control Protocol (TCP) and Unreliable Datagram Protocol (UDP). A student of a networking course (like myself), would examine the requirements discussed above, and conclude that what is needed is a reliable, lossless delivery which is what TCP offers us. In practice, however, many games actually choose UDP, and then have to implement a TCP-like application layer on top of UDP. The reasons for this are two-fold: some bandwidth savings can be made due to the smaller UDP header (offset to some degree by the increase in the payload of the segment to implement TCP-like sequence numbers, acks, etc.); and second: the actual operation of the protocol can be more finely tuned. Despite the predominant usage of UDP, it is more economical in the short time frame of this project to use TCP to quickly get the simulation operational, rather than spending a significant amount of time re-implementing TCP-like functionality at the application layer. Beyond the benefit of allowing more time for running simulations, this also affords the opportunity to investigate the feasibility of using TCP as an alternative to UDP for networked games. The following section discusses the creation of an ns-2 model to simulate such a network.

4. Simulating a meshed peer-to-peer game in ns-2

As this project involves the author's first experience with ns-2 (or Tcl scripting, or even interpreted languages for that matter), a non-trivial portion of the time was spent learning about ns-2 on the fly. This section therefore discusses not just the final implementation of the simulation, but the process by which it was developed.

4.1. The tools

All code in this project was developed using the latest stable version of ns-2 presently available: ns-2 version 2.26. It was downloaded and built on a RedHat 9 Linux box, along with the visualization package nam, as well as the Tcl, OTcl, and TclCL supporting packages.

The ns-2 documentation suggests that intense packet processing be coded in C++, while topology and non-numerically intense work be done in Tcl [6]. Since the network being simulated doesn't involve detailed routing algorithms, rather we're observing the flow of standard TCP data, the clear choice was to begin coding in Tcl, and in fact all code for this project was written in Tcl.

While ns-2 is geared mostly towards the analysis of network performance where the actual contents of the data don't matter, the analysis of application-layer performance makes it crucial to know what data has been received from each client – since out of order packets could all be transmitted together in one chunk to the application, there would be no way of knowing how many separate game packets were received without observing the received data. Fortunately, ns-2 provides a TcpApp wrapper class, which can be layered on top of a TCP agent to provide easy access to the data, which is exactly what is needed [6].

4.2. Step one – two peers sending periodic data

The first step in building up the solution was to create a new OTcl class called GameApp that represents the application layer protocol running on each peer. Beginning with two GameApp objects, one attached to each of two nodes connected by a direct link, a TCP connection (formed of two TCP Agents wrapped in a TcpApp instance) was created between them. The first iteration of the GameApp code merely defined a method to periodically send data to the other peers, using a concrete instance of the Timer Class provided by ns-2 to cause periodic timeout events.

4.3. Step two – handling TCP buffering issues

From this simple prototype one problem quickly became apparent: typically game packets are quite small, much less than the default TCP maximum segment size of 536 [4], and unfortunately ns-2's TCP agents buffered the data even though the slow start period should have been expanding the congestion window and thus allowing more data to flow. The number of frames of game data buffered caused substantial problems, as the default maximum segment size would cause the TCP agents to buffer many game frames making the delay before they were received (let alone sent) so great that the game would always be frozen. A simple workaround was to reduce the TCP max segment size to an appropriate value (double the packet size in a deterministic setting – in the case of variable sized packets of similar sizes a value slightly larger than most packets would work), forcing the TCP sender to send out packets quickly enough to satisfy the delay-sensitive demands while still allowing large enough packets in bursty periods to avoid queue overflows.

4.4. Step three – adding state

The next important step was to add the notion of tracking data received from the other peers. To implement the peer-to-peer gaming algorithm discussed previously it was necessary to enforce that no peer could send more than a few frames of data beyond the number received from each other peer. The maximum frames outstanding for the purposes of this project were set to 6 (i.e. if the difference in frames between one peer's received data, and the data it needs to send reached 7, the peer would freeze), and the frame rate to 25 frames per second. These values satisfy relatively high-performance delay requirements for a broad cross-section of games. The most interesting feature to observe was the "max delay" – that is for each peer the largest difference between the frame number to be sent, and the most recent received frame from each other peer. Adding labeling to each peer to display the maximum current gap, and colour coding them for further clarity created a useful tool for quickly visualizing the state of the network (see figure 2 in the following section for an example of the nam output).

4.5. Step four – expansion from two peers to N peers

The next logical extension was to move from a two-peer point-to-point configuration to a mesh-connected network with multiple peers. The GameApp class was modified to handle multiple TCP connections (one to each remote peer), and to keep track of the data received from each remote peer. While valuable for debugging the multiple connection code, additional topology setup code was required to model peers as having only one connection to an ISP, shared amongst all TCP connections. This was easily accomplished by adding an ISP node for each game application node. Each game application node could be connected to its ISP via a simulated ADSL, or cable modem, or old-fashioned modem connection. Connections between ISP's were then modeled by a fully meshed structure, where it was assumed that bandwidth over the Internet backbone is plentiful, and the main parameter of interest was delay. Figure 2 shows a sample nam visualization of this network in operation – inner nodes represent ISP's while nodes around the outer loop represent the game peers, with the max delay shown textually and graphically (green represents max delays of 4 or less frames, yellow represents 5 or 6, and red for more than 6 represents a frozen peer starved of data). Figure 3 shows this network later after the slow start TCP phase is over, and the peers are all transmitting data in a steady state.

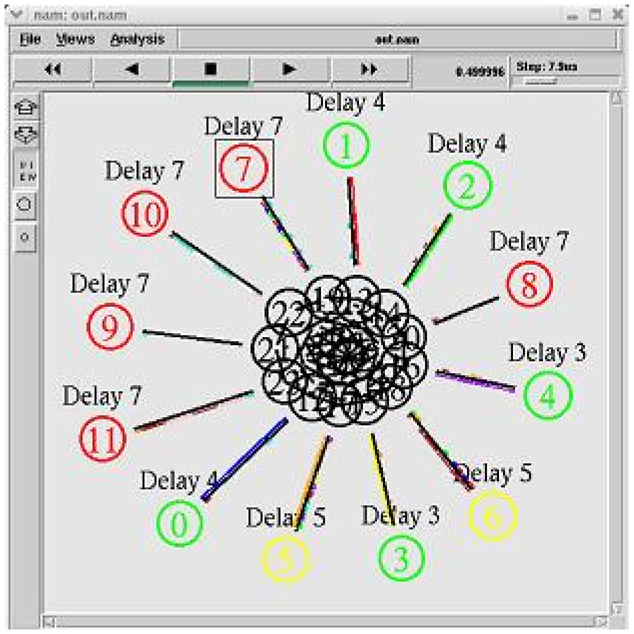

Figure 2: Sample nam output during TCP slow start phase (0.5 seconds after start of simulation)

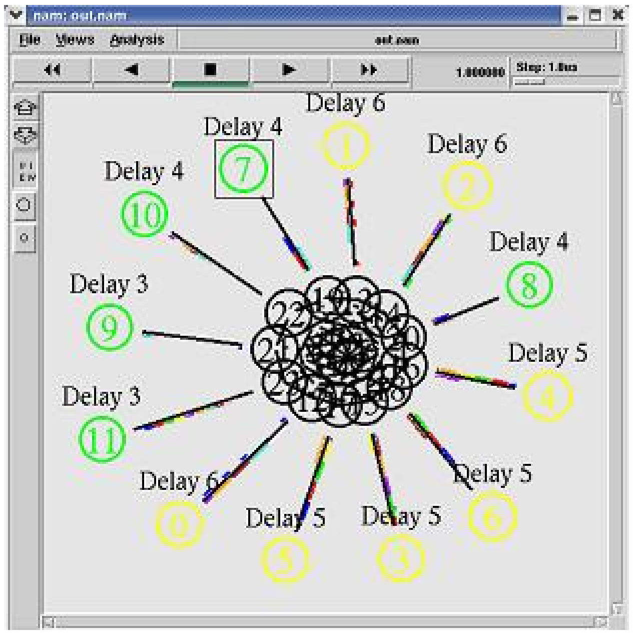

Figure 3: Sample nam output during steady state (1.0 seconds after start of simulation)

Clearly, the modeling of the Internet as a fully meshed set of ISP's is not entirely realistic. At first, this model was selected for ease of implementation, and while it is one of the main areas for improvement on this simulation, there are a few reasons why this model is not such a bad one: first, games are usually short, from 5 minutes to an hour, so daily traffic patterns do not play a strong role; second, the network path followed by packets often stays constant with fairly constant delays, as can be witnessed by running traceroute continuously over short periods to a variety of hosts. Further, from years of first-hand experience, network gameplay quality is normally relatively constant throughout a game – if all opponent's are observed to have low ping rates before a game begins (i.e. low end-to-end delays), that typically guarantees a smooth game, while high ping rates virtually always result in choppy gameplay throughout.

4.6. Initial test scenario

For initial testing, all parameters were fixed to the following values:

- Peer-to-ISP connections were modeled with typical ADSL connection values: 384kbps upstream, 1.5Mbps downstream and 10ms delay
- ISP-to-ISP connections were modeled with delays of 50ms (typical of connections across North America)
- The game frame rate was set to 25 frames per second
- The data payload size was fixed at 64 bytes – a fairly large value for typical games

4.7. Optimizing the algorithm

Before moving on to a thorough investigation of variable conditions, it was important to consider if the application layer communication was performing well enough. In fact, one serious issue occurred which was addressed. From the steady-state diagram (figure 3 above), it can be clearly seen that some peers have much higher delays than others, and in fact in this completely deterministic model the delays remained unchanged once the steady state was reached. Since each node has identical conditions, this raises the obvious question, why do some peers behave differently? The key lies in the naïve approach to sending data that was taken in the initial implementation – each time a peer had data to send, it sent it to the remote peers in the same order, i.e. node 0, node 1, node 2 (this is true of all peers, i.e. every single peer sends to the same node "0" first).

A more intelligent approach to sending data was clearly needed – and one was devised from the information already available. At each periodic sending of data, a peer checks the difference between the frame it's about to send and the latest frame received from each other peer – those peers which are the furthest behind are precisely those that it would be desirable to receive data from sooner. So, by informing each remote peer of the perceived delay, they can prioritize the order of packets to be sent. This approach was implemented (with a considerable reworking of the code), and did in fact yield improved performance. Both the overall delays were lower, and several more frames could be sent in the same time frame as various peers recovered more quickly from the starvation condition. In fact, with twelve peers there were approximately 3.7 fewer frames per peer at which each peer was starved of inputs. This new algorithm thus will be the basis of the remainder of this project, and along with the initial test configuration from the previous section will be referred to as the Reference Model against which other sets of data will be compared.

4.8. Initial Observations

While still an idealized simulation, with fixed packet sizes and network parameters, nevertheless the reference model captures enough of the idea of a peer-to-peer network to allow us to measure some reference performance numbers. The first observation we can make is how many peers can participate in the network before it reaches a point such that a steady state of continuous data transfer does not occur. In the above case with identical packet sizes, connection speeds, and delays, 20 peers could participate. Another useful characterization of the performance of the network is the amount of bandwidth consumed – an efficient algorithm would be able to operate even when the bandwidth required is close to the maximum available (i.e. it wouldn't require a lot of "spare" or wasted bandwidth). In fact 83% of the available (upstream) bandwidth was required to transmit each game packet once (including TCP/IP headers) in the 20-peer configuration, so the model can be considered fairly efficient. With the maximum TCP segment size set to twice the payload size (as previously discussed), a minimum average overhead of 20 bytes per game packet is required (i.e. the TCP/IP header combined size of 40 split over 2 game packets), thus the required bandwidth can be computed as the product of: 19 remote peers, 84 bytes / game packet (64 byte payload + average 20 byte overhead), 25 game packets / sec, 8 bits / bytes = 319.2kbps, which is 83% of our assumed ADSL upstream bandwidth of 384kbps.

Table 1 summarizes the results obtained in testing various numbers of peers in the Reference Model, and following is a brief explanation of the data columns. The frames frozen is the number of times a peer was ready to send data but had to abort due to input starvation (these frozen frames all occurred during the TCP slow startup phase). Steady-state time is the time (in seconds) after which the game stabilized, and data was transferred continuously – note that in a realistic game, test packets would typically be sent before the beginning of the game, so that the game could actually start after the completion of the TCP slow-start phase, thus avoid choppy gameplay in the first few seconds. Each peer tracks the max delay in frames between itself and all other peers, this value is averaged over the entire run for each peer – both the worst peer as well as the mean over all peers are displayed. Minimum bandwidth required is the minimum bandwidth required per peer to transmit data at steady state.

Table 1: Performance Characteristics of the Reference Model

Peers	Total Frames Frozen (avg per peer)	Average of All peer's Max Delays (frames)	Average of Worst peer's Max Delays (frames)	Bandwidth Required Per Peer (kbps)	Seconds to Reach Steady State (s)
2	0	1.79	1.79	16.8	0
4	0	2.10	2.13	50.4	0
6	0	2.37	2.72	84.0	0
8	0	2.60	2.89	117.6	0
10	0.10	3.06	3.37	151.2	0.32
12	0.42	3.08	3.51	184.8	0.32
14	0.79	3.60	4.31	218.4	0.52
16	1.6	4.49	5.05	252.0	0.76
18	6.3	4.83	5.20	285.6	1.52
20	10.6	4.89	5.42	319.2	1.60
Beyond 20 peers, the network did not reach a steady state.					

5. More Realistic Network Models

The network configuration tested in the Reference Model is clearly highly idealized, and useful for obtaining base performance measurements for comparisons against more realistic configurations. While more work could be done to model the Internet's performance to a greater degree of realism, the following sections describe the modification of various parameters, and their effects on the overall delay performance.

5.1. Adding deterministic delays

The first step in varying the completely symmetric and idealized reference model was to add some deterministic delay. The following subsections discuss the effects of adding delay to a peer-to-ISP connection, and then to an ISP-ISP connection.

5.1.1. One slow Peer-ISP link

Users will have varying latencies between their computer and their ISP, so the impact of a longer peer-to-ISP delay was the first effect studied. Tests were run with one peer-to-ISP's delay increased to 20ms (from the 10ms default in the reference model), and then 30ms, with all other parameters held constant. Figure 4 (below) summarizes the effect of this increased delay – essentially the average network delay was increased for smaller numbers of peers; presumably for larger numbers of peers the dominating factor was the overall congestion of the Peer-ISP links and the impact of the additional delay was negligible. It is interesting to note that the peer with the slow link itself was rarely the peer with the worst delay; while at first this may seem counterintuitive, it does make sense as the incoming and outgoing packets are both delayed by the same amount, so the slow link affects all nodes, not just the peer connected to it.

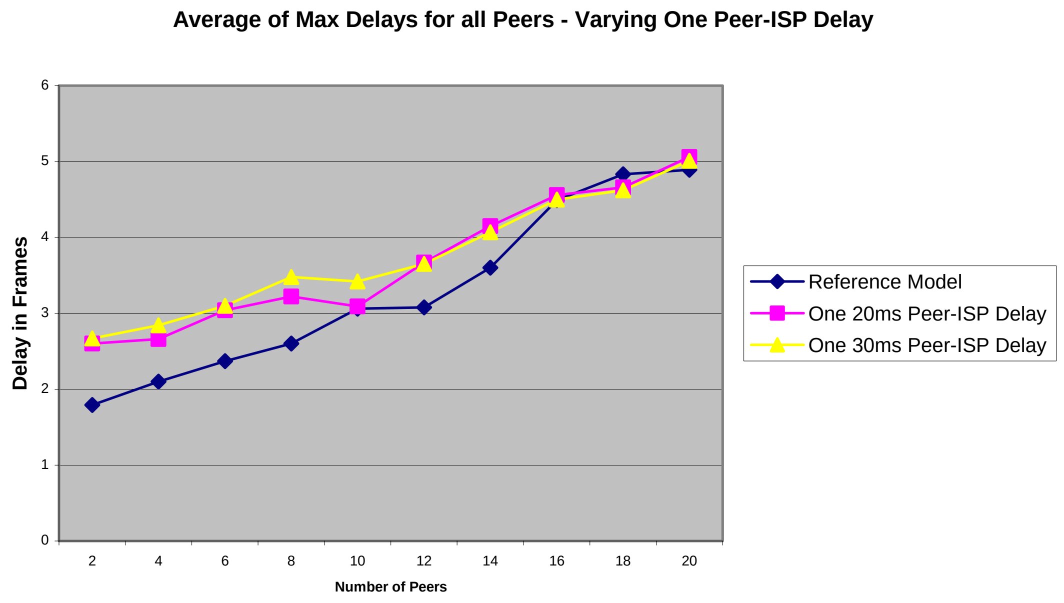

Figure 4: Average of Max Delays for all Peers – Varying One Peer-ISP Delay

5.1.2. One slow ISP-ISP link

Connections between ISP's will also generally be quite varied amongst each pair of the peers, due partly to geographical distances, but also due to the routing of IP packets. Similarly to the previous section, the delay on one link was increased, this time an ISP-to-ISP link was modified from the 50ms default in the reference model to 70ms, 85ms, and then 100ms. Figure 5 (below) summarizes the effect of this increased delay – again for low numbers of peers, the increased delay has a significant impact on the frame delays experienced at each peer, but for larger numbers of peers the overall congestion of the network is the dominant contributor to the delay, and the increased ISP-ISP link delay has a negligible effect. It is interesting to note that although in the slow start phase the two peers connected by the slow link were significantly affected, they did not perform significantly worse in the steady-state. In essence any effect of the increased delay on the two directly affected peers indirectly caused additional delays between them and all other peers, which in the end resulted in an overall increase in the average network delay.

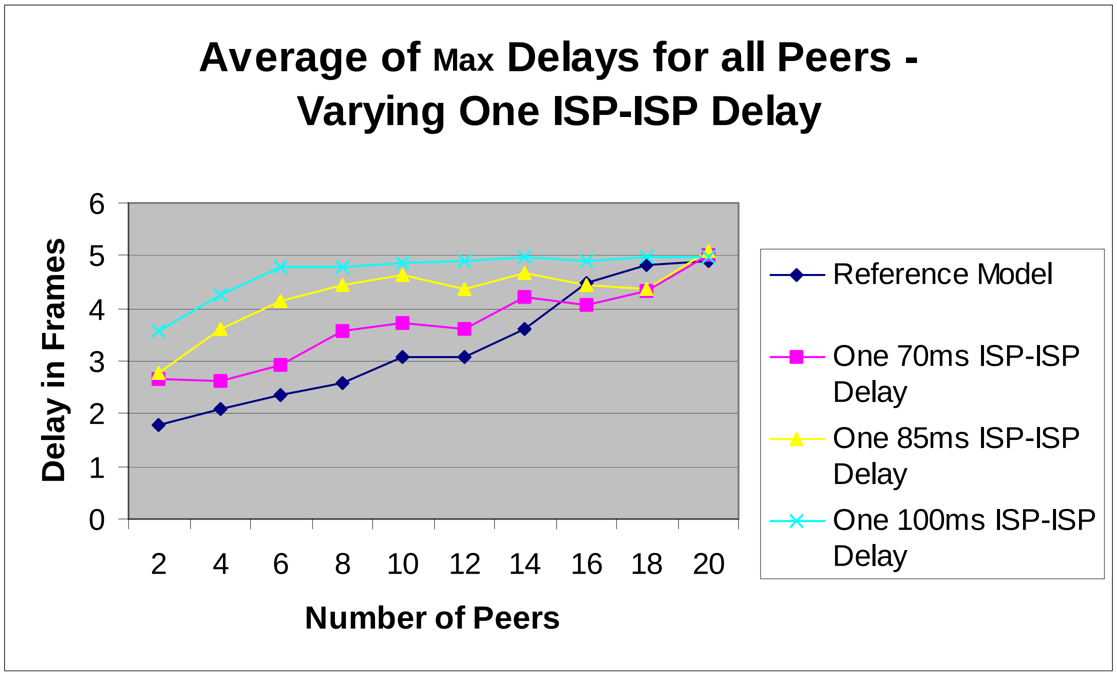

Figure 5: Average of Max Delays for all Peers – Varying One ISP-ISP Delay

5.2. Fully Randomized Delays

The previous sections investigated the impact of adding some delay to one link in the network, it is interesting instead to consider the impact of a completely randomized set of link delays. For comparison with the reference model, the mean of the peer-to-ISP delays was kept at 10ms, and the mean of the ISP-to-ISP delays was kept at 50ms, but they were chosen randomly at startup time to vary uniformly (between 5ms to 15ms for the peer-to-ISP, and 40ms to 60ms for the ISP-to-ISP). Three trials were run for each number of nodes, and the results were averaged, in order to avoid the impact of any peculiar sets of random link delays. Figure 6 (below) summarizes the impact of randomized delays on the performance of the network. As was seen in the previous sections, for low numbers of peers the randomness causes increased delay for the network as a whole, but for larger numbers of nodes the congestion of the network is the dominant factor.

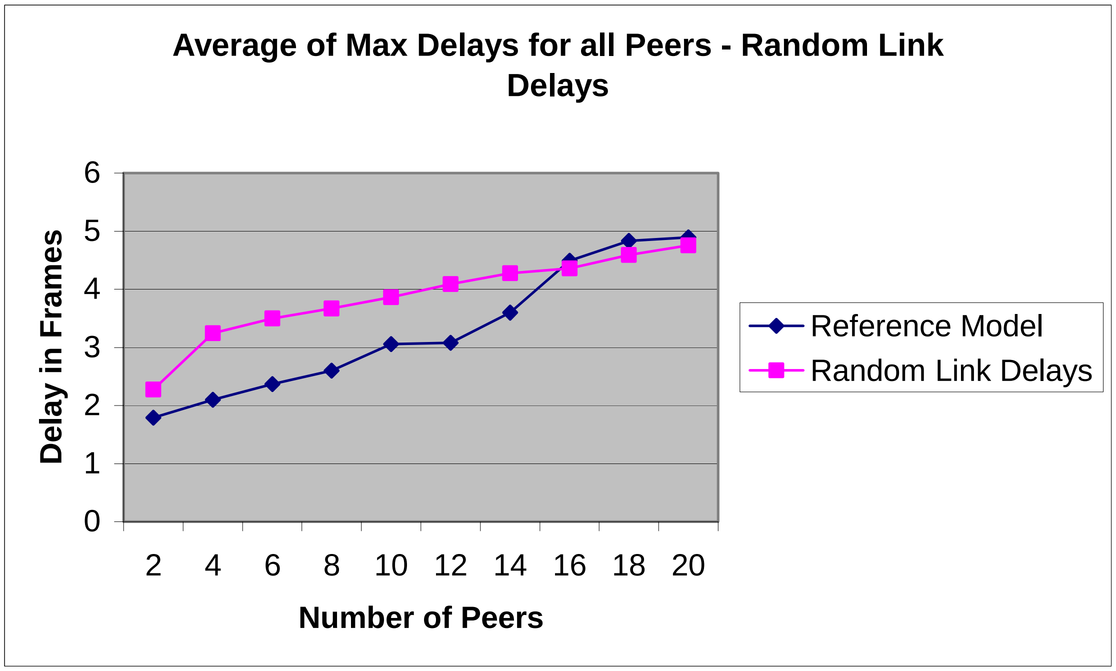

Figure 6: Average of Max Delays for all Peers – Random Link Delays

5.3. Randomized packet sizes

Some games may use fixed packet sizes, but it is more likely that there will be more or less data to send at various times, and this is the reason for testing with randomized packet sizes. Several sources quote the extreme distribution: F(x) = e^(-e^((x-a)/b)) as the best description of packet sizes [4, 5], so again to match with the reference model, an extreme distribution is selected that yields a mean of 64 bytes per packet, but clipping at the lower and upper ends to 24 and 128 bytes respectively (it is assumed that any game implementation would have a minimum amount of data to be transferred at each frame, and also a maximum amount that it would ever need to send, hence the extreme distribution is a good approximation except in the tails), therefore the value for a was set to 52, and b to 20 (a roughly affects the mean in the extreme distribution, while b mainly affects the standard deviation). Just like randomized link delays, three trials were run for each number of nodes, and the results were averaged, in order to avoid the impact of any peculiar set of random packet sizes. Figure 7 (below) summarizes the impact of randomized packet sizes on the performance of the network. As was seen in the previous sections, for low numbers of peers the randomness causes increased delay for the network as a whole, but for larger numbers of nodes the congestion of the network is the dominant factor.

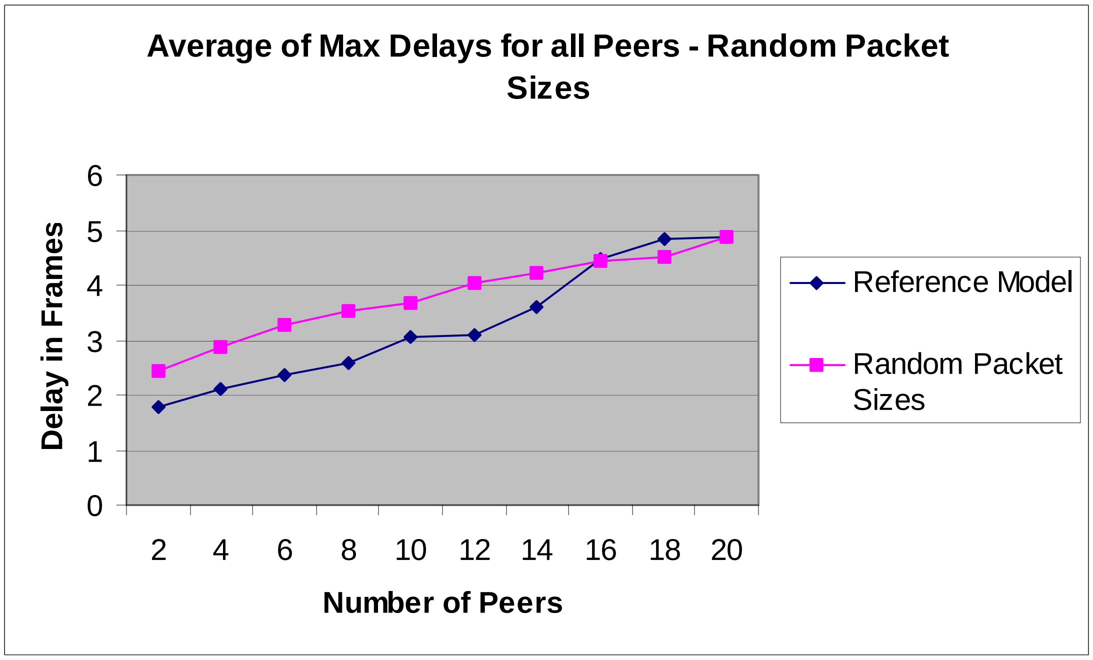

Figure 7: Average of Max Delays for all Peers – Random Packet Sizes

5.4. Putting it all together

As a final test, the effects of the previous two sections – randomized delays and randomized packets sizes – were investigated in conjunction. Once again three trial runs were performed for each number of nodes, with the results averaged. Figure 8 (below) summarizes the impact of both random link delays and random packet sizes on the performance of the network, compared with the impact of only random link delays, as well as only random packet sizes. The figure shows that there is very little difference between the impacts of just one parameter being random, or both, i.e. the effects are largely orthogonal.

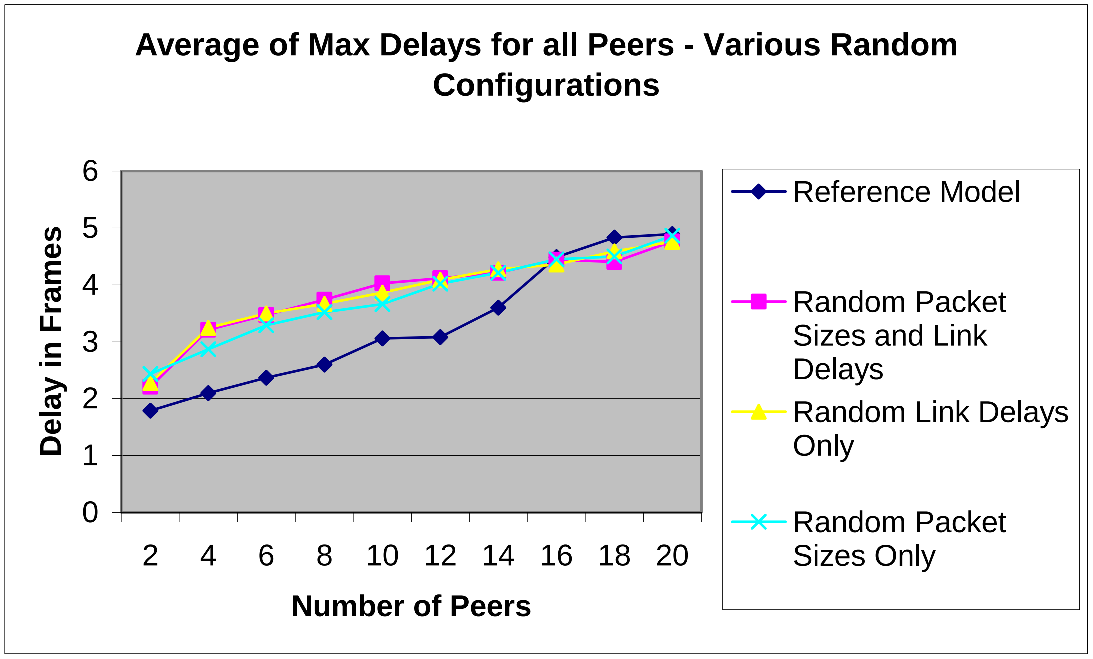

Figure 8: Average of Max Delays for all Peers – Random Link Delays and Packet Sizes

6. Future Work

As can be seen from the results obtained in this paper, a relatively simple TCP implementation for networked games appears promising, but significant work could be added in many areas. The following are four key areas recommended for future students to take this project from an interesting introduction to the performance of networked games to a full-fledged analysis of them.

1. **Realistic network models.** As alluded to several times in class, the modeling of the Internet is a difficult task, and unfortunately time did not permit a very realistic implementation of the network core to simulate real Internet behaviour. With more time and computational resources, as detailed a simulation of the Internet core as possible would lend much validity to the results. Also, it is possible that more than one user may be connected to a large ISP – while this paper essentially considers the worst case of users each connected to a separate ISP, it would be interesting to obtain statistics on the likelihood of multiple users being connected to one ISP, and the corresponding performance improvement.
2. **Comparison to existing application layer protocols.** This paper presented a fairly simple but robust TCP model for networked games, it would be very interesting to compare it's performance to several existing games. Source code for some games is available in the public domain, and could be added to ns-2, then compared against the TCP model presented in this paper (with parameters appropriately adjusted to make the comparison as fair as possible).
3. **Real traffic traces.** This paper analyzed the performance with both fixed-size data packets and the extreme distribution that is characteristic of many games, but for even more realism it would be interesting to use real traffic traces. These traces are of course easy to obtain, as any game of interest can be purchased and played on a home computer, with a traffic tracing utility to capture statistics of interest.
4. **Enhancing the intelligence of the application layer.** Most video games model some abstraction of a real-world physical arena, such as city streets, a hockey rink, etc. A key concept in these games is the location of players (or the units they control) – clearly players far apart are unlikely to interact with each other until they "approach". It would be interesting to investigate how a much larger game simulation in the hundreds (or thousands) of users could perform if only users in close proximity needed to update each other's state at a high frequency. If it turns out that the physical distribution of game players (and hence the distribution of frequencies of packets sent to the corresponding hosts) is similar for a wide spectrum of games, this would make a very interesting area of research.

7. Conclusion

This project successfully implemented an application-layer protocol using TCP to model a peer-to-peer networked game. While much work can be done to improve the realism of the Internet core, simulated here as a fully-meshed collection of ISP's, the initial results show that it is quite robust, especially given the simplicity of the algorithm. The protocol performed well in a completely deterministic case, and as expected the performance was affected to a certain degree by randomizing delays and packet sizes. The effects were most pronounced for low numbers of peers, while as expected the dominant factor affecting average delays with a large number of peers was the high congestion of the peer-to-ISP links.

8. References

1. Postel, J. “TRANSMISSION CONTROL PROTOCOL”, IETF-RFC-793, http://www.ietf.org/rfc/rfc0793.txt?number=793, September 1981
2. Bonham, S., Grossman, D., Portnoy, W., Tam, K. “Quake: An Example Multi-User Network Application – Problems and Solutions in Distributed Interactive Simulations”, University of Washington, http://www.cs.washington.edu/homes/grossman/projects/561projects/quake/Quake.ps, May 2000
3. Bettner, B., Terrano, M. “1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond”, http://www.ensemblestudios.com/openjournal4/story/networking.ppt, March 2001
4. Borella, M. “Source Models of Network Game Traffic”, 3Com Corp., http://www.borella.net/mike/academics/game-traffic.pdf, November 2003
5. Faerber, J. “Network Game Traffic Modelling”, Institute of Communication Networks and Computer Engineering, http://www.ibr.cs.tu-bs.de/events/netgames2002/presentations/faerber.pdf, April 2002
6. Fall, K, Varadhan, K. “The *ns* Manual (formerly *ns* Notes and Documentation)”, UC Berkeley, LBL, USC/ISI, and Xerox PARC, http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf, December 2003