Project Number: MLC-NG01

Analyzing and Simulating Network Game Traffic

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

Dave LaPointe

_____

Josh Winslow

Approved:

_____

Professor Mark Claypool

Date: December 19, 2001

ABSTRACT

Network games are becoming increasingly popular, but have received little attention from the academic research community. The network traffic patterns of the multiplayer games Counter-strike and Starcraft were examined and documented. Analysis focused on bandwidth usage and packet size. The games were found to have small packets and typically low bandwidth usage, but traffic patterns varied widely between games. Modules for these two games reflecting these findings were added to the network simulator NS.

Table of Contents

1 – Introduction

Since the release of the first multiplayer computer game, the level of attention game developers have devoted to the network aspect of a game has dramatically increased as games continue to grow in quality, depth, and complexity. However, the size and behavior of the networks on which such games have been designed to run are also changing rapidly, and games are becoming increasingly demanding on network hardware to run quickly over the Internet.

However, the Internet was not designed with time-intensive applications taken into account. The Transmission Control Protocol (TCP) is the most widely used protocol on the Internet. Almost all traffic that travels over the Internet, with the exception of streaming media and real-time games, is carried by TCP, but it was built on the premise that packets had to be delivered reliably and in sequential order. Streaming media applications and other real-time activities such as multiplayer games do not require fully reliable delivery. The delay experienced in re-transmitting packets and the overhead of packet acknowledgements can cause an enormous amount of slowdown in a game, as the underlying protocol must wait for these retransmissions and consume extra bandwidth for the acknowledgements.

Games often use the User Datagram Protocol (UDP) to avoid these problems, as UDP packets are not guaranteed to reach their destinations, and thus do not require a mechanism for retransmissions and acknowledging successful deliveries. However, any of the routers through which these packets are passed can be overburdened by traffic and lose packets. A game must therefore provide its own means of tolerating packet loss, as a particularly overburdened router could lose a large amount of data before routing schemes can divert the flow of packets to compensate.

These considerations are relatively new to the world of networking, as most Internet applications transfer text or files over TCP, and only newer technologies dealing with streaming media parallel some of the issues introduced by games. Therefore, the amount of research devoted to network behavior such as these programs generate is deficient, and progress is somewhat restricted compared with research on traditional TCP traffic.

The lack of research into networked multiplayer computer games has left a large informational gap concerning the network traffic flows that these games create. Because of this, routers may be able to provide different queue management services than are used for traditional traffic. Better router queue management would improve the ping[1] times for gamers leading to an increase in playability for many games, especially on high loss or low bandwidth connections.

1.1 – Status of the Gaming World

The lack of research into network gaming was never a problem until fairly recently. Before Doom,[2] released in 1993, nearly all networked games were text based and used telnet or similar protocols to transmit data from player to server and back. But even with the advent of Doom, networked gaming was still confined to a small portion of the population. However, in the last 5 years, with the growth of the Internet, this has changed drastically.

In the current environment, the vast majority of networked gamers play card games, chess, checkers, and similar games.[3] However, the traffic generated by each of these games is quite small and infrequent.[4] The genres of games that have the most players, after parlor games, are First Person Shooters (FPS) and Massively Multiplayer Online Role Playing Games (MMORPGs), followed closely by Real Time Strategy (RTS) games.

Since Doom, FPS have made up a large portion of networked gaming. In these games, the player views the world through the perspective of this character (the first person part) and is usually required to move around various locations slaying monsters and other players, with an amalgamation of ranged weaponry found along the way (the shooter part). On an average night, there are well over 10,000 servers for games using

---

[1] Ping is a simple program that measures the amount of time a small (usually 64 byte) packet takes to travel to a remote machine and back. High ping times correspond to high latencies between machines.

[2] http://www.idsoftware.com/killer/doomult.html

[3] Archives of lumthemad.net, currently offline Potential mirror at http://www.brokentoys.org/

[4] Archives of lumthemad.net, currently offline. Potential mirror at http://www.brokentoys.org/.

the Half-Life engine supporting over 40,000 gamers.[5] Other FPSs support slightly smaller user populations[6].

MMORPGs have been a rapidly growing field since Ultima Online's[7] release in 1996. A MMORPG can be safely thought of as a graphical Multi-User Dungeon.[8] All MMORPGs released thus far provide some mechanism for character advancement, large areas of landmass to travel across, and other players to interact with. The "big three," Asheron's Call,[9] Ultima Online,[10] and Everquest,[11] claim to have nearly 1 million subscribers combined, and while only a fifth of them login on any given day,[12] these players consume a non-negligible amount of bandwidth. In addition, several more MMORPGs have been released in recent months,[13] adding to this total.

The first RTS game was Dune 2,[14] which was based loosely on the world from the Frank Herbert series of novels. RTS games are generally characterized by resource collection, unit construction, and battles that consist of large numbers of animated soldiers standing a few feet apart going through the same animated attack motion over and over. All of these actions happen continuously, unlike earlier strategy games (most notably Civilization[15] and various war games from SSI and others) in which the player could take as much time as he or she needed to plan his or her turn before pressing the

---

[5] Average of 3 randomly selected nights between 10/2/01 and 12/15/01

[6] Tribes 2, average 3 randomly selected nights between 10/16/01 and 12/15/01

[7] http://www.uo.com

[8] First pioneered in the 70s. For more information, see http://www.legendmud.org/raph/gaming/book.html

[9] Asheron's Call (http://www.zone.com/asheronscall/) has an average of 12k players a night. Observations and developer comments seem to indicate that about 1/5 of all subscribers play on any given night. However, this doesn't seem to match previously released subscriber figures. Asheron's Call most likely has between 75,000 and 120,000 subscribers.

[10] At one point, UO had advertised having over 300,000 subscribers, but this has almost certainly fallen due to Asheron's Call, Dark Age of Camelot, Anarchy Online, World War Two Online, and Jumpgate all having been released since then.

[11] Everquest (http://everquest.station.sony.com/)has over 410,000 subscribers.

[12] Comments made by several developers on the now defunct www.lumthemad.net.

[13] Dark Age of Camelot, Anarchy Online, World War Two Online, and Jumpgate

[14] http://www.dune2k.com/duniverse/dune2/

[15] See http://www.civ3.com for the second sequel of Civilization.

process turn button. Since Dune 2, there have been several more games released,[16] each with their own variation on the theme. Currently, the number of RTS fans playing Starcraft[17] on an average night numbers at least 20,000 players.[18]

1.2 – Trends

The rapid growth of the Internet and the fall in computer hardware prices started the growth of multiplayer gaming. The cost for a computer capable of playing the latest popular games has fallen from almost $2,500 in 1997[19] to around $1,600 in 2001.[20] Also, the recent Internet boom has placed a computer in 54% of American households with 80% of those having Internet access.[21] These two points taken together show a dramatic increase in potential game players.

Another factor in the increase in network gaming is the change of developer focus from single player games to multiplayer games. This is most obvious in the development of FPS. Up until Quake 3,[22] all FPS came with an expansive single player game with multiplay usually added on as an afterthought.[23] However, the player communities would modify these games, adding new multiplayer content and game styles.[24] Realizing this, the developers of Quake 3, id software[25] released Quake 3 with minimal single player

---

[16] Command and Conquer, Total Annihilation, Age of Empires, Black and White, Myth, and Shogun: Total War

[17] http://www.blizzard.com/worlds-starcraft.shtml

[18] Still gathering data

[19] Custom built Pentium/200, 64mb RAM, 4mb Video Card, AWE64, 4 GB hard disk, 24x CDROM

[20] Dell Dimension 8200 Series as of 10/9/01

[21] http://www.internetnews.com/isp-news/article/0,,8_879441,00.html

[22] http://www.quake3arena.com/index.html

[23] See http://www.gamasutra.com/features/19990903/lincroft_06.htm

[24] These modifications, called mods, have been around since the release of Quake. Many of these mods radically change the game play from that of a straight deathmatch or team based capture the flag to soldier Sims, counter-terrorist games, and games styled off of action movies. The mod community for Half-Life has put out over 15 mods that have been registered with the main WON servers. Though few are as wildly popular as Counter-Strike, many have devoted followings.

[25] www.idsoftware.com

content and a determined focus on the network code and multiplayer level design. Since then, many games, regardless of genre, have been released with some form of multiplayer gaming.[26] Many have emulated Quake 3 in dropping single player game play entirely (Tribes 2,[27] Unreal Tournament,[28] and Majestic[29]).

Also, games industry has also begun shifting from hardcore gaming (FPS, MMORPG, RTS) to more mass-market games. The best selling PC game in 2000 was The Sims, a real life simulator that generally doesn't appeal to the traditionally gaming community. Of the top 10 PC games, only 2 (Age of Empires II and Diablo 2) were games enjoyed by "traditional" gamers.[30] While none of these games have been multiplayer, the first mass-market game that is multiplayer will decidedly impact network traffic.

However, computers are not the only source of multiplayer gaming. Consoles systems from Nintendo, Sony, and Sega have traditionally been the bastion of multiplayer games. It is no surprise then that all of the next generation console systems from the Sega Dreamcast[31] forward have included some way of connecting to the Internet to play games against others. Examining sales figures, it becomes apparent that consoles are a major part of the games industry. An average computer game sells between 20,000 and 50,000 copies,[32] but console games easily outsell PC games. The top 10 Console games sell as many copies as the top 200 PC games.[33] Phantasy Star Online,[34] the first console game with a strong network based multiplayer component, and its sequels are among the best selling console games in the last year. Clearly with the release of more and more PC

---

[26] See Arcanum: Of Steamworks and Magicka Obscura from Sierra, Vampire The Masquerade - Redemption from Activision, and Pool of Radiance: Ruins of Myth Drannor from UbiSoft for examples.

[27] http://tribes2.sierra.com/

[28] http://www.unrealtournament.com/

[29] http://www.ea.com/worlds/games/pw_majstc00/hatted_jump_page.jsp

[30] http://www.idsa.com/releases/SOTI2001.pdf

[31] Released in Japan in November of 1998.

[32] http://www.bighugegames.com/jobs/industryjobtips.html

[33] http://www.sega.com/games/post_gamearticle.jhtml?article=art_consolevspc

[34] http://www.sega.com/games/dreamcast/post_dreamcastgame.jhtml?PRODID=187

multiplayer games on to consoles (Quake 3, Solder of Fortune,[35] Unreal Tournament, Half-Life, etc) and the release of newer networked games the numbers of players playing these games will drastically increase.

Finally, multiplayer games have been rapidly spreading to countries outside of North America. Europe has a large number of computer users has a large number of network gamers.[36] The Asia rim nations are also very involved in network gaming, with nearly 10,000,000 Koreans playing one game, Lineage, alone.[37] Since few game servers are physically located in Asia or Europe, a large volume of traffic must cross the transoceanic connections.

1.3 – State of Research

Despite this massive growth and large user base, issues related to the effects of these games on network congestion have been largely neglected in both academic and industry publications. Industry articles are more concerned with the management aspects of game development rather than the technical issues confronted by the programmers. Setting milestones correctly, making the scheduled ship date, and setting realistic technological goals are all very important, but issues like developing a robust network layer or minimizing network load suffer due to the lack of economic pressure.

Conversely, most academic game studies have focused more on usability and game play issues brought about by new network protocols rather than analyzing the performance of real game protocols. This leads to a significant knowledge gap as to what kind of traffic patterns, bandwidth usage, and protocols games use.

Several issues contribute to the lack of research into games. Gaming has not been a traditional research field in academia. In general, games are looked at as more of a fun diversion rather than a business related application, despite games earning nearly as much revenue as the movie industry.[38] The majority of grants go to research into hardware,

---

[35] http://www.ravensoft.com/soldier.html

[36] http://extra.gamespot.co.uk/pc.gamespot/features/aroundtheworld_pc/

[37] Though it should be noted that Lineage has a much different subscription model: http://www.happypuppy.com/features/bth/bth%2Dvol10%2D9.html

[38] http://www.kanga.nu/archives/MUD-Dev-L/2001Q2/msg00211.php

routing, and web related research.[39] However, it is important to look at other types of traffic and games will soon be making up a sizable percentage of network traffic.

The games industry as a whole, where one would expect most of the games related research to be performed, has also shown a notable lack of interest in doing network related research. Most game developers chose their professions because they enjoy the process of making games and the end product, rather than the often less applied work of academia. In addition, most of the published papers concerning games are postmortems[40] concerning the development process, written by an often non-technical producer. Finally, since the games industry is on a very tight development cycle, most research time is spent on graphics, an area that changes radically every 6-8 months as each new generation of video hardware is released.

Because of this lack of knowledge, reasons for good and bad performance are also not well researched. Various implementations might perform well under most conditions, but might perform very badly under others due to one easily changed design decision. Fixing this sort of issue requires the knowledge that the problem exists, knowing how different types of traffic are queued at the router, and having the resources to implement a quality solution.

It is our goal to analyze the network traffic of two of the most popular games and develop a module based on them for the network simulator NS, a popular simulator used in academic research. We chose two of the most popular genres of games, First Person Shooters and Real Time Strategies, with the intent of representing a larger range of data than a single genre would likely provide. During our analysis of these games, we measured bandwidth and packet size by player number due to the effect the number of players has on the network footprint of games. We also wrote an extensible module to simulate these games that will provide an easy way to add further games into NS. This module will allow researchers to gain a better idea of what a few games' network traffic impact is by simulating them. It will be possible to construct better router queue

---

[39] http://www.interact.nsf.gov/cise/abst.nsf/anirabst2000?OpenView

[40] An after the fact analysis of what the main problems were in the development cycle, how they were solved, and what new approaches saved time.

management techniques that take into account the lack of flow control in most games, as well as the size and number of packets that games produce.

1.4 – Synopsis of this Document

This report is divided into seven chapters that are organized as follows. Chapter 1 is the introduction; it contains a brief overview of the project and the motivation behind performing it. Chapter 2 is the background; it contains a list of related works and commentary about them. Chapter 3 is the methodology; it contains the process utilized when performing our analysis of the data. Chapter 4 is game traffic analysis; it contains an overview of the data collected and its properties. Chapter 5 is game traffic simulation; it contains a description of our work on NS and validation of our simulation. Chapter 6 contains our conclusions. Chapter 7 details the further work we would like to see done in this area.

2 – Background

Discussing the most current developments in the world of multiplayer games is useful in understanding the need for extensive research in any particular game's behavior when running over the Internet. There are, however, few publications that relate to the network aspect of gaming, as most pertain to more general design issues. A sizeable portion of the articles that discuss network issues were written by game developers faced with the specific challenge of adapting a game to multiplayer functionality. These developers faced issues such as minimizing user-perceived latency, bandwidth limitations, coordinating the game environment between users, and compensating for Internet transmission latencies.

Network Simulator (NS) is a program that accurately simulates network traffic using an event-driven, object oriented technique. Useful mainly for studying the effects of variable packet loads on routers, the simulator is a powerful tool for analyzing network performance and illustrating traffic patterns. However, it goes not currently support traffic generated by multiplayer games. This functionality is to be implemented in our project.

This chapter discusses some of the conclusions reached by game developers working on network aspects of multiplayer games and relates them to the goal of this project. In addition, the functionality of NS is explained in detail, so comparisons can be made between existing simulations and the sort that this project seeks to produce.

2.1 – Examples of Network Game Development: Articles by Game Developers

The lack of research into the facets of network gaming is reflected in the low quantity of academic articles published about the subject. In addition, a number of articles illustrate this lack of research with a consistent shortage of references. Very few articles actually make note of previous work, mostly because this work was never accomplished, and many share the same structure: a general warning to colleagues that there are a number of lessons to be learned in designing games for Internet playability.

Typically, an article must relate the issues faced as unique to their designs, as there are no standards in network-game interfacing as of yet. In fact, many articles barely describe the most critical issues behind efficient networking plan. An example of this kind of article is a post-mortem article by Peter Lincroft on issues he faced while working on X-Wing vs. Tie Fighter.

2.1.1 – "The Internet Sucks: Or, What I Learned Coding X-Wing vs. Tie Fighter" [Lin 99]

This article, originally published on Gamasutra,[41] describes the problems involved in designing a networking model for an existing game engine. Originally, this engine had been designed for the original *X-Wing* game, which was strictly single-player and therefore not designed with the Internet as a consideration.

The designers had a number of factors to consider in re-writing their engine to effectively implement a scheme for running multiplayer games. First, they knew that the level of complexity the original engine was capable of attaining was going to have to be supported by the new model. This requirement set the amount of information that had to be passed between players to a large amount, which would be difficult to achieve over slow connections. Second, they knew that they would not be able to provide dedicated servers, because of the high expected cost of maintaining them. They could not allow gamers to set up their own servers either, due to the licensing issues. This meant they had to use a peer-to-peer networking model.

The problem that arose with this, however, was that the amount of bandwidth required per user would be proportional to the number of players in a game. The Internet has no generally available multicasting capability available to game developers, so sending the same amount of data to each user required a separate transmission to every other player. This would not necessarily have been very problematic, but the designers' primary goal was to provide adequate performance for users with the bandwidth of a 28.8 bps modem.

---

[41] Gamasutra is a website devoted to game development. A solid majority of the information available to game developers at the time of this writing can be found here. The site is located at http://www.gamasutra.com.

It was decided that the amount of bandwidth needed could be greatly reduced with the proper information-coordinating algorithm. They opted to have each game client send only information about its player's actions. These could then be assembled to determine the state of their environment. In addition, the game would be structured so that one player acted as a game "host," assembling the information collected about each player's actions and distributing it to the other players. This increased the amount of bandwidth needed by the host, but greatly reduced the amount needed for other users, as they needed only now to send out one copy of their information. The game then proceeded in a lock-step fashion, in which player's commands (turning, shooting, etc…) would be sent to the game host, complied with those of other players, and distributed to the other players. Then each machine, with identical sets of information, could process the data on its own and the environment would appear exactly the same for every player.

It is clear, however, that this is where traditional networking concepts for the Internet are generally inapplicable to synchronizing a real-time environment between a number of distant users without a great deal of latency between cycles. The majority of applications designed for networking require the reliable and orderly delivery of packets. This means that if a packet is lost in transit, it is retransmitted, and its receiver must simply wait for the packet to arrive. This model functions very poorly for real-time applications, because time is as important as reliability. A good example of this is a simple FPS with two players, in which the game environment (player position, direction, etc), is updated every few milleseconds. If, in a transmission of environment data, a packet is lost, it may not affect the quality of the game very much. More likely, the players would barely notice the change, as the missing data would likely only result in a slight adjustment of each user's perception. However, if the packet must be resent before any more could be processed, the game would halt until that packet arrived. This is obviously undesirable, and extends to many real-time applications.

The X-Wing vs. Tie Fighter team realized this problem with using TCP after trying to run their game over the Internet. Almost all traffic on the Internet was in the form of TCP packets, which provided reliable transmissions. On LANs, the game ran very well, because there was little to no need for re-transmitting packets. The Internet, however, is considerably more lossy than a LAN, and packets were regularly lost in

transit. This led to re-transmissions, which stopped the game while packets were re-sent, and tended to frequently produced latencies that ranged from 5 to 10 seconds. This was simply unacceptable. The natural solution was to use UDP in place of TCP. UDP is connectionless and does not guarantee reliable transmissions, so lost packets are not retransmitted.

This made the game run a great deal faster, but introduced a new problem. Their model required every packet delivered to function properly. In addition, the game could still only run as fast as the slowest user, i.e. a cycle could not be completed until the necessary information was received from every player, so the cycle would take as long as the longest transmission time. Attempts at correcting this by putting a limit on the amount of time a host would wait for this information before ignoring it only made the gameplay sporadic for the affected player.

A few conclusions can be drawn from this example. First and foremost, real-time multiplayer games should not be run over TCP. It is therefore reasonable to hypothesize that any games that attempt to use TCP will experience a great deal of latency when packets are lost. Second, to effectively implement the algorithms necessary to run a fast, well-coordinated game, the developer must account for the timing issues introduced by the Internet before the design process even begins. By today's standards, the process by which this game was adapted for multiplayer capability is exhaustive, that is, they had to begin with the most basic concepts in networking. An effective traffic measuring program would significantly increase the developer's ability to see problems in a game's multiplayer implementation long before the testing phase of the development cycle. This is the intention of our NS module.

2.1.2 – "1500 Archers on a 28.8: Network Programming in *Age of Empires* and Beyond" [BT 01]

The existing engine for this game was a single-threaded cycle of taking input and processing the game environment accordingly. Just as the *X-Wing vs. Tie Fighter* developers, this design team used an algorithm that involved passing only user inputs between machines and using them to run the same simulation simultaneously on all machines. The difference here, however, is that the tolerable latency for a Real-Time

Strategy (RTS) game is much higher than for other real-time environments. This is true primarily because the amount of input from the user is a great deal less precise and tends to be less frequent. A person playing a flight simulator will usually be changing direction and speed almost continuously, but a person playing an RTS will generally issue commands to units in the game a maximum of only a few times per second. In addition, the player in a flight simulator requires instant results. If there is a perceptible delay between moving a joystick and seeing the craft turn, the game becomes unplayable. But a RTS player will generally not notice a unit in the game take a few milliseconds to start walking to where the player clicked. An engine can therefore take more time to process a RTS player's game cycle, as the player cannot detect such small latencies.

The Age of Empires (AoE) team decided that, given the time required to process commands, they would implement an algorithm that allowed the game to process a cycle while receiving commands for the next cycle. They achieved this by scheduling commands to be executed two cycles in the future, and since the user could not detect the latency, the system would run smoothly under normal conditions.

However, when communications latency is introduced, the system is slowed. The engine had to receive all inputs to process a turn, so transmission reliability was required. They did not make the mistake of using TCP, but when a game cannot proceed without all players' actions accounted for, the game halts until the information is received. The development team addressed this issue by maintaining a target frame rate for all users and adjusting this rate based on average ping times, previous latencies, and machine speeds. A game would slow down when network traffic became heavy between users and gradually speed up as traffic returned to normal. Unfortunately, with this algorithm, the game only runs as fast as the slowest machine or network connection, and everyone experiences the effects.

The team reported that 250ms was barely noticeable, that 500ms was "very playable," and that over 500ms tended to be sluggish in terms of user-perceived latency in their game. It follows intuitively that this would be the case for most RTSs, so this algorithm is, though not very efficient, more than suitable for its purposes. Games that require more information to be passed between players, however, would likely begin to experience a greater amount of slowdown as the bandwidth requirements increase.

Ensemble Studios' plan for their next RTS game is heavily network-oriented. They are putting a great deal of consideration into writing their own network libraries to avoid the effect of third party software slowing the game down. They have made extensive plans involving information coordination algorithms and network game debugging utilities, and are beginning to concentrate more on what most game developers will be finding important: good networking. This is a sign that research in this area will develop quickly with future releases.

Most important, however, is their decision to implement extensive metering throughout the development of the project. Essentially, this means they were always aware of the latencies caused by network bottlenecks and slow users. NS is an excellent program for simulating the effects of network traffic, and would likely benefit these and other developers.

2.2 – Academic Research

As mentioned earlier, the level of academic research into network gaming has been on the rise for several reasons, including growing interests in new protocols as well as concepts that can be applied in other types of programs. Some of the motivations and findings of this research are illustrated by these examples.

2.2.1 – "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization" [Ber 01]

In this article, the author goes over the basics of multiplayer game networking on the level of how an FPS controls data sent by various users and deterministic methods of compensating for latencies. The article first presents a model for the typical sequence of events on the client side of the client/server architecture used in both games:

Sample clock to find start time
Sample user input (mouse, keyboard, joystick)
Package up and send movement command using simulation time
Read any packets from the server from the network system
Use packets to determine visible objects and their state
Render Scene
Sample clock to find end time
End time minus start time is the simulation time for the next frame

Though this model primarily seems to address an algorithm for determining frame rates, the focus of the article is placed on keeping large latency differences between individual clients and the server from having a detrimental affect on gameplay. The article does not explicitly mention this in the title, but its data is based on the *Half-Life* and *Quake* engines.

The problem faced is simple to state, yet difficult to solve. Take as an example a simple FPS running with two players in the game. One player has a much greater ping time to the server than the other. This "slower" player fires a shot directly at the low-ping player, and from his perspective, hits the other player. The other player, however, has moved out the way of the shot since it was fired. The high-ping player, however, has not yet received data from the low-ping player indicating the dodged shot, so it appears to the high-ping player that the shot hit. Coordinating this action with the low-ping player makes that player see an evaded shot, but feel the effects of being hit. If, however, the action was coordinated in such a way that the low-ping player's real position was taken into consideration at the server before the shot was reported, the high-ping player would believe the shot had hit, when in fact the other player had moved out of the way. In either case, one player experiences a result that is contrary to what he perceived.

The issue raised here is that there are a number of steps that must be taken in order to keep the user's perceived latency to a minimum, and this cannot always be accomplished by streamlining the network side of the game. Developers that must rely on client-side prediction and lag compensation to keep a gamer from becoming frustrated by latency are at a serious disadvantage in keeping a decent level of accuracy in the game environment. Speeding up the communications between clients is therefore essential to keeping players interested in a multiplayer game.

2.2.2 – "Designing Fast-Action Games for the Internet" [Ng 97]

When considering the requirements for network usability in multiplayer games, it is important to characterize the network over which these games will be played. In the case of the Internet, there are a few expectations as far as bandwidth and latency are concerned. This articles attempts to explain these expectations, and finds client-side lag compensation methods to be critical.

The first issue raised in network performance is a set of reasonable expectations regarding bandwidth. The number of nodes in a star topology network directly affects the amount of data that must be sent and received by each node. It is therefore necessary to consider the amount of data needed to coordinate the game environment between players and adjust the bandwidth limitations accordingly. However, this is difficult when the Internet tends to exhibit sporadic periods of congestion. This is one area where the ability to map traffic patterns on a variable network environment is exceptionally useful.

The other network performance related issue deals with latency and types of user connections. It appears that modems are the worst types of connections for multiplayer gaming, as modem compression schemes and bandwidth limits tend to keep latencies consistently high. It appears that most games should be designed with broadband as a basis for latencies these days, but considering modem users is also necessary to reach the entire gaming audience. Simulating traffic over modem connections is therefore an important step in studying in-game performance.

Another significant contributor to latency is the packet router. Basically, most routers are not set up to handle multiplayer game traffic very efficiently. They tend to use a store-and-forward scheme in which they accumulate packets in their buffers before forwarding them, thus increasing latency. In addition, once these buffers begin to overrun, packets are dropped, and in the case of unreliable transmissions, lost entirely. This can lead to extended periods of time in which a user in completely out of sync with game servers or other users, leading to game halting or poor gameplay. Simulating this tendency of routers is key to developing a good scheme for overcoming these problems.

2.3 – Network Simulator

Network Simulator (NS) is a powerful tool for mapping networks in a controlled environment. It provides the means by which researchers can analyze the effects of variable levels of traffic on different types of routers, determine network behavior based on empirical data, and run a myriad of other experiments on the simulated network. NS is particularly useful for testing new protocols in a local environment and without hardware dependencies. Other means of developing and testing a new, unique protocol (meaning one that does not reside above another protocol) would be to physically set up

and configure real or emulated models to support the protocol. Even with this case, however, the researchers are limited to the hardware resources they can obtain for the experiment, and their network configuration would likely not be able to imitate the Internet.

NS supports many kinds of traffic, ranging from web traffic and FTP[42] to real-time video and audio streaming.[43] And because it is an open-source project, anyone can extend it to suit specific needs.

[42] File Transfer Protocol. Uses TCP.

[43] The modules for streaming media are not yet a part of standard NS builds, but may be acquired from researchers directly. http://perform.wpi.edu/downloads

3 – Methodology

In our study of multiplayer games, we took the following steps in order to acquire, analyze, and simulate the network traffic they generated.

1. To obtain packet data, we elected to use a packet sniffer, which is a program that captures packet data passing through a network card. The process of selecting a sniffer is discussed in sections 3.1.1 and 3.1.2.
2. Adapting the data taken by the sniffer to meet our statistical modeling needs became an issue, and we relate our solution for this in sections 3.1.3 and 3.1.4.
3. Once a suitable packet sniffer was acquired and supplemented with our parsing own tools, we were able to collect traffic data, and our next step was to choose the games that would supply this data (section 3.2).
4. Before analyzing the data, we were met with the issue of how it would be best simulated for each game. We found it beneficial to write a tool for performing a variety of operations on the data (section 3.3). This tool was useful in helping us analyze and parse our data, and also found use in generating code for our modules.
5. We were then prepared to conduct an analysis on the data (section 3.4). Our results for this may be found in chapter 4.
6. With a solid understanding of the patterns we found in our data, we then set out to build modules to simulate our findings (chapter 5).
7. Finally, we ran tests designed to measure the accuracy of our simulated data (section 5.2).

3.1 – Packet Sniffers

In order to do any kind of meaningful analysis or simulation, we needed to gather data from actual network traffic. In order to do this, we had to find some way of taking packets from the network and reading them. Fortunately, tools to do this have already been developed by several different groups. These, tools, called packet sniffers, record all of the traffic that the network card in a computer sees. However, different packet

sniffers have different sets of more advanced functionality, and it was difficult to decide on which one to choose.

3.1.1 – Sniffer Requirements

We started by specifying a set of requirements. Any packet sniffer we used had to record packets to permanent storage devices so that we could perform our own statistical analysis on the data. We also wanted to be able to maintain records of various games so that we could write a simulator that took these files and generated an accurate traffic pattern from them. Any packet sniffer we used also had to run in Windows because a second computer dedicated to capturing packets was not readily available on the same subnet. The packet sniffer also had to capture the time each packet was sent accurately because games tend to send many packets over a brief amount of time. Any sniffer we used would need to generate summary measures of the data as it was collected so that we could determine which types of statistical analysis we should generate. Finally, any packet sniffer we used had to be relatively inexpensive. There was no funding for most commercial sniffers and many had high fees.

3.1.2 – Selecting a Packet Sniffer

There are a number of sniffers available, and in accordance with our requirements, the following were considered.

Windump (http://netgroup-serv.polito.it/windump/)

Based on Tcpdump for Unix, Windump is very similar in that it only records packet data. There are a large number of packet sniffers available that use Tcpdump as a back-end, but virtually all of them are strictly Unix ports. To effectively use Windump, we would have to write a number of wrappers to transform the data into a more useful form for us.

Analyzer (http://netgroup-serv.polito.it/analyzer/)

Analyzer is a windows based packet sniffer with a good GUI and a decent, though somewhat lacking, set of features. However, the main reason Analyzer (which is in an

experimental stage of development) does not suit our needs is that logging traffic locks the rest of the program so that no results can be analyzed while the sniffer is sniffing. Without the ability to observe changes in traffic as they happen, we could not adequately make our observations unless we were to develop a means of marking game events over time.

Spynet (now called Iris - http://www.eeye.com/html/Products/Iris/)

Since this was first written, the Spynet packet sniffer was sold to eEye Digital Security and renamed Iris, evidently replacing the unique sniffer previous known as Iris. The Spynet packet sniffer is a part of a larger suite of networking utilities that are as sophisticated as they are expensive. The amount of functionality and statistical metering options was extensive, and this utility would have made an exceptionally useful tool in our project. However, the cost of the Spynet package is approximately one thousand dollars, so it was not a viable option.

Iris (replaced by Spynet – no official URL available)

Iris was a simple packet sniffer with a reasonably well-constructed interface. It clearly displayed the data on each individual packet and broke down the header data for each layer. However, it did a poor job of sorting aggregate packets, and had a major flaw in its lack of a good logging implementation. The data the former Iris collects goes directly to memory; and with a machine running a game at the same time, it can cause serious memory usage problems when setting the packet cache to a reasonable size. When logging to disk, Iris also stops recording packets entirely, missing those that arrive while the logging operation is happening. The disk access also uses large amounts of system resources, which causes problems in games.

Commview (http://www.tamos.com/products/commview/)

Commview seems to fit our requirements very well. It is a robust sniffer with the ability to log packets according to rules we set, to take a number of statistics, and to generate reports periodically. It does not have the logging stage/observing stage

restrictions of Analyzer, and it is one of only two sniffers ported to Windows that appear to be a finished product. For these reasons, we decided to use it.

Ethereal (http://www.ethereal.com/)

Ethereal is a widely ported sniffer that has all the versatility of Commview, but does not generate statistics as well, and does not graphically display them. Ethereal would have been an equally good choice as Commview for our project due to its exceptionally well designed interface and data export functions, but was unfortunately not considered before the project began. Commview remained suitable after we found Ethereal, so we did not feel the need to switch sniffers. However, for most packet sniffing purposes, such as further research in network gaming, we would recommend Ethereal.

3.1.3 – Usage

After several recorded traces yielded no useful information due to configuration errors or the traces missing vital parts of each game, we developed a methodology to use when recording a play session. One of the initial problems we had, before we mastered using filters in Commview, was having extraneous packets from other applications in our traces. To minimize this, we closed all other Internet software before loading the packet sniffer. If we knew which IP addresses from which we were going to be playing the games, we would add filters to Commview to ignore all traffic except from those IPs. At this point, we would begin logging, and then proceed to normally load the game, find a multiplayer session, and join it. After completing the game, we would quit, and then stop the trace. This process allowed us to capture complete traces of joining, playing, and quitting the games without capturing unwanted packets from other applications.

3.1.4 – Issues

Despite developing this system, we had several problems with Commview. There was no way, other than recording by hand, at what time during the trace significant game play activity took place. While having this information turned out to be unnecessary

when we came to develop our simulator, it was difficult to gain an overall understanding of the traffic flows without it.

We also ran into a problem when logging packets. If the packet buffer in Commview is set to large, and most of the computers resources are in use (often the case when playing a game), when Commview logs to disk, it will stop recording incoming or outgoing packets. It was necessary to save the packets to disk frequently to prevent this problem, though it was still an issue if the computer lacked sufficient computational power. This frequent logging also caused problems playing the games. When Commview would begin logging the packets, frame rates would fall, often dramatically in slower computers, making it difficult to play the game. Even when Commview was not logging, the extra memory and processing power made a noticeable difference in some games.

However, these problems were over come by using a powerful computer. The gaming systems we used were significantly above the top of the line when the games we worked with were released and had sufficient power to run the games and run the sniffer at the same time. There was a 10% reduction in frame rates in Counter-strike under the most computationally intensive circumstances and virtually no noticeable frame rate reduction in Starcraft.

3.2 – Game Selection

With the myriad host of titles currently on the market, picking games that would be representative of the market as a whole was a difficult process. With several hundred viable titles from which to choose and not nearly sufficient time to even perform a preliminary analysis on them all, we decided to look at games that sold well using the rational that if they sold well, they would have many players. With that decided, we determined that it would be best if we chose from a list of games that one or both of us had played. This did not narrow the list a great deal, but we felt it was important to spend most of our time analyzing a game rather than learning how to play it.

After doing some preliminary analysis, we discovered several more criteria we needed to use in order to determine the first few games to examine in depth. It became apparent that nearly all games layered their own protocol on top of UDP, but there were

some that used TCP. Because the games that used TCP, most notably Diablo 2, were not representative of network games as a whole, we decided to exclude them. We also decided to select games from several different genres. Since both Starcraft and Counter Strike were familiar, best selling games in different genres, we decided to select them.

We considered several other candidates including Tribes 2, Asheron's Call, Age of Empires 2, and Quake 3. Quake 3 and Tribes 2 were both rejected because Counter-strike was a clear choice for a FPS due to its popularity, and we felt that comparing across genres was more important than comparing within genres. Age of Empires 2 suffered from a low online user base and significantly longer games than Starcraft. Finally, while we wanted to study Asheron's Call, the time it took to develop our analysis process precluded studying another genre.

3.2.1 – Starcraft Game Environment Issues

Starcraft is a real-time strategy game that revolves around constructing buildings and fighting units, and issuing commands that cause the units to move, engage enemy units, and perform other such tasks. Every game is played on a map, and there are a variety of maps available. There are three races from which a user can choose, and each has a balanced set of advantages and disadvantages over the others. There are a number of ways in which players can be competitively grouped. In a free-for-all, all players vie to be the last remaining player on the map. Players can also team up against each other and/or AI scripted "computer" players in myriad ways. In order to control as many variables as possible in our experiment, all games were played on the same map, and all were structured so that there were two teams of equivalent size; 2 vs. 2, 3 vs. 3, and 4 vs. 4 player games were recorded. In addition, the local player played as the same race in each game, and employed the same building strategy throughout.

3.2.2 – Counter-strike Game Environment Issues

Counter-strike is a modification to Half-Life that is distributed free over the Internet[44] for owners of Half-Life or as a retail product in most game stores. Counter-

[44] http://www.counter-strike.net

strike puts the players in the role of either a terrorist attempting to hold hostages, blow up landmarks, or assassinate a VIP or a counter-terrorist agent trying to thwart the terrorists. To play, the player must connect to a server, either located on his or her own machine or one across the network. When more than 1 player joins this server a game begins.

Games are divided in two ways. All games are played on a map, each of which has its own set of objectives. Most involve either the Counter-terrorists (CTs) attempting to rescue a set of hostages from close to where the Terrorists (Ts) start the round or the Terrorists attempting to plant a bomb close to where the CTs start. Each map is played several times (rounds). Each round ends either when the victory conditions are met, time runs out, or when one team has been totally eliminated. At the start of each round, both sides are allowed to buy weapons and ammunition with the money they earned from previous rounds. The better each team did the round before, the more money they have to spend. Once each team has equipped, they attempt to wipe out the other team with their weaponry or complete the objective, though the former ends far more rounds than the latter.

3.3 – Tool

Once we had picked the games, played them several times, and recorded some game sessions, we began to analyze the packet logs. We almost immediately realized that although the native Commview statistical tools were reasonable for getting a rough idea of overall bandwidth, they were not satisfactory when trying to visualize traffic senders and receivers. We also felt that we needed to see several types of graphs beyond the packets per second and bandwidth graphs that Commview generated. Because of these problems, we set out to design and write a tool to allow us to generate statistical analysis on our data and aid in creating graphs.

The first step in developing this tool was to decipher the file format that Commview used when outputting packets. This was far easier than it could have been because we were able to load the log file into Commview and look at the data in plain text rather than the hexadecimal format in which it is stored. The biggest stumbling block was figuring out where in the Commview header portion of the packet the time and direction information were located. The rest of the packet, including transport and

network layer headers, was saved exactly as it was when sent or received. For an example of a Commview packet broken down into its component parts, see Appendix A.

Once we had a firm idea of how the packets were saved, we developed a Java application to parse the file, load the data into classes, and perform some statistical analysis on them. Our first few attempts were complete successes, but when we moved from small traces (between 1 and 5 minutes long) to longer traces, it became apparent that Java was unable to handle that amount of data. We could load somewhere on the order of 8,000 packets into memory, but any more than that and the Java Virtual Machine would run out of available memory or crash. Changing the GUI from the Swing toolkit to the AWT (Advanced Windowing Toolkit) helped, but we were still unable to load a 20-minute, 100,000-packet game of Starcraft. At this point we also started running into issues with graphing the packets.

When developing the tool, we had initially planned for it to perform all the graphing functionality itself. However, finding a good graphing package for Java was difficult, and the one we chose turned out to be unable to render a larger percentage of our sample data. We decided at that point to output the packet data into a comma delimited file and import this file into Microsoft Excel for graphing purposes. It was initially difficult to determine the correct procedure for rendering the kinds of graphs we wanted from Excel. However, with some practice and modification to the output the tool generated, it became an almost trivial process to generate useful graphs. With these graphs we were able to determine characteristics about the traffic that each game generated. By the end of the project, the tool was capable of outputting size and time bucket files used in our NS simulation, files containing bandwidth per second, and trace files containing time and size with an option to include IP addresses. Throughout the project it was also used to auto-generate code for loading the buckets in our NS simulation, though this was determined to have limited usefulness and excluded from the final product.

3.4 – Analyzing Game Traffic

In order to build an accurate simulation, solid knowledge of the activity being simulated is necessary. There are several factors that are important to determine before

creating a network simulation. Most important among these are bandwidth, both average and instantaneous, and packet throughput. In order to determine what kind of traffic pattern each game generated, we took a number of steps. First, we gathered several traces of the same type. For example, for Counter-strike we took traces of games on the same server with the maximum number of players. With this data in hand, we ran it through the tool to get basic statistics that showed average bandwidth, packet sizes, time elapsed, and a few other metrics. Using this data, we were able to determine if we needed more data or if it was safe to proceed to graphing.

Once we determined that we had enough solid data so that a graph would be reasonably representative of typical, similar game sessions, we loaded some of the files generated by the tool into Excel and generated scatter plots. We generated a graph of the entire trace each time, and if there was an area that appeared different, we would graph a smaller time slice that contained that particular feature. As a general rule, the traffic from each IP address was contrasted with another trace from a different game session that we thought would have a similar traffic pattern. It was determined early in the process that most of the players in a given game generated similar traffic patterns, so it was generally unnecessary to look at more than one player's trace.

4 – Game Traffic Analysis

Running games and recording the packet data produced a great deal of information, which we used to analyze strategies for our NS application. There are a number of ways in which the data we acquired can be structured for viewing, but we decided that the best representation is in the form of annotated graphs.

Section 4.1 relates results found in our analysis of Starcraft game traffic. First, we studied the relationship between traffic received by each remote player in a typical 6-player session. Next, we compared the traffic generated by each of several games of the same size, and then games of varying size.

Section 4.2 is devoted to our analysis of Counter-strike. As this game has a client/server architecture, we ran an analysis for both the typical client and servers running sessions of varying size. Throughout this process we were mainly concerned with the size of and time between each packet. These results would produce the data needed for simulating these games in NS.

4.1 – Starcraft Traffic Data

Data collected for this particular game was graphed to illustrate the ways in which games varied by number of players, and how they varied across game sessions of similar size. The purpose of collecting this data was to determine the means by which it would be possible to create a simulation for a typical game of Starcraft.

All Starcraft data was collected on the same machine. This machine's relevant specifications are as follows:

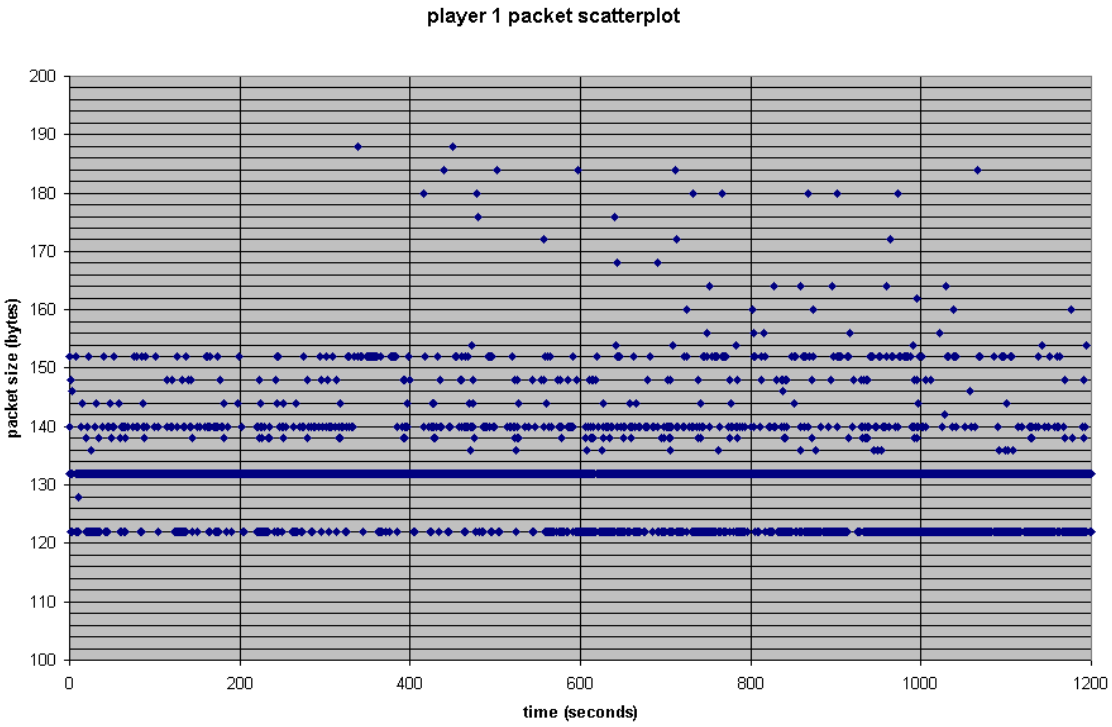
- Intel Pentium III 800mhz processor with 100mhz FSB
- 512 megabytes PC-100 SDRAM
- nVidia geForce2 3d graphic accelerator with 64 megabytes of DDR SDRAM
- UltraWide SCSI hard drive interface
- 10baseT network card connected to 608/108mbps DSL modem
- Windows 98B Operating System running Commview version 2.6 (build 103) packet sniffer

Controlled in-game variables were as follows:

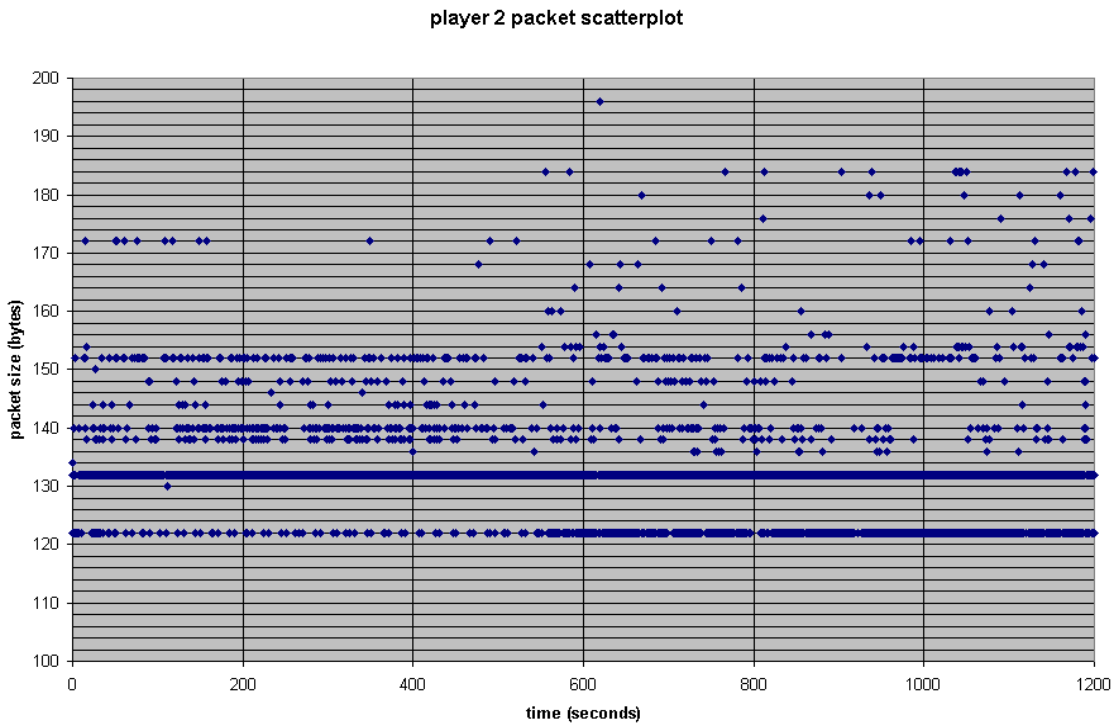
- Games were played using Starcraft: Brood War version 1.7.
- Local player logged on to Battle.net using the USEAST gateway, and created the game sessions. The game type was Top vs. Bottom for each.
- The same map was used for every game. The map is called Big Game Hunters, and can be found in the maps/broodwar/webmaps directory from where the game was installed..

4.1.1 – Comparing Data Streams from Remote Players

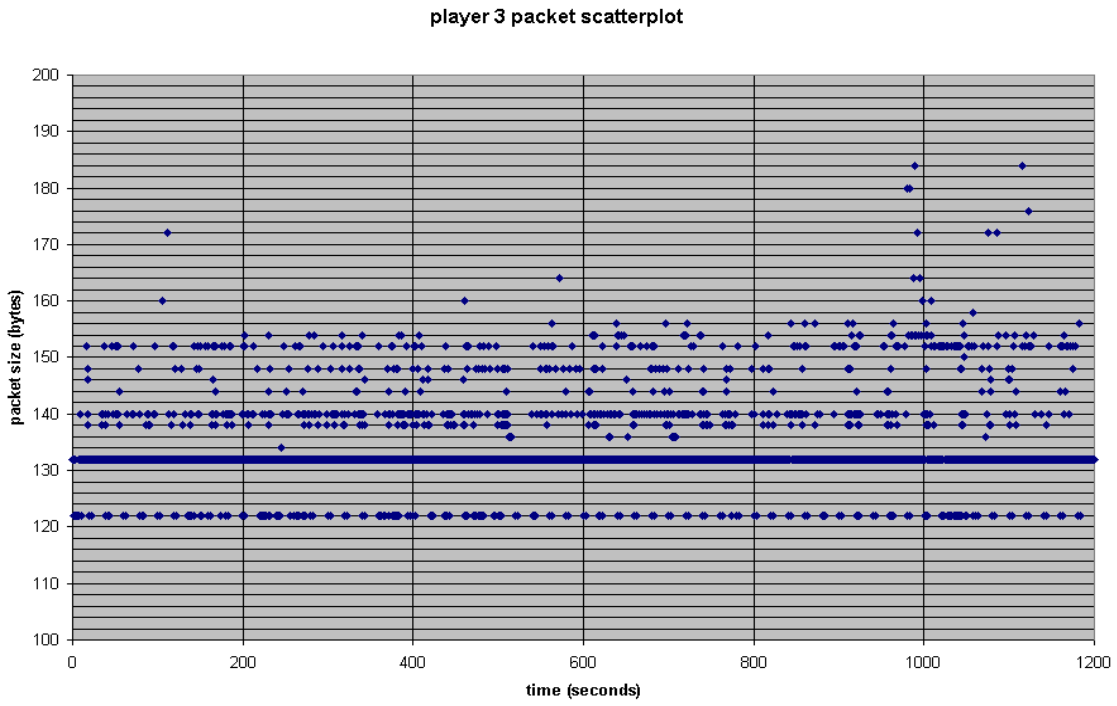
The following charts represent the traffic received by the local player from each of the 5 other players in a 6-player game. They serve to show that individual players generally produce similar traffic patterns in comparison with each other. Each graph represents 20 minutes of packets received from each remote player. Any packets lost in transmission are not represented on these graphs, as they never arrived.



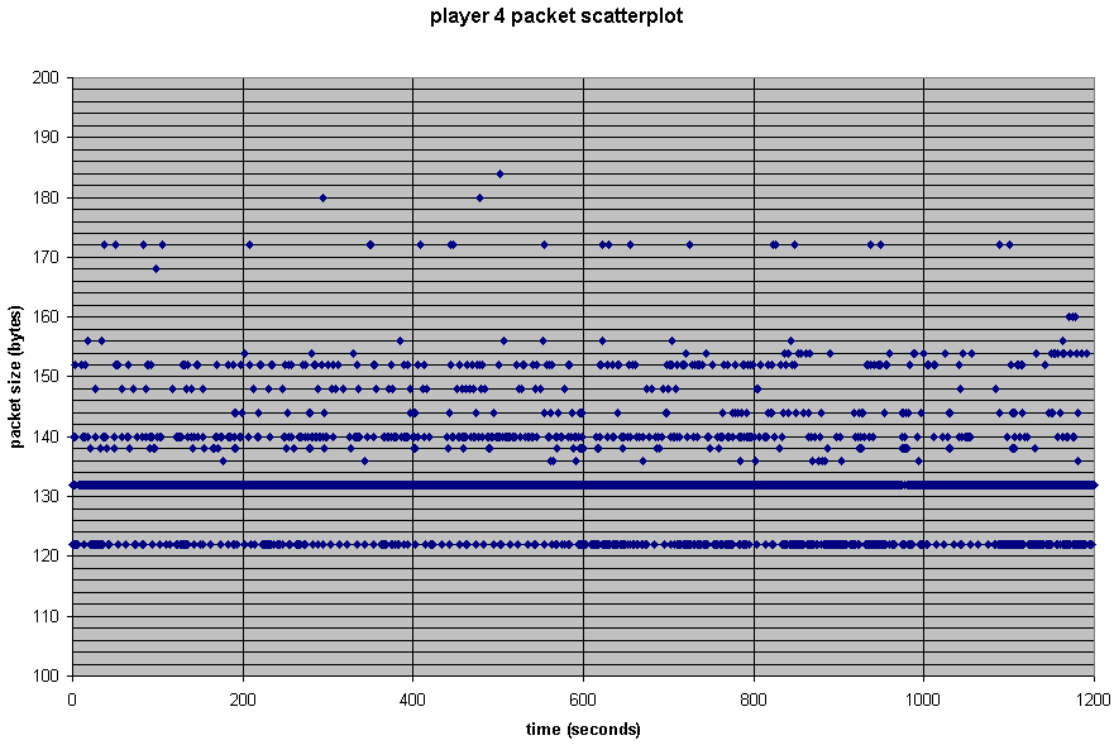
The bands of points are separated by multiples of 4 bytes in size, with 132 bytes comprising the solid majority of points. The density of each band indicates the frequency with which each packet size appears.

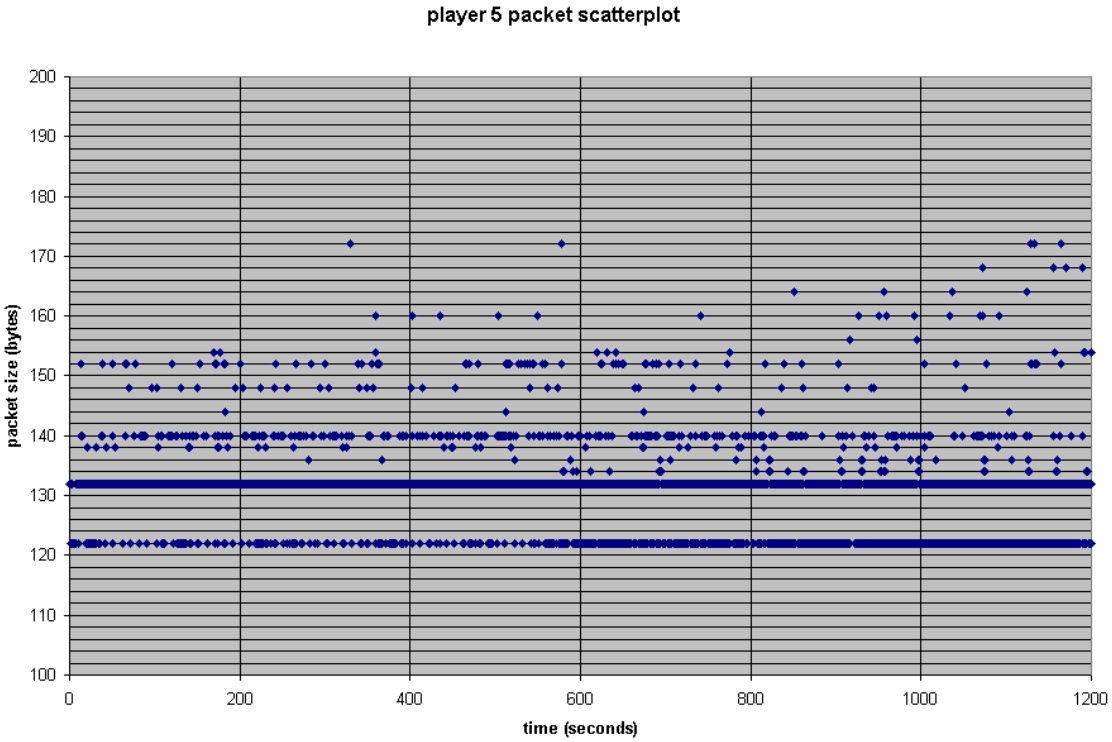


The distribution of points on this graph is very similar to that of player 1; the vast majority of packets are 132 bytes in length. The second-most dense band is again at 122 bytes. There also seems to be a trend forming in the number of 122 byte packets per second increasing at about 480 seconds into the game.

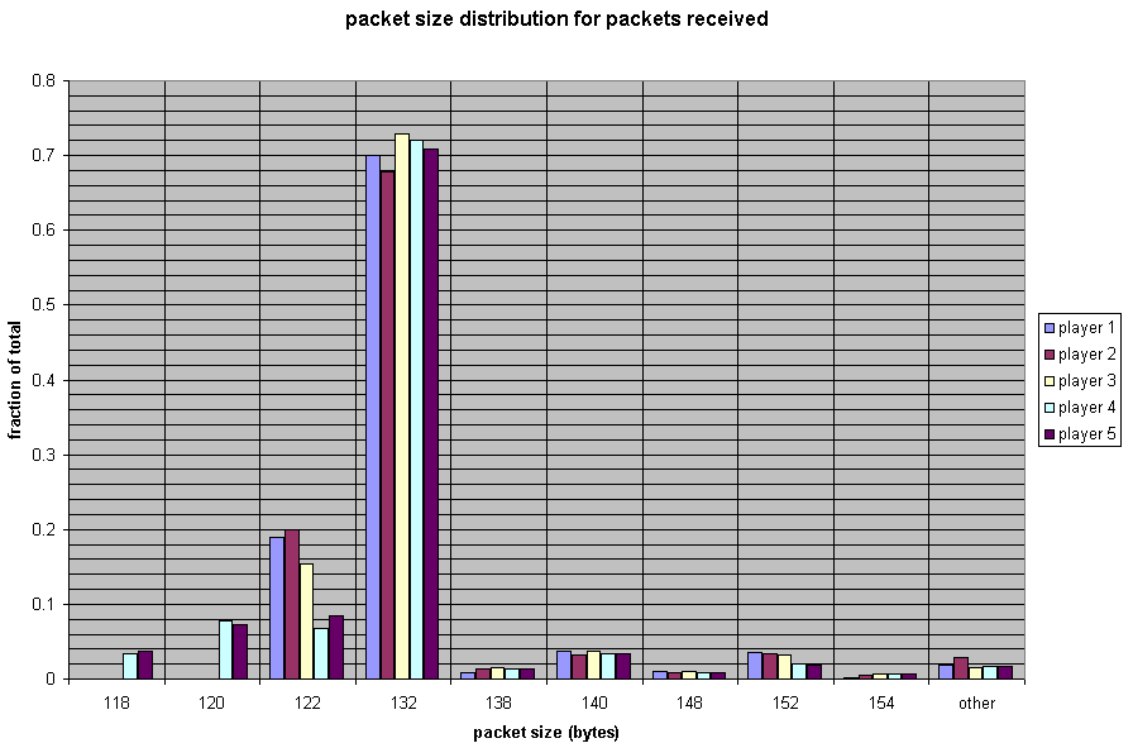


At this point, it appears that all players might adhere to the same general distribution of points, except that this one does not express the aforementioned trend in the 122-byte packet band.



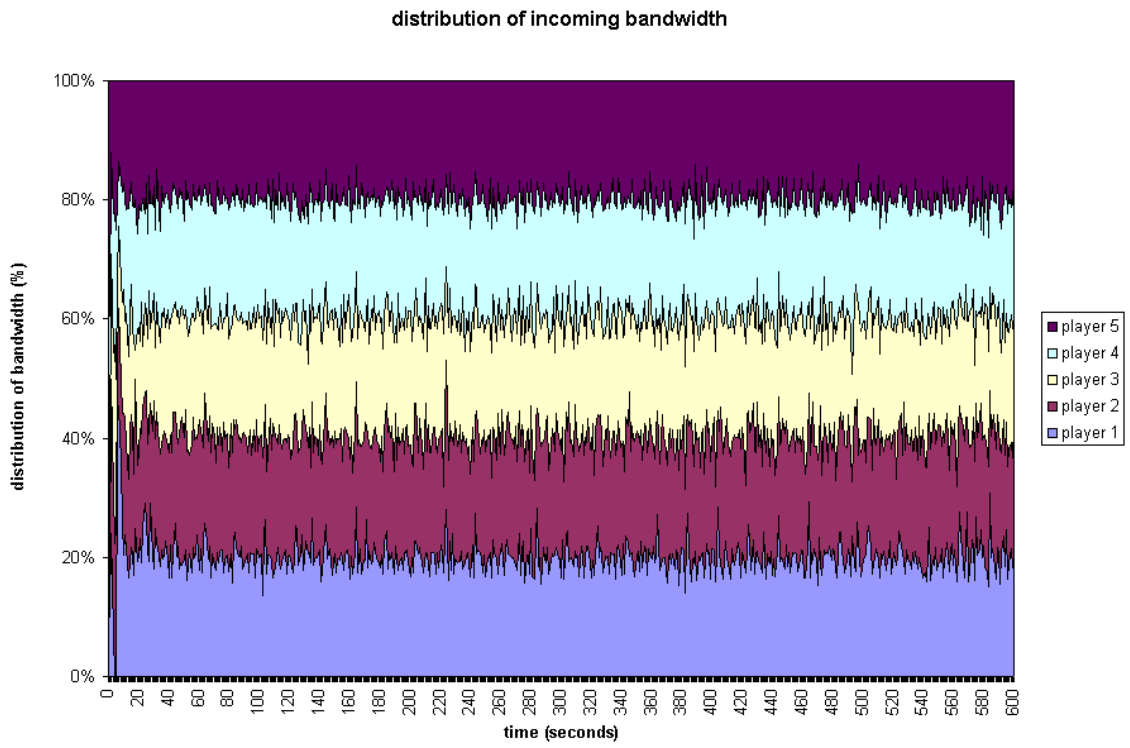


The distribution of points across each of the graphs is strong evidence that each player sends roughly the same pattern of traffic. It was this observation that led us to conclude that there was no need to account for differences between incoming packet streams in a simulated game of Starcraft, as they are all statistically equivalent. However, since most of the plot points on our graphs overlap, it is difficult to derive statistical information from them alone. Following is a graph of the relative frequency of packet sizes across the 5 remote players in this Starcraft session.



The distribution is clearly around 70% 132-byte packets, with 120-122 byte packets comprising the next largest group. This relates to the bands of packets, at these size levels, on the scatter plots. With only relatively few packet sizes used by Starcraft, it is visually easy to associate those represented here with their levels on the scatter plots.

Finally, a comparison of the bandwidths generated by each of the remote players solidifies the argument that they behave very similarly.



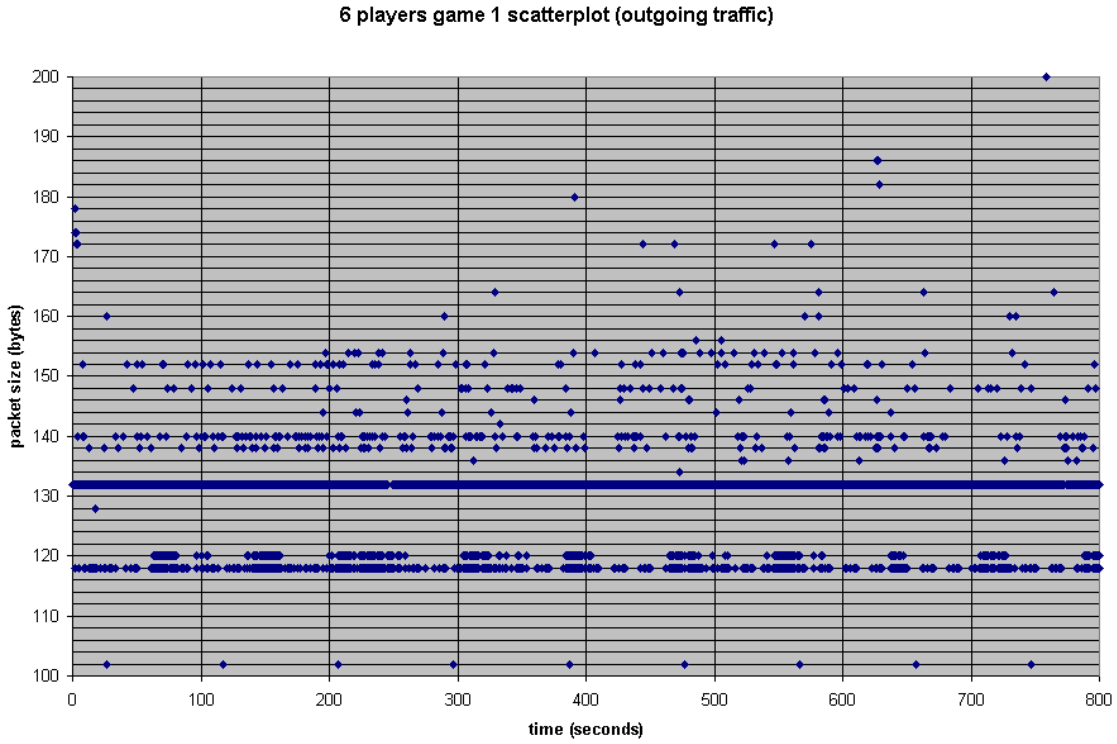
	Player 1	Player 2	Player 3	Player 4	Player 5
Average bandwidth (bytes/second)	680.4592	676.1897	669.1414	665.8436	675.1947
Standard deviation	137.2658	114.8658	113.5378	100.0684	104.0294
Std dev/mean	0.201725	0.169872	0.169677	0.150288	0.154073

This graph illustrates the division of total bandwidth by the remote players. From the graph, it is clear that they each contribute nearly equally to the total bandwidth used. It is for this reason we decided it would not be necessary to differentiate between remote players in our simulations. In addition, this graph suggests that each player is sending the same amount of data to every other player, as the amount of data received from each is the same when perceived by the local player.

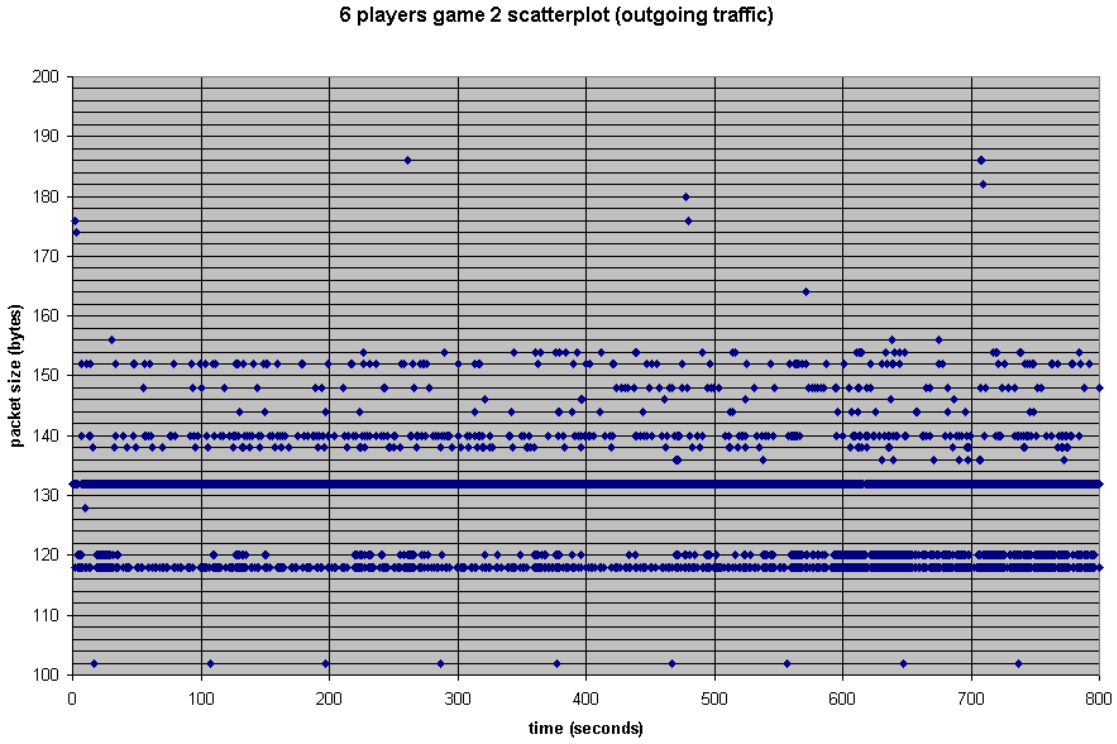
4.1.2 – Comparing Outgoing Data Streams Across Similar Games

The prototype for our NS game application was intended to simulate a typical 6-player game of Starcraft using the probabilistic methodology described earlier in this document. A typical game, however, was as yet undefined, so it was necessary to run a

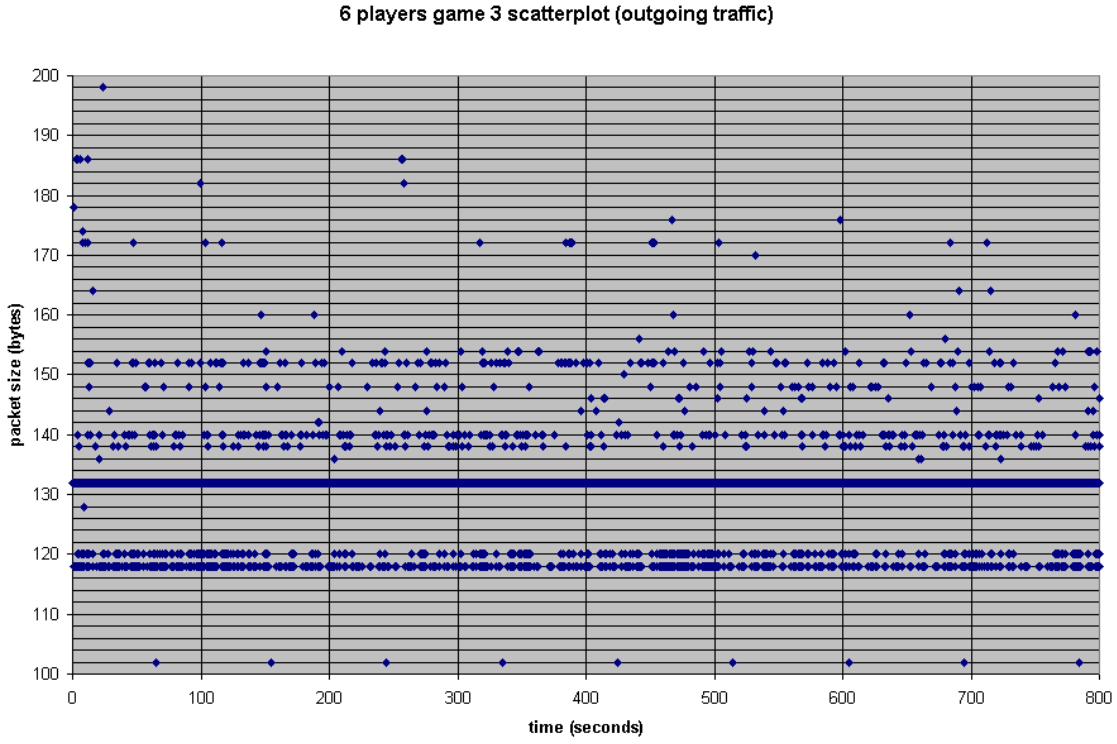
number of game sessions and compare them. Of the ten sessions recorded, we have decided to display four as a succinct demonstration of their overall similarity. All four graphs were cropped to 800 seconds for the sake of comparison. Every game has yielded the same pattern throughout, however, and it should be noted that showing only the first 800 seconds does not limit the analysis.



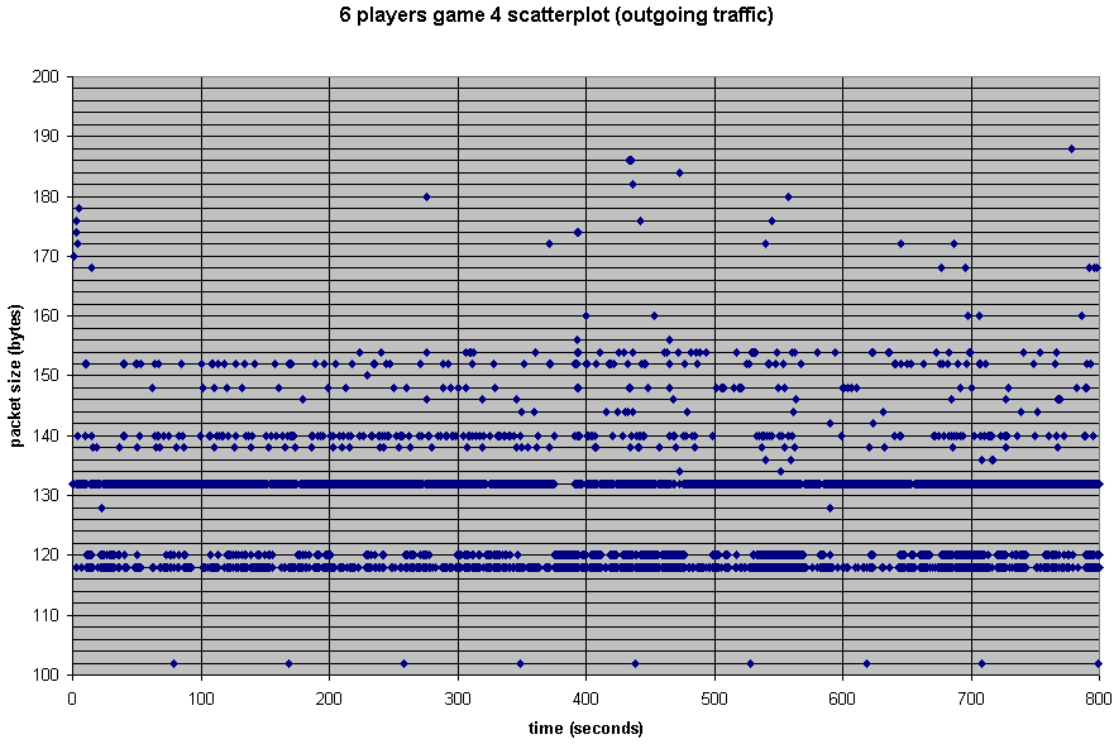
All traces looked like this one. The majority of packets are 132 bytes in size, just as the incoming traces showed. There are two packets that lie out of this plot's range. They are just over 500 bytes in size and are delivered to Battle.net servers for purposes unrelated to gameplay, itself.



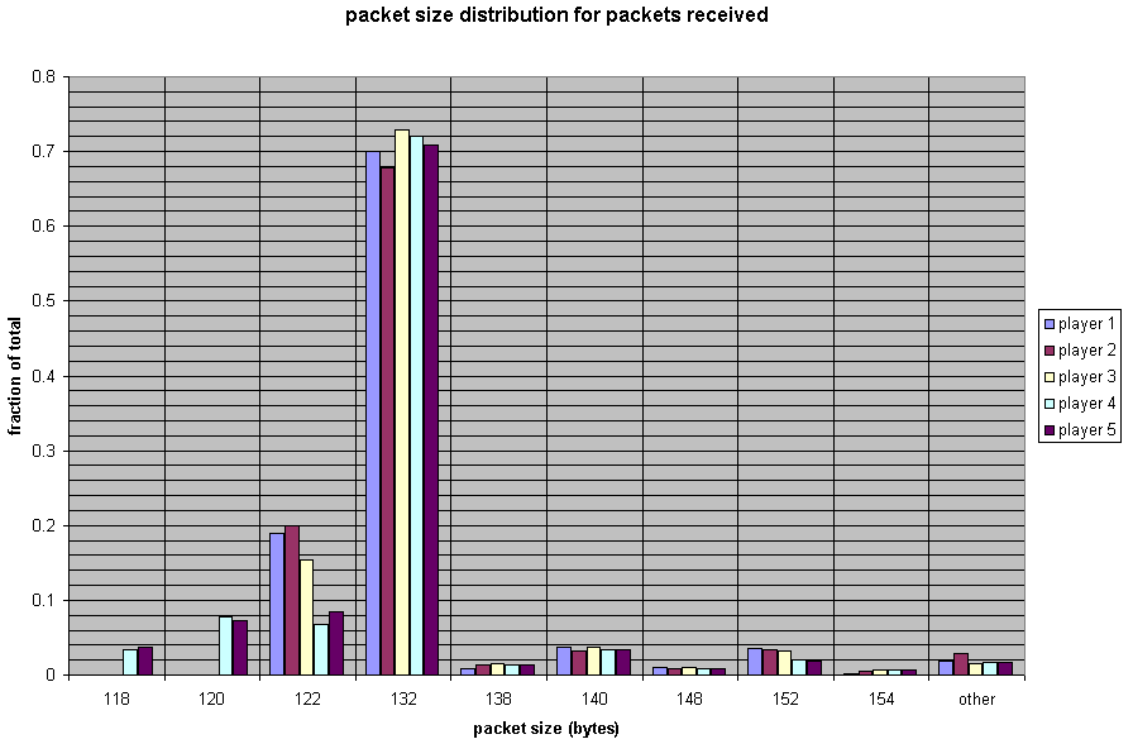
The similarity between games is more evident with each graph.



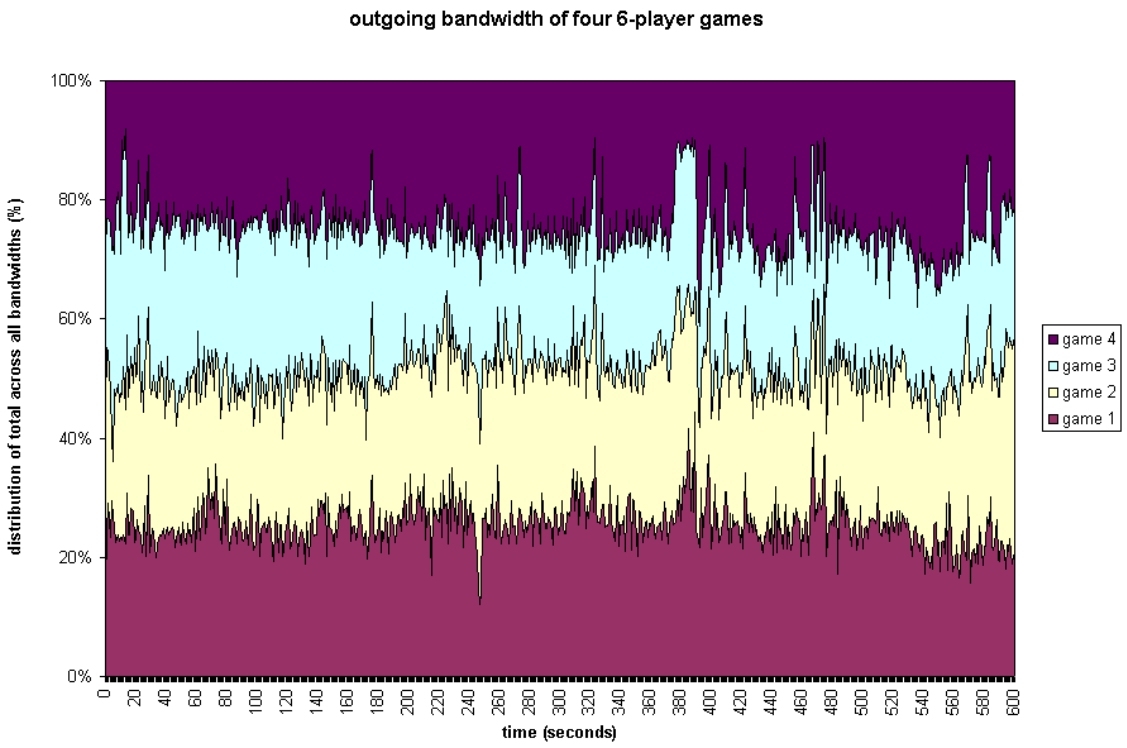
The 100-byte packets that appear to be uniformly distributed across the time axis are not sent to any of the other players in the game, but actually represent some kind of persistent connection with a Battle.net server.



As each player in a Starcraft session sends the same amount of data to every other player, it follows that the packet size distribution of the outgoing traffic should match that of the incoming traffic. Comparing the following graph with the previous size distribution graph illustrates that this is true.



It became apparent very early that acquiring a typical 6-player game's simulation was as simple as choosing a lengthy trace as input for our simulator. Games of other sizes (2, 4, and 8 players) also proved to be similar within their sizes; the 6-player comparison shown in this document is only an example of this.



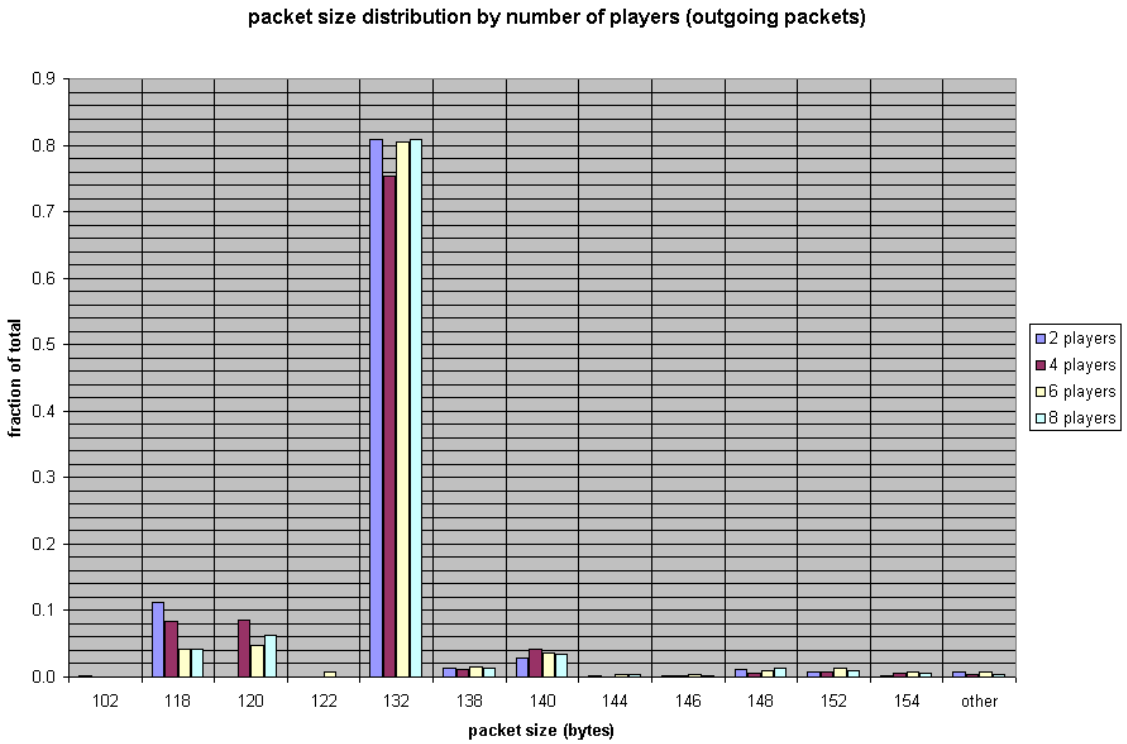
	Game 1	Game 2	Game 3	Game 4
Average bandwidth (bytes/second)	3372.932	3403.082	3096.799	3318.153
Standard deviation	589.523	493.6111	627.2109	820.4449
Std dev/mean	0.174781	0.145048	0.202535	0.24726

The bandwidth graph in this case is almost evenly distributed between the four separate sessions. Again, the significance of this is that a typical game of Starcraft of the same size is, in reality, any game of that size. The anomalous data spike at 300 seconds has been attributed to a period of unusually high latencies in game 4 at that time (noted during gameplay), resulting in fewer packets sent.

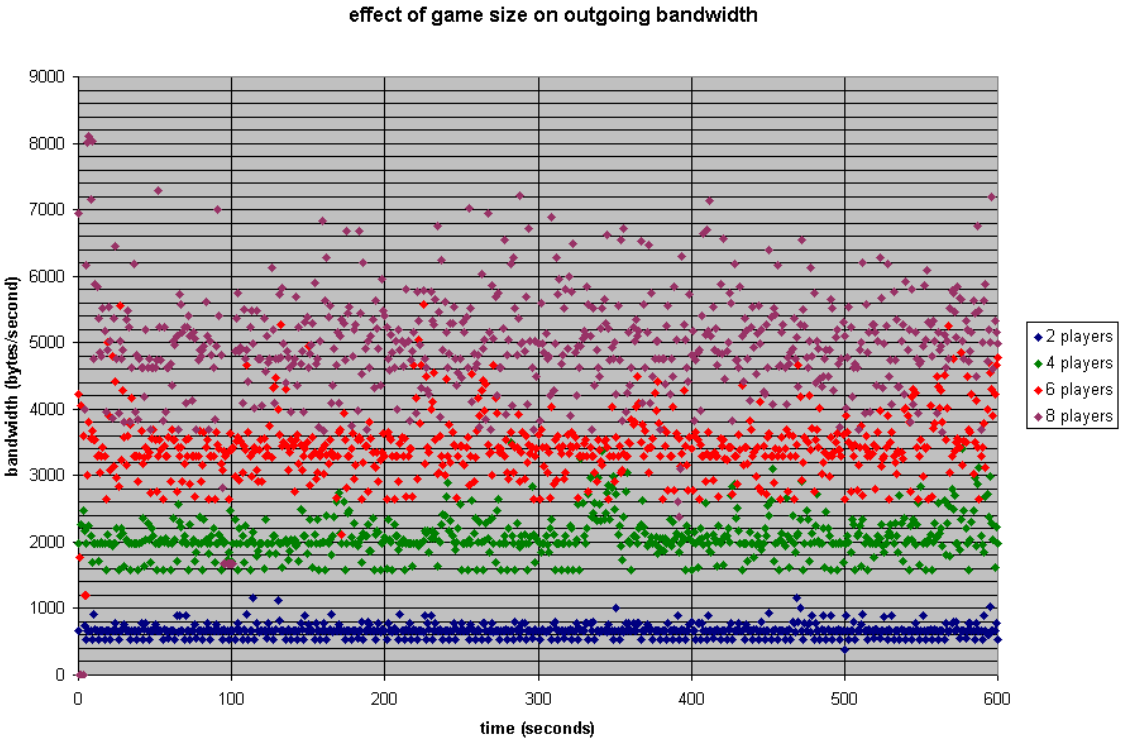
4.1.3 – Comparing Starcraft Games by Number of Players

The number of players does not have a direct affect on the average packet size, as is indicated by the following graph. Each ring represents a game, and each segment a percentage of the total packets from each trace. The graph shows that 132 is the most common packet size, with significant numbers of 118, 140, and 120-byte packets

appearing in the trace. This matches closely to the bands of points on Starcraft packet plots.



The average time between packets is very difficult to produce with our tool due to innate problems with floating-point numbers in Java. Packet times for the Starcraft data are generally poorly represented by standard graphs, so we decided to express the next portion of our analysis using graphs of average bandwidth.



	2 players	4 players	6 players	8 players
Average bandwidth (bytes/second)	662.2995	2076.483	3437.764	4953.268
Standard deviation	106.4026	323.7356	495.3148	896.1799
Std dev/mean	0.160656	0.155906	0.144081	0.180927

This graph illustrates that the amount of bandwidth used by Starcraft sessions is proportional to the number of players in the game. A 2-player game sends around 650 bytes/second. Each session depicted here is about 1400 to 1600 bytes/second from its neighbors, though the 2-player game runs at only 660 average bytes/second. Thus they do not only appear to be related, but linearly related.

As the number of players increases, so does the variance in bandwidth consumed. This is likely due to the increased probability that any player in the game is experiencing a high amount of latency to another. Since the game runs in a general lock-step fashion, this directly affects the rate at which data is sent by the rest of the players in the process.

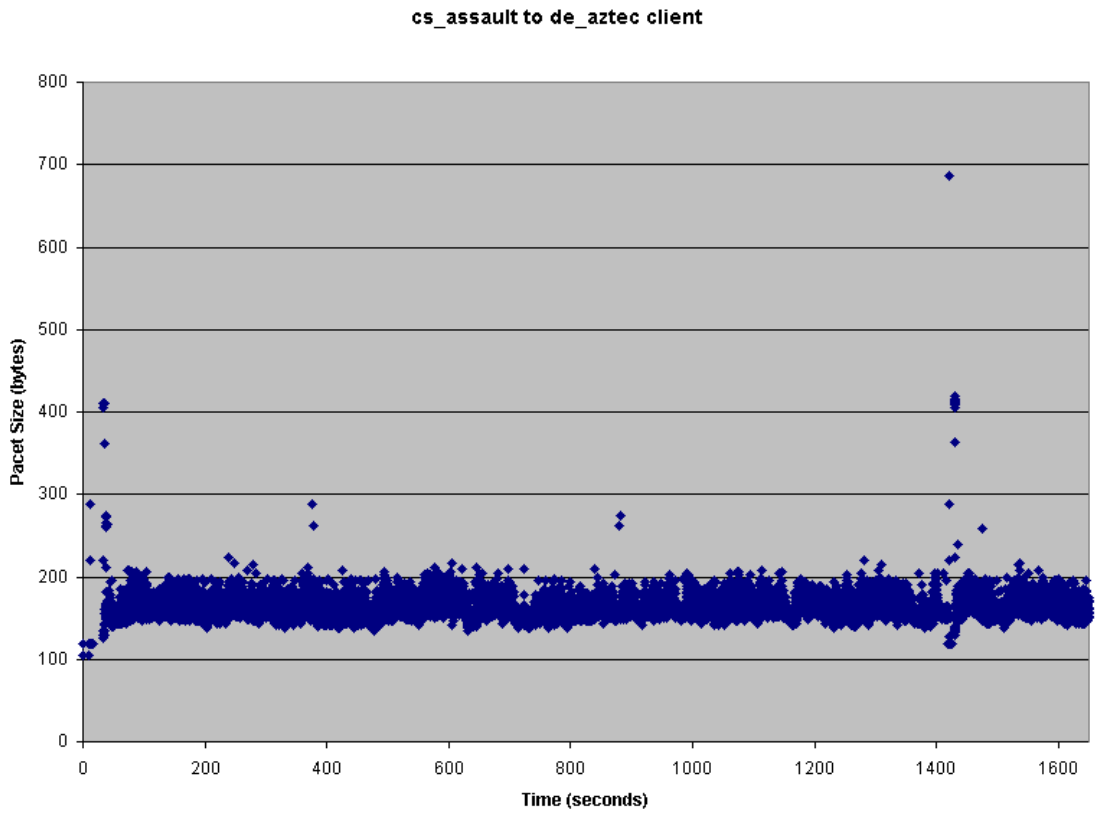
4.2 – Counter-strike Traffic Data

While a Starcraft game has very little deviation throughout its run, Counter-strike, especially on the server side, has a distinct repeating pattern. Throughout the section below, the names of the maps played during the trace will be contained in the title of the graph. All Counter-strike data was collected on the same machine. This machine's relevant specifications are as follows:

- AMD Athlon 800mhz processor with 200mhz FSB
- 256 megabytes PC-100 SDRAM
- nVidia geForce 3d graphic accelerator with 32 megabytes of DDR SDRAM
- ATA-66 hard drive interface
- 10baseT network card connected to WPI LAN through residence hall connection
- Windows 98 v4.10.98 Operating System running Commview version 2.6 (Build 103) packet sniffer
- Counter-strike version 1.3 on Half-life version 1.1.0.8
- Maps used (from standard install): de_dust, de_aztec, and cs_assault
- Games all played on LAN server located on WPI network

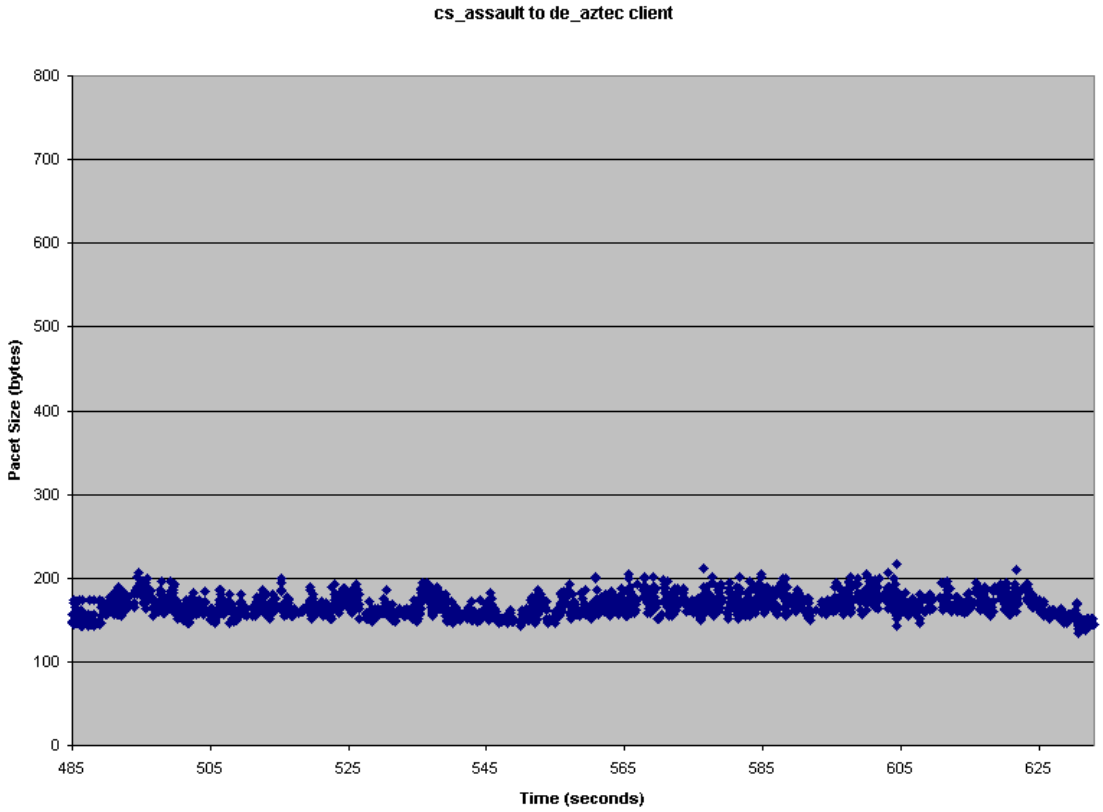
4.2.1 – Client Traffic

The client-server architecture of Counterstrike creates a specific set of traffic patterns. For example, regardless of how many players are in any given game, the data sent by the client looks remarkably similar. There are very few outlying data points, and most of the packets are of nearly the same size. Take for example, this graph:

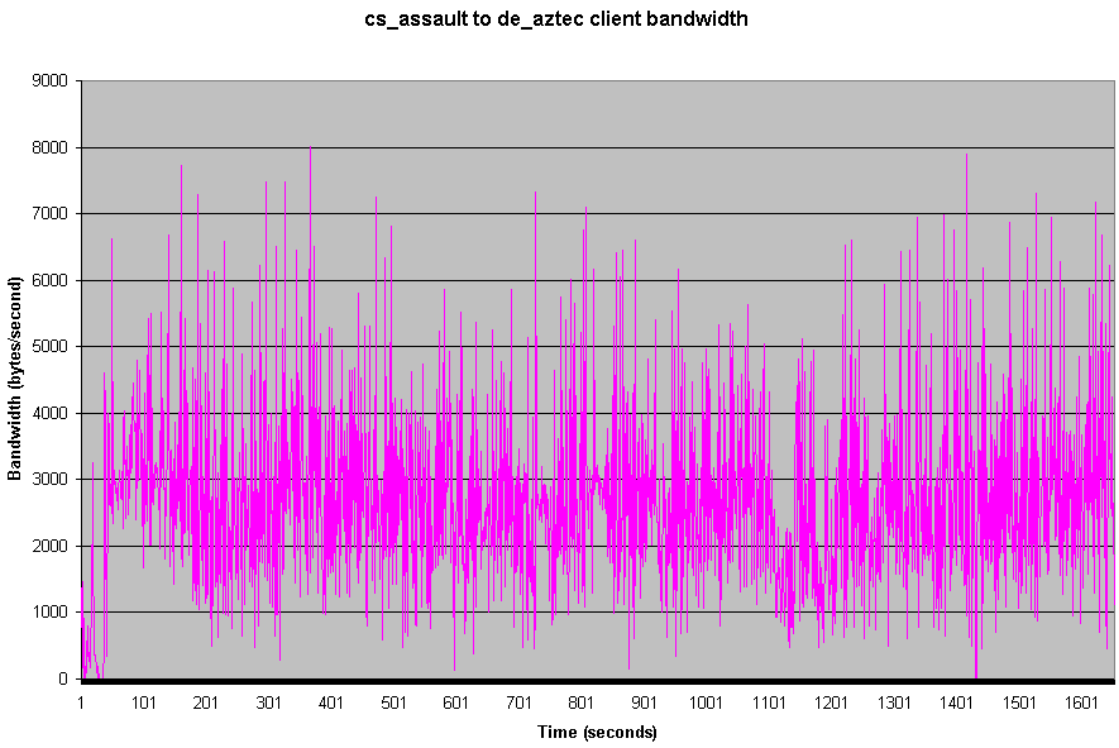


The mean size of the packets sent by the client is around 165 bytes with a standard deviation of 40 bytes. While the data are quite variable second by second, over several minutes, even across player deaths, the data looks quite uniform as is shown above. Due to the total lack of variability by player number, (there were between 24 and 32 active players during the time this graph shows), it seems reasonable to conclude that the client data rate is not dependant on the number of players.

The round cycle becomes less clear as the number of players on the server decreases. The play session below once again covered a map change but in this particular session, the number of players varied between 7 and 11. Another key difference is that the first map, de_dust, tends to run quickly compared to other maps and rounds average around 2 minutes. However, we can still see the end of rounds by observing the large packets sent out whenever a round ends.

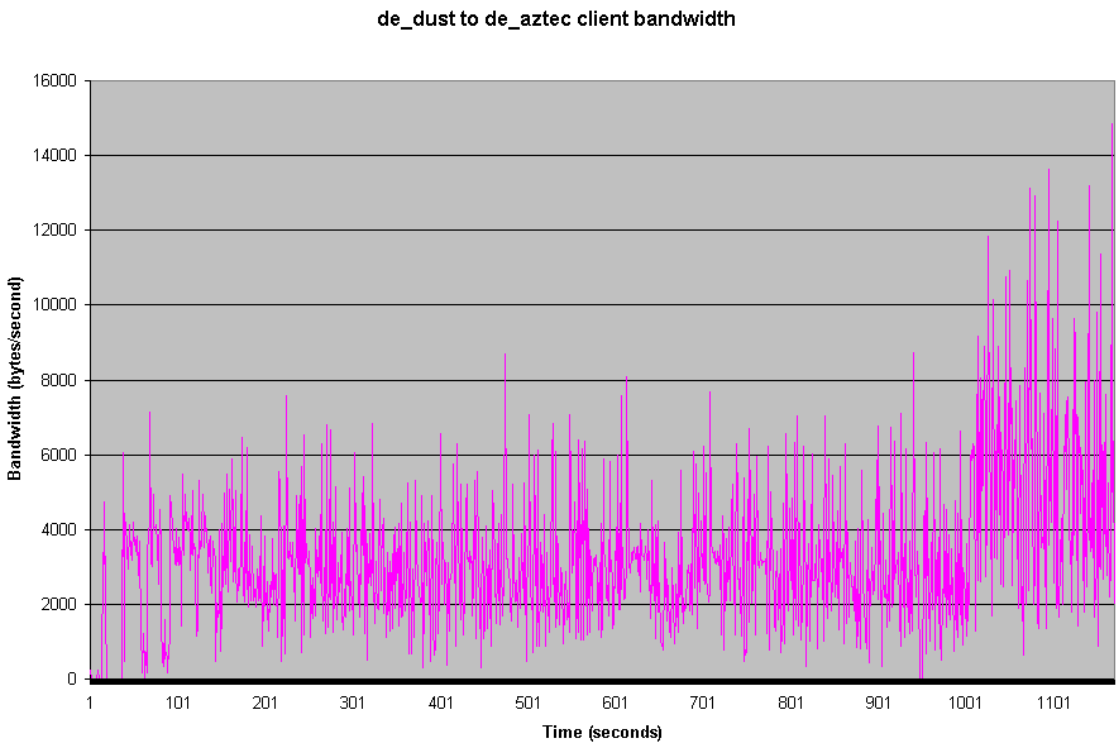


The traffic generated by one round of Counter-strike is a nearly flat line from start to finish. It would appear that the client sends updates constantly, even when the player isn't providing input. There is an interesting fall off at the end of this graph, more than likely corresponding with the end of round/start of round sequence. An interesting feature is the slight increase in average packet size at 555s due to an unknown cause that is also reflected in the bandwidth graph below.



Average bandwidth: 2693.92 bytes/s
Standard deviation: 1324.82 bytes/s

It is interesting to contrast the bandwidth over time with the packets sent. While the granularity on the bandwidth graph is significantly better than the packet size graph above, it still shows distinct drop offs in bandwidth usage per second. This is more than likely the result of prolonged periods of waiting still within the game, as it seems to happen several times per round.



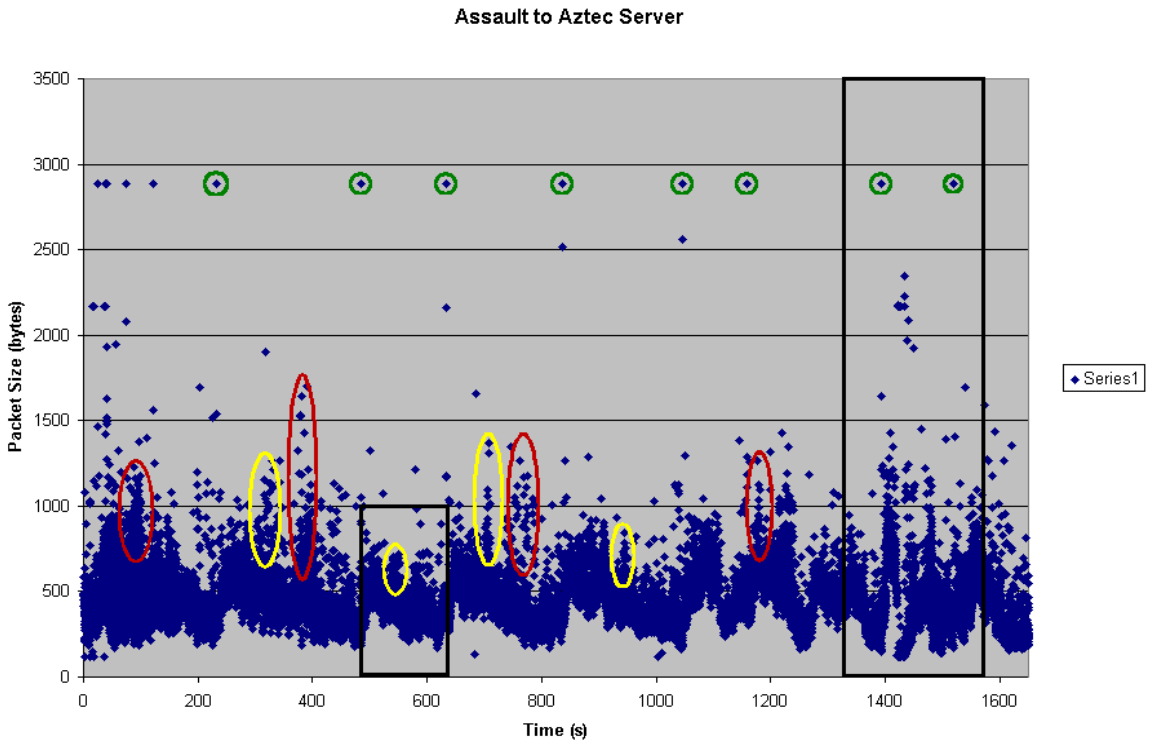
Average bandwidth: 3209.42 bytes/s
Standard deviation: 1958.83 bytes/s

Examining this graph and contrasting it to the assault to aztec client bandwidth graph above, it is interesting to note the differences between them. During the first map in both cases the bandwidth usage looks quite similar but after the map change from dust to aztec at time 950s, the bandwidth spikes here jump dramatically in both frequency and size. However, the same phenomenon is present in the server graph taken during the same play session. The assault to aztec play session has a slight increase in bandwidth usage, but it is not as dramatic as it is in this case. However, the rounds were quite short and bloody during the beginning of the aztec map in this case, which may explain the increase in bandwidth usage.

4.2.2 – Server Traffic

The traffic generated by the server, however is quite different. There is a large amount of variability based on the number of players alive on the server at the time. This variability affects both the size of the packets sent and their frequency. For very large

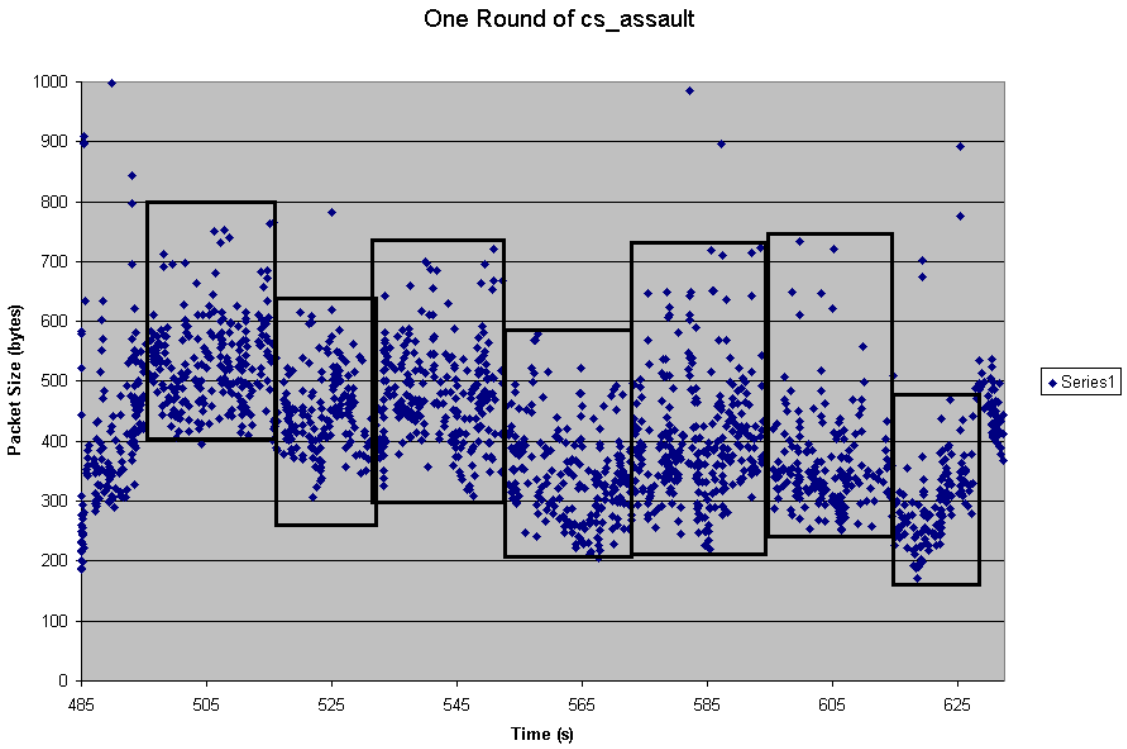
numbers of players (24-32) the rounds become rather obvious by the variation in packet size. Take for example the graph Assault to Aztec Server.



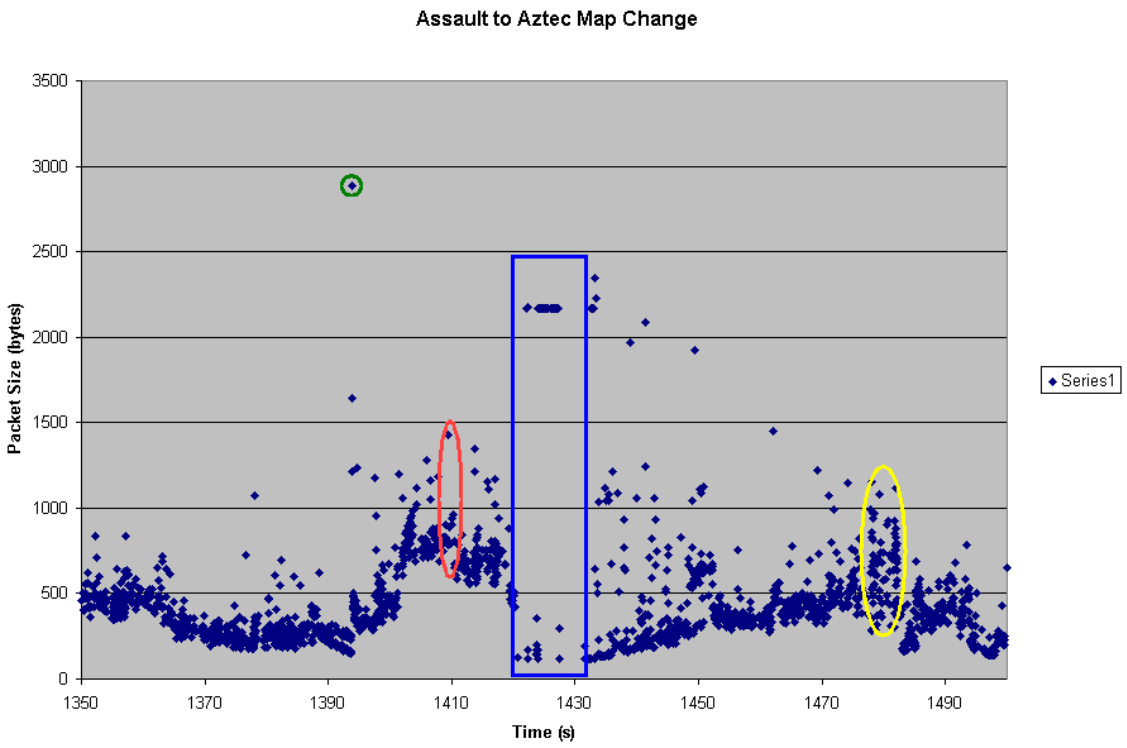
Total Packets: 20815
Mean Size: 465.25
Size Standard Deviation: 205.95
Largest Packet: 2886
Smallest Packet: 122
Mean Time Between Packets: 0.079
Total time: 1650

The game started on the map cs_assault, which is a hostage rescue map. The player was a terrorist. Rounds can be distinguished by the slow decline in the packet size sent from the server. For example, one round goes from approximately 500s to 650s, featured above in the first small box. It seems likely that the large packets of nearly 3000 bytes, circled in green are round initialization or round termination packets, as they strongly correlate with the end of the decline of packet size. The red and yellow ovals correlate with large firefights within the game. The red ovals also correspond to firefights in which the player died. They seem to have larger packets than normal

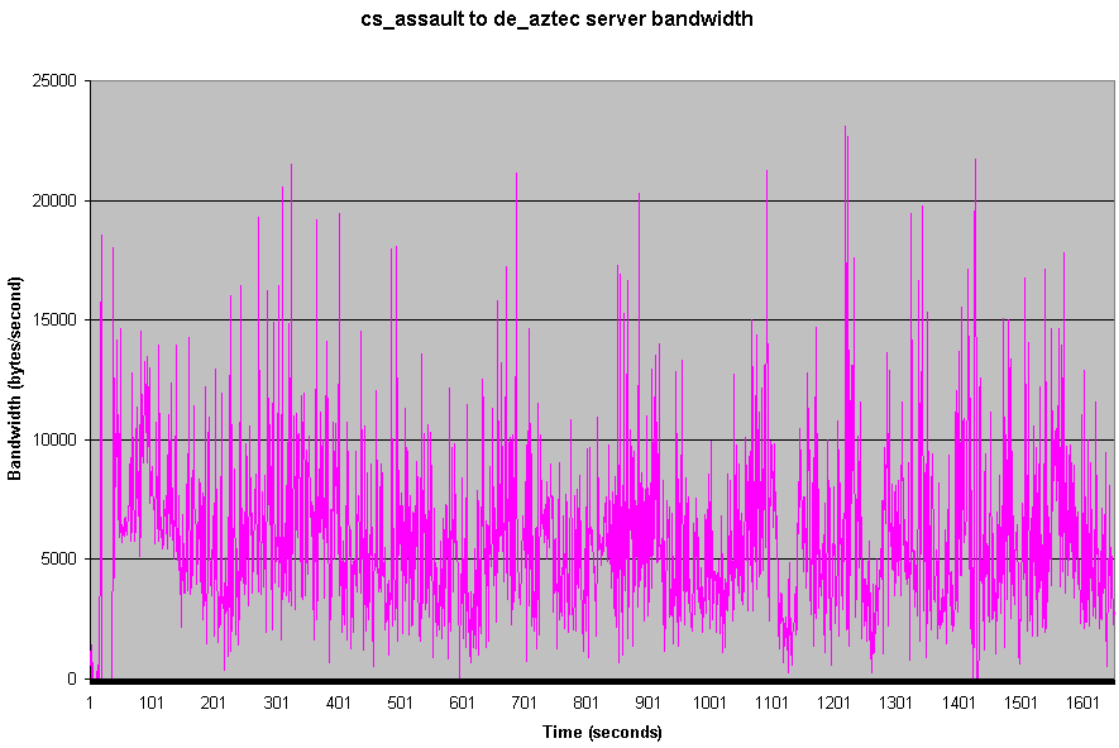
firefights, possibly due data sent to the player containing chase mode data. Another important feature is the break in the graph around 1400 seconds, indicated by the second large box. At this time, the map was changed to de_aztec almost immediately after the start of a new round. The map change can be seen more clearly on the graph labeled Assault to Aztec Map Change.



The graph above shows a bit more clearly the progression of a round of Counter-strike. The time up to the first box contains what are most likely messages sending what weapons the player's teammate bought and their movement to their initial positions. After that, each box represents one phase of the round, ending with the end of a firefight. Most rounds consist of multiple battles happening at the same time in different places, followed by the players, reloading, regrouping, and moving onward until they get into another firefight. Each box represents one of those phases.



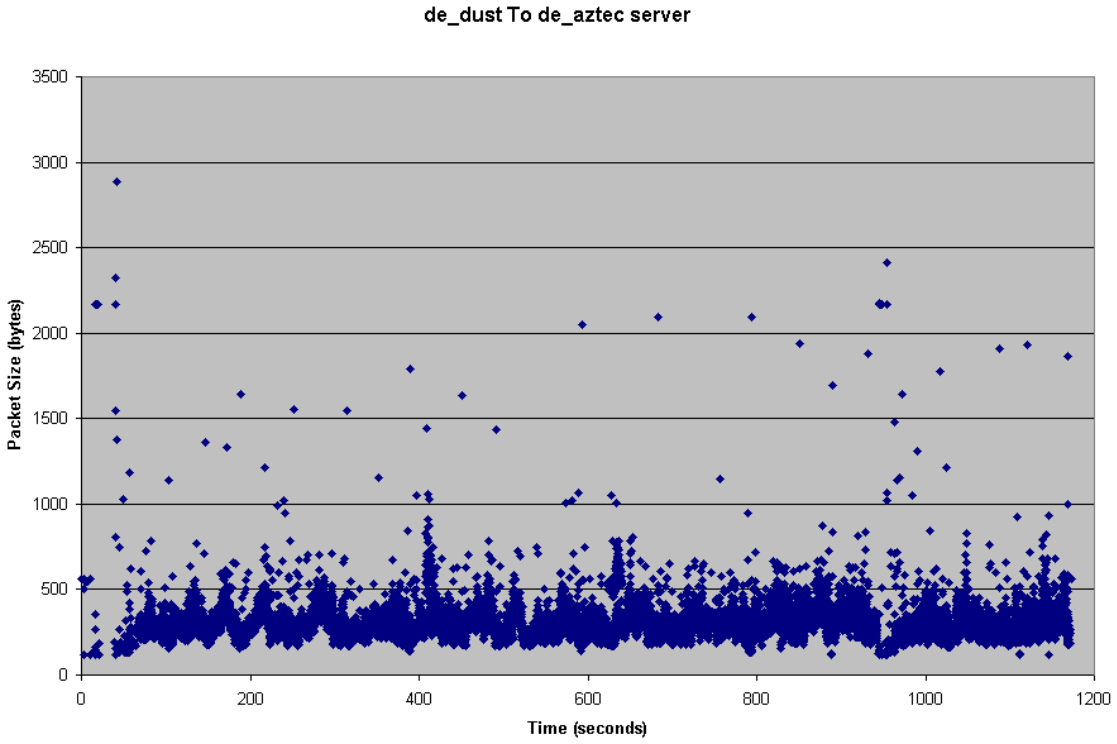
The image above corresponds with a round change and then a map change shortly afterward. The green circle again corresponds with a round change and you can see the rapid climb of the traffic shortly afterwards. This graph also shows the general downward trend of the packet size just before a round change. From 1350s to 1390s the packet size is consistently going downward. There was a brief firefight resulting in the player's death. Within 15 seconds, a map change was initiated. The map change is indicated by the blue box. It took several seconds for a round to begin after the map change, so the first firefight, the yellow oval, is significantly delayed compared to most rounds. It should also be noted that the peak of activity is smaller than the round before. Most players do not reconnect to the server in time to get into the first round.



Average bandwidth: 5871.07 bytes/s
Standard deviation: 3553.95 bytes/s

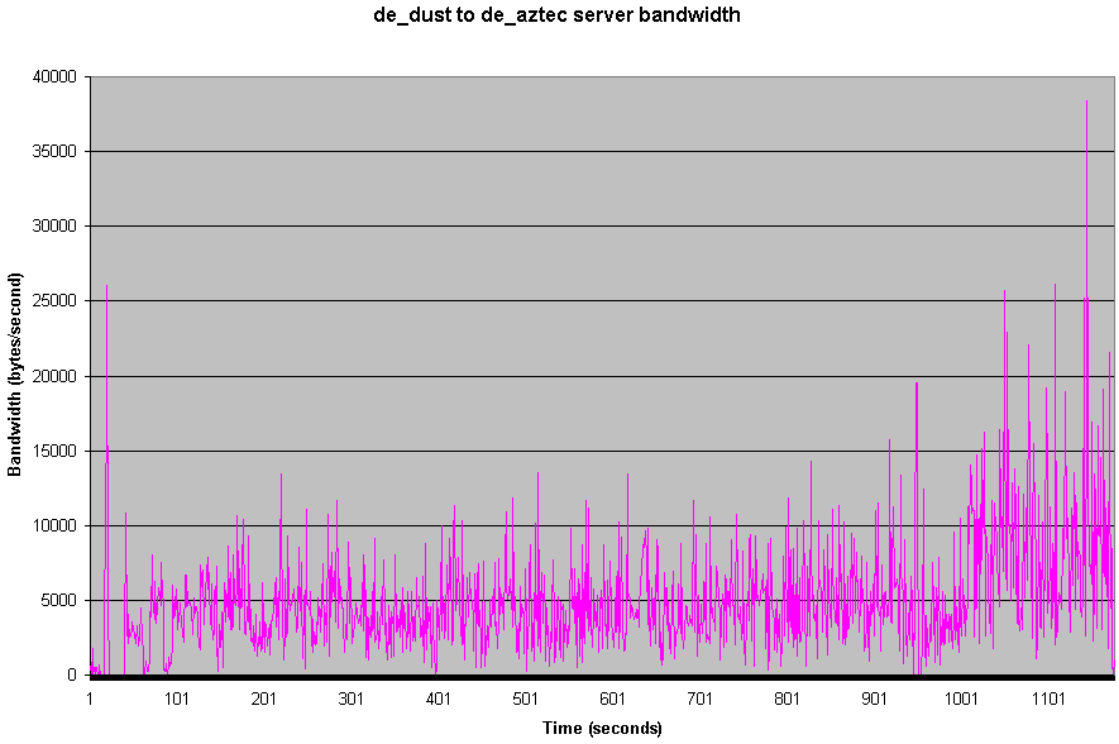
The bandwidth usage by the server still shows the cyclical nature of Counter-strike's network usage. The bandwidth peaks, with the exception of the extreme outliers that are more than likely firefights, trend downward for a few minutes, then the jump back upward at a round change. The large dip at 1100s is difficult to explain, since there was nothing in the game play to indicate any sort of dramatic prolonged bandwidth usage drop but it may be the result of network congestion.

The round cycle becomes less clear as the number of players on the server decreases. The play session below once again covered a map change but in this particular session, the number of players varied between 7 and 11. Another key difference is the first map. Dust tends to run quickly compared to other maps and rounds average around 2 minutes. However, we can still see the end of rounds by observing the large packets sent out whenever a round ends.



Total Packets: 18715
Mean Size: 309.95
Size Standard Deviation: 155.65
Largest Packet: 2886
Smallest Packet: 122
Mean Time Between Packets: 0.063
Total time: 1173.48

It should also be noted that these packets vary in size correlating exactly with the number of players in the game. The round packets just under 1500 bytes correlate with 7 players in the game, while those just above 1500 correlate with 8 and so on. Another notable feature is the reduced number of large firefights but an increase in their intensity. With smaller numbers of players, the teams tend to stick together and firefights usually involve all the players in the game at the time. The final feature is the map change to Aztec occurring at 950s. Note that this map change looks quite different than the one shown in the previous play session. The lack of the round change immediately preceding the map change makes it much easier to see the steep drop in packets sent by the server.



Average bandwidth: 4950.51 bytes/s
Standard deviation: 3695.54 bytes/s

The key feature on this bandwidth graph is the very clear “buy-time” intervals. After a new round, the players are unable to move or fire but are allowed to buy equipment. There are distinctly noticeable dips in bandwidth usage strongly correlating with the new round packets that would indicate that the bandwidth usage is significantly less during buy-time.

4.3 – Comparing Starcraft Traffic With Counter-strike Traffic

The network traffic generated by Starcraft and Counter-strike look very different. There is a significant degree of randomness in the Counter-strike traces, with no two games looking the same. This is in stark contrast to Starcraft where games are barely distinguishable. The differences between the two games are significant in terms of packet size and bandwidth consumption.

One such area of difference is the model the two games use to transmit a larger than normal amount of data. Starcraft packet sizes are close to uniform across the

number of players, with more packets transmitted when the bandwidth requirements increase. The Counter-strike client also follows this model, though the packet sizes are more variable than those in Starcraft. The Counter-strike server, on the other hand, increases the size of the packets when confronted with a need to send more data to the client. These differences are not visible when viewing a bandwidth graph but are important to note due to their effects on congestion.

The games are also different in the bandwidth they consume over time. Starcraft's bandwidth consumption varies very little over the course of a game regardless of events occurring within the game. Counter-strike, however, has a distinct cyclic pattern in its bandwidth distributions, which vary over time and have a marked correlation to game events. Overall, the amount of bandwidth consumed by a Starcraft player is comparable to bandwidth consumed by a Counter-strike client. A 6-player game of Starcraft has the local player sending between 3000 and 3500 bytes/second, and a Counter-strike client connected to a mostly-full server typically sends a little over 3200 bytes/second.

5 – Game Traffic Simulation

With the analysis completed, we began to develop algorithms for generating simulated traces that would mimic traces generated by actual games. We developed several algorithms, but it became obvious that a relatively simple solution could generate a generally accurate simulation, and that by fine-tuning this process we could develop a generic, sample trace generator that could be built into NS with few problems.

5.1 – NS Integration

In analyzing our data, we realized that the traffic patterns generated by games of Starcraft generally consisted of packets of distinct sizes delivered at relatively short intervals, but always steadily. From these observations, we hypothesized that a typical Starcraft session could be simulated by selecting weighted values for packet size and time since the last packet was sent, and running a simulation based on these probabilistically selected numbers.

The basis of our algorithm was that the NS timing for an application involves sending a packet of a specified size, waiting for a specified interval, and repeating the process. We used our tool to generate, for a single source IP, two probability buckets. The first contained the sizes of every packet in a given trace, and each size's corresponding frequency of appearance in the trace (essentially, the number of packets for each size). The other bucket contained the time between packets and the number of packets associated with each time interval.

Building our probabilistic simulator into this structure was relatively simple. We created an application, called game-app, that upon invocation reads bucket files generated by our tool (see section 3.3), picks a packet size from the size bucket, sends it, waits for a time chosen from the time bucket, and repeats. The functions that generate any given packet's size and time delta use an algorithm that uses a random number generator in tandem with our probabilistically weighted buckets.

Using this system, it is possible to probabilistically generate any trace, although it currently only supports the UDP protocol. We experienced some difficulty in dealing with the coarse time granularity introduced by the nature of operating systems (50ms in Windows 98) and by Commview. There were a large number of 0-value time deltas in

our initial time buckets that indicated a number of packets were sent in such rapid succession such that Commview or the operating system could not detect the time between packets. These bursts in traffic tended to create inaccuracies in our simulations, as they sometimes range higher than the node queues can handle. A simple, yet effective fix to this problem was to limit the number of packets that could be delivered in a single burst. This modification kept unusual bursts from distorting the simulation, and can be adjusted to suit any application that experiences similar effects.

Further research into Starcraft's behavior with differing numbers of players resulted in a specialized application for the game, which is derived from the probabilistic application. Typical traces were generated for 2, 4, 6, and 8 player games, and these were built into the Starcraft NS application. NS users can select the number of players by adjusting a variable (gameSize_); the default value is 6 players.

The Counter-strike application currently comes in two parts, a client application and a server application. The data sent by the client in Counter-strike does not vary based on the number of players in the game or the status (alive or dead) of the player. The data is remarkably uniform over time with respect to size, but varies slightly in the number of packets sent per second. This data pattern was a good match for the use of the bucket algorithm developed for game-app, so the cstrike-app class was derived from game-app and only overrides its parent class to load the buckets.

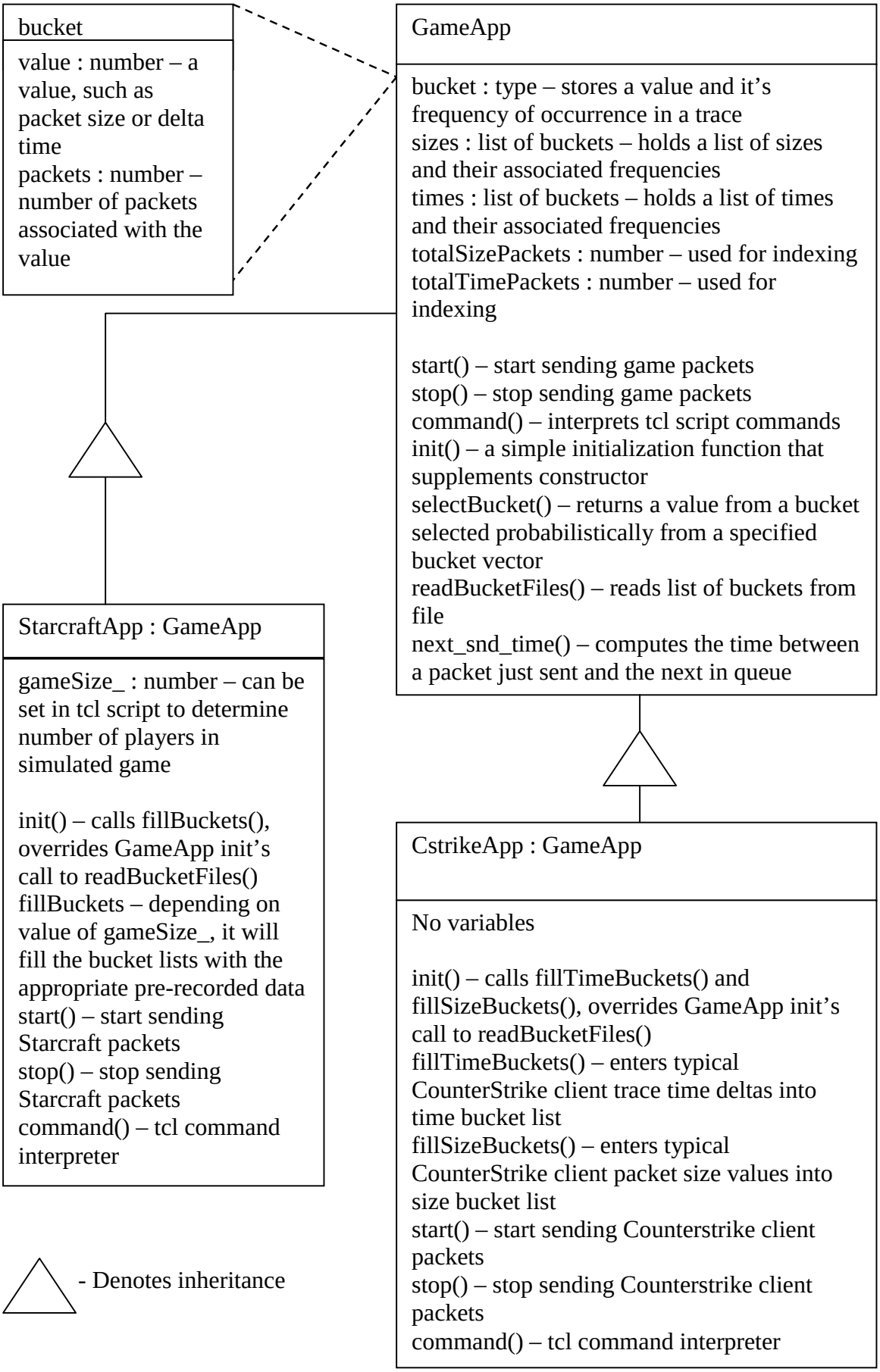
The server application, however, is quite different. Due to the cyclical nature of the packet size the server sends and the bursts of packets that are sent during firefights, we decided that the probabilistic model would not work well. Our analysis showed that within each round there were several segments, each ending with a firefight and a drop in the overall packet size. At the end of each round, the packet sizes would climb again and the process would repeat itself. With these factors in mind, we developed a model with several variables, each able to be tuned based on the number of players in the game and the average round length.

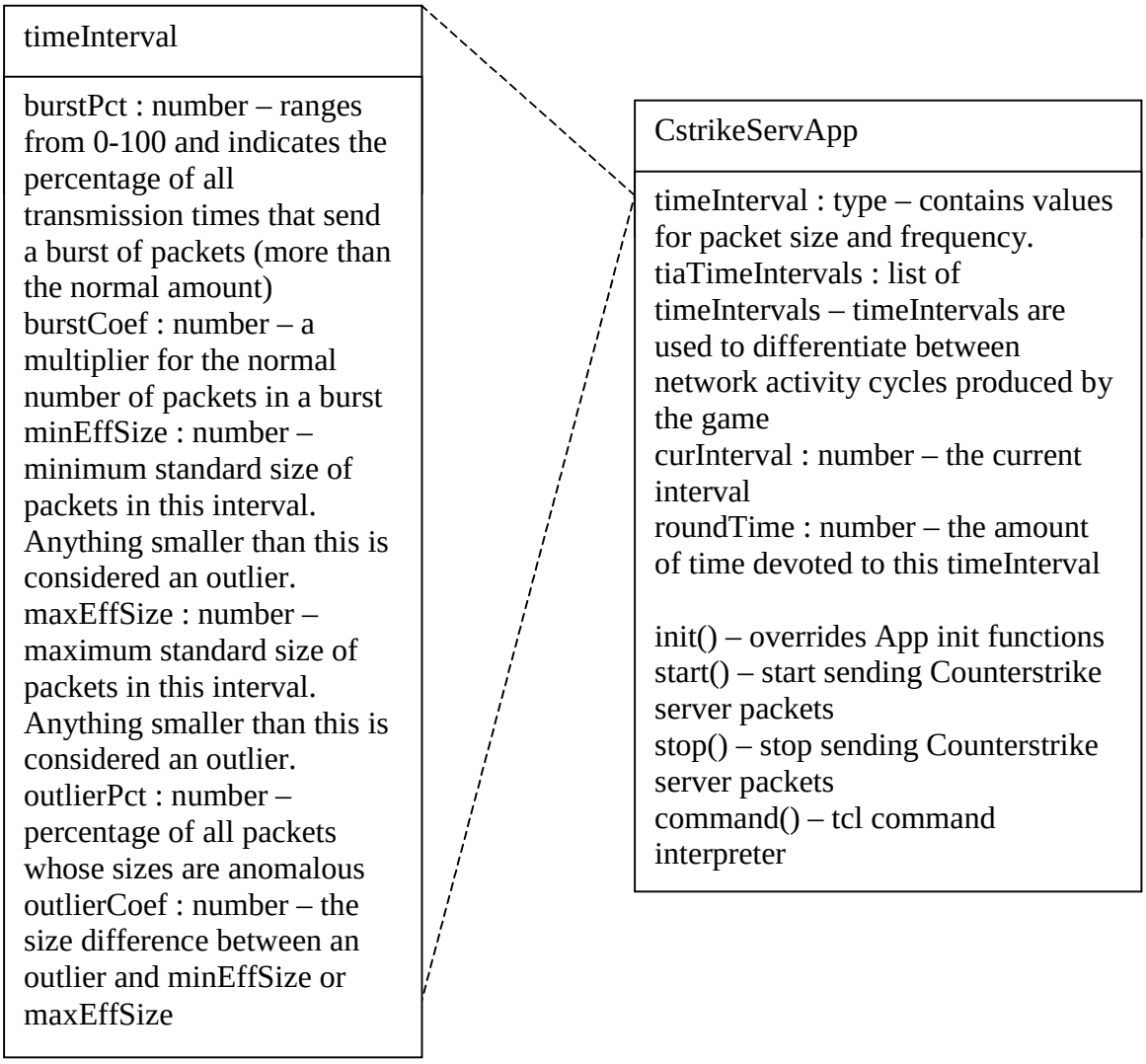
Each segment of a round has its own effective maximum and minimum packet size, as well as a burstiness and outlier percentage. As the round goes on, the packet size drops as well as the number of packets per burst and the frequency of bursts. The number of outliers seemed to correlate more strongly with the number of players in the game and

the map being played. A rudimentary version of this is included in our ns sample, however, due to time constrains, the transition between segments and the packet timer were not able to be fully implemented.

5.1.1 – Class Diagrams

The aim of the class hierarchy implemented in our simulators is to create at least one unique class for every game simulated. This provides a high level of customizing ability and will allow games to be compared to others within and outside of their genres. We had considered grouping games by type, but found that this would limit the games to a specific traffic pattern that does not necessarily define every game of its type.


bucket
value : number – a value, such as packet size or delta time
packets : number – number of packets associated with the value
GameApp
bucket : type – stores a value and it's frequency of occurrence in a trace
sizes : list of buckets – holds a list of sizes and their associated frequencies
times : list of buckets – holds a list of times and their associated frequencies
totalSizePackets : number – used for indexing
totalTimePackets : number – used for indexing
start() – start sending game packets
stop() – stop sending game packets
command() – interprets tcl script commands
init() – a simple initialization function that supplements constructor
selectBucket() – returns a value from a bucket selected probabilistically from a specified bucket vector
readBucketFiles() – reads list of buckets from file
next_snd_time() – computes the time between a packet just sent and the next in queue
StarcraftApp : GameApp
gameSize_ : number – can be set in tcl script to determine number of players in simulated game
init() – calls fillBuckets(), overrides GameApp init's call to readBucketFiles()
fillBuckets – depending on value of gameSize_, it will fill the bucket lists with the appropriate pre-recorded data
start() – start sending Starcraft packets
stop() – stop sending Starcraft packets
command() – tcl command interpreter
CstrikeApp : GameApp
No variables
init() – calls fillTimeBuckets() and fillSizeBuckets(), overrides GameApp init's call to readBucketFiles()
fillTimeBuckets() – enters typical CounterStrike client trace time deltas into time bucket list
fillSizeBuckets() – enters typical CounterStrike client packet size values into size bucket list
start() – start sending Counterstrike client packets
stop() – stop sending Counterstrike client packets
command() – tcl command interpreter
- Denotes inheritance



NS has a virtual base class called App from which all application-level simulation modules are derived. GameApp is derived from this class, and all inherited functions from class App are listed in GameApp's diagram. selectBucket() and readBucketFiles() are the only functions not derived from App. GameApp provides the probabilistic functionality described earlier, and also serves as a template for game classes that can use this form of simulation.

StarcraftApp is a game that utilizes the probabilistic algorithm, and is therefore derived from GameApp. Since Starcraft sessions can specify a number of players, its class differs from GameApp in that it has a variable (gameSize_) for this, and its "bucket filler" functions read from a set of previously generated buckets that each correspond to a particular game size.

The Counter-strike client is represented by the CStrikeApp class, which is also derived from GameApp. As the data from any given client is generally uniform between game sessions, a typical trace was used to fill the buckets for this game.

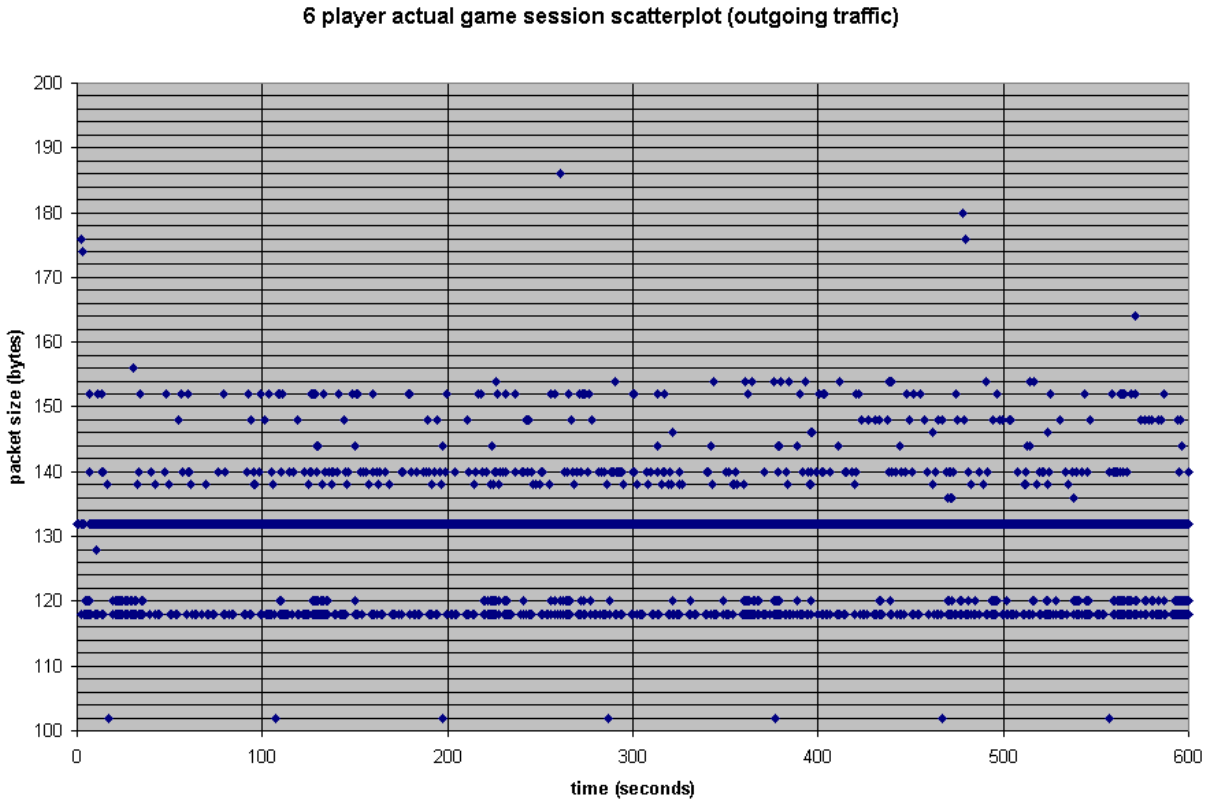
The Counter-strike server, however, applies a cyclic pattern to the data it sends. This class was not fully implemented due to time limitations, but its algorithm is as follows: Every distinct cycle of traffic patterns is rotated in a round-robin manner. Within each cycle, there are sets of values that determine its packet distribution. These values are explained in the class diagram. See section 4.2 – Counter-strike Traffic Data for details on how the cycles differ from each other.

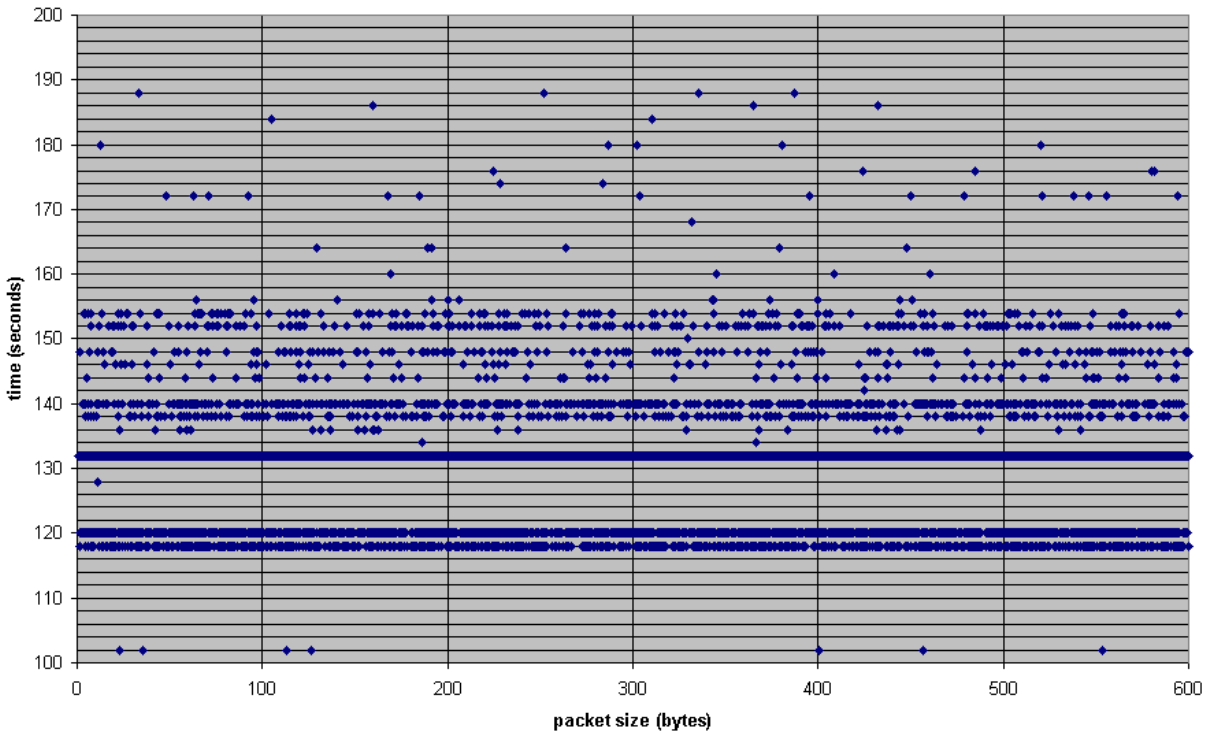
5.2 – Comparing Real Traffic to Simulated Traffic

In order to determine whether our NS implementation could portray an accurate representation of the data collected, we ran several simulations and compared them with actual trace data.

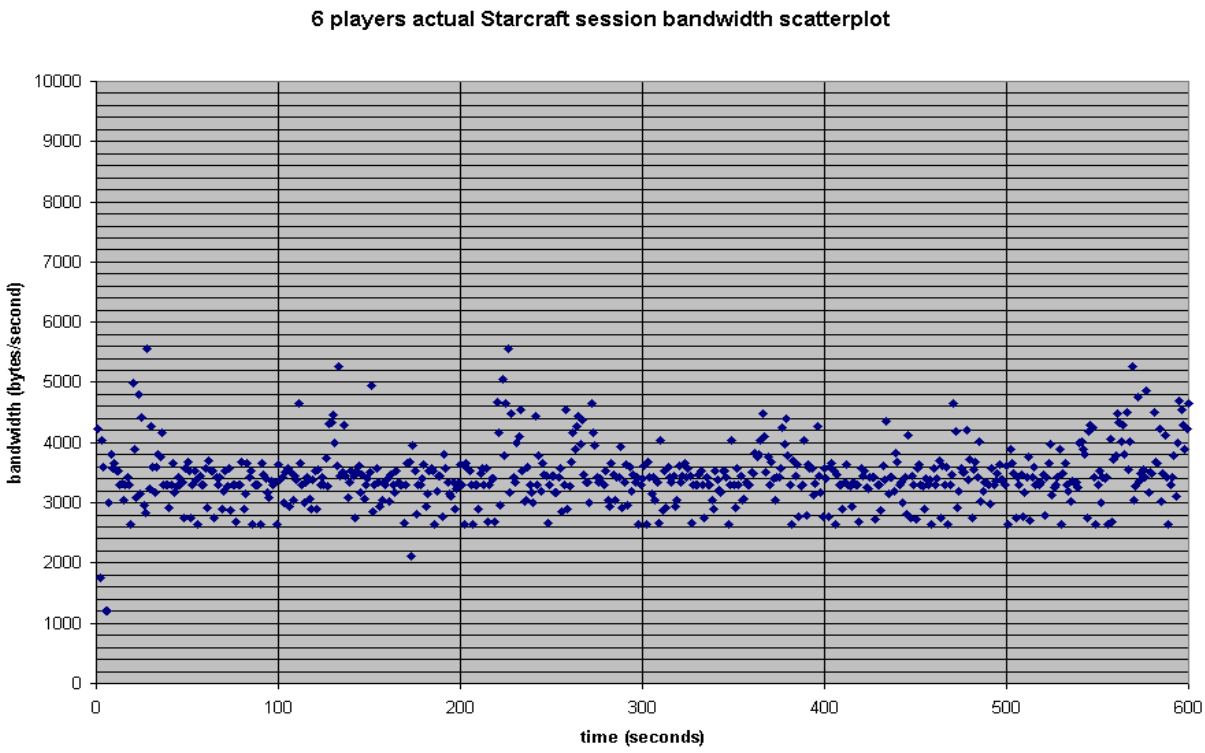
5.2.1 – Starcraft

Starcraft results turned out to be close to the actual trace data in terms of average bandwidth, but with an increased standard deviation in the bandwidth. The following graphs are scatter-plots of the data collected from a session of Starcraft and its corresponding simulation in NS.

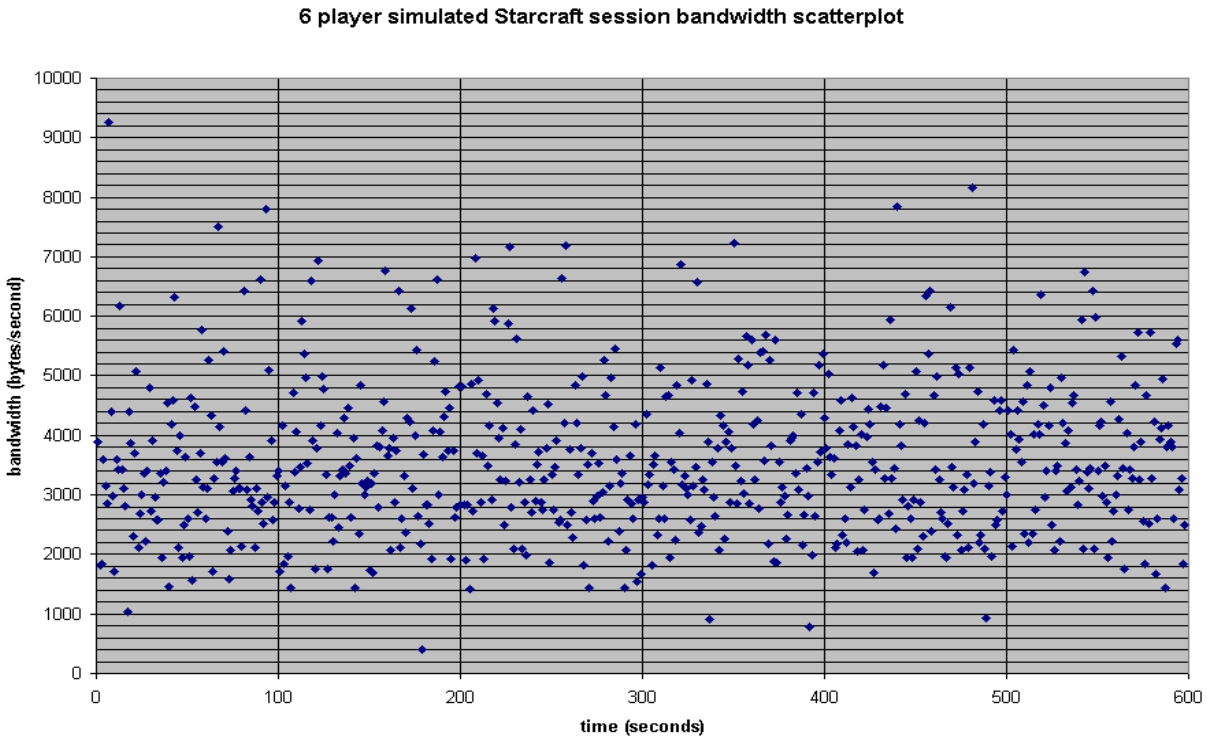

6 player actual game session scatterplot (outgoing traffic)
packet size (bytes)
200
190
180
170
160
150
140
130
120
110
100
0
100
200
300
400
500
600
time (seconds)


6 player simulated game session (outgoing traffic)
time (seconds)
200
190
180
170
160
150
140
130
120
110
100
0
100
200
300
400
500
600
packet size (bytes)

These graphs show that the simulated data produces more evenly distributed bands of 118 and 120 packets especially, and a more uniform distribution of packets larger than 152 bytes. The scale of these graphs, only 100-200 bytes, greatly accentuates differences in traffic. The simulated data looks much more dense than the real data, for instance, but they each send about the same amount of traffic. Bandwidth graphs can more clearly distinguish the difference between the actual and simulated data. The following graphs illustrate this:



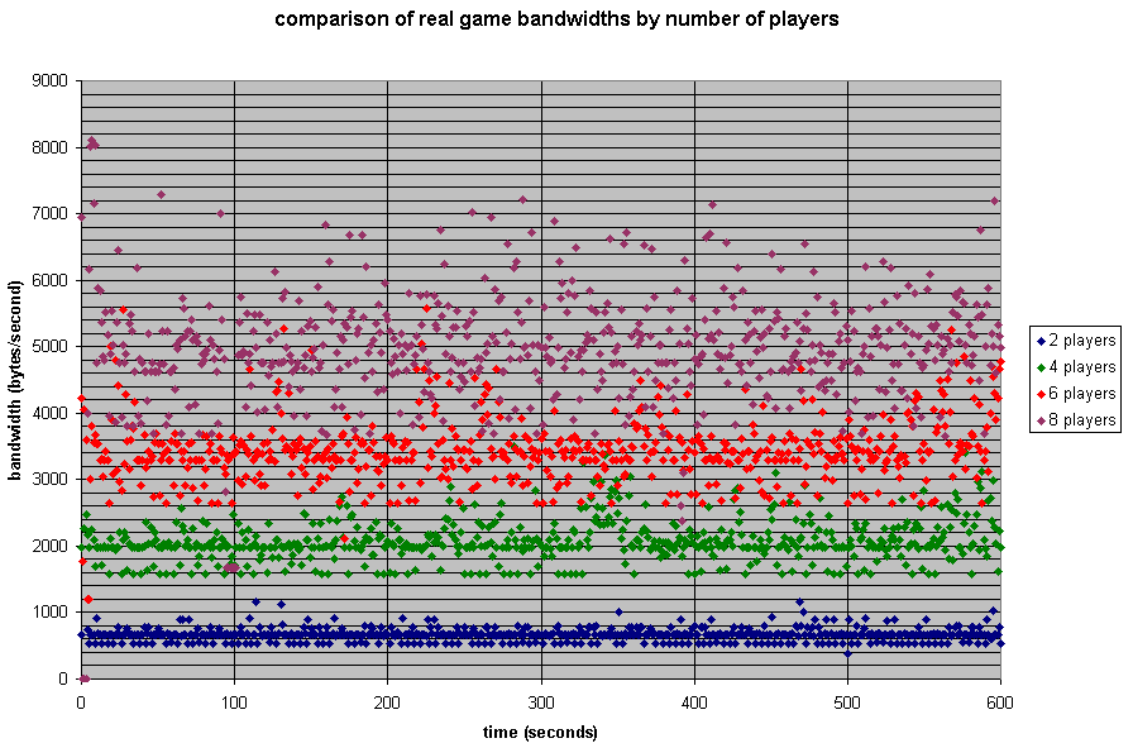
Average bandwidth: 3437.764
Standard deviation: 495.3148
Std dev/mean: 0.144



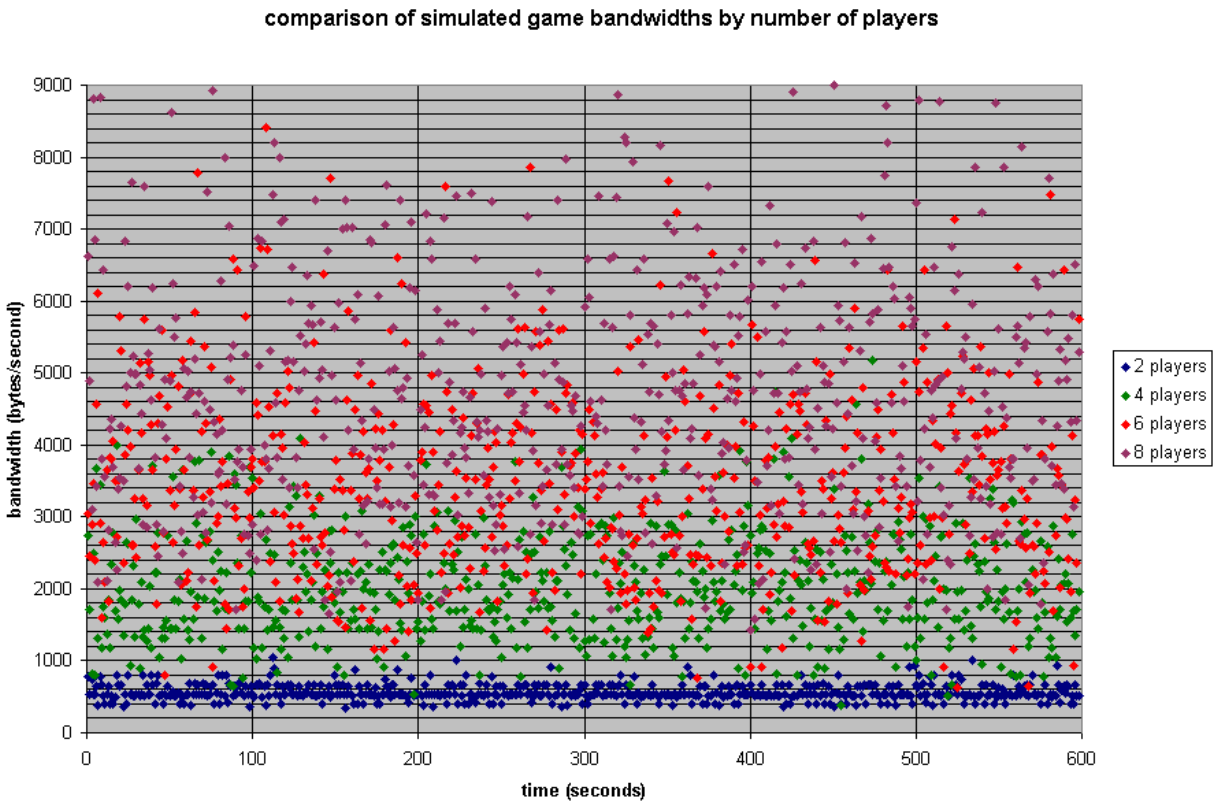
Average bandwidth: 3493.92 bytes/second
Standard deviation: 1373.403
Std dev/mean: 0.393

The variance of the simulated data is very large in comparison with that of the actual data. The simulation for this typical 6 player game shows a 36% increase in standard deviation over the real trace, even though the average bandwidth for each is less than 60 bytes/second from the other. We believe this is due to the large number of 0 time deltas in the buckets used for this simulation. These 0 values create a level of burstiness unlike real traces, and it would likely be beneficial to find a way around this problem in future work. As the amount of data sent in every burst is better curbed, the simulated data should move toward a real trace's packet transmission time variance. It seems the next important step in honing this algorithm is to find a way of achieving this.

We found that these results are typical of every 6-player game we had recorded, and this session can therefore represent the rest, as well. We were interested at this point in determining whether these results would be similar between games of varying size.



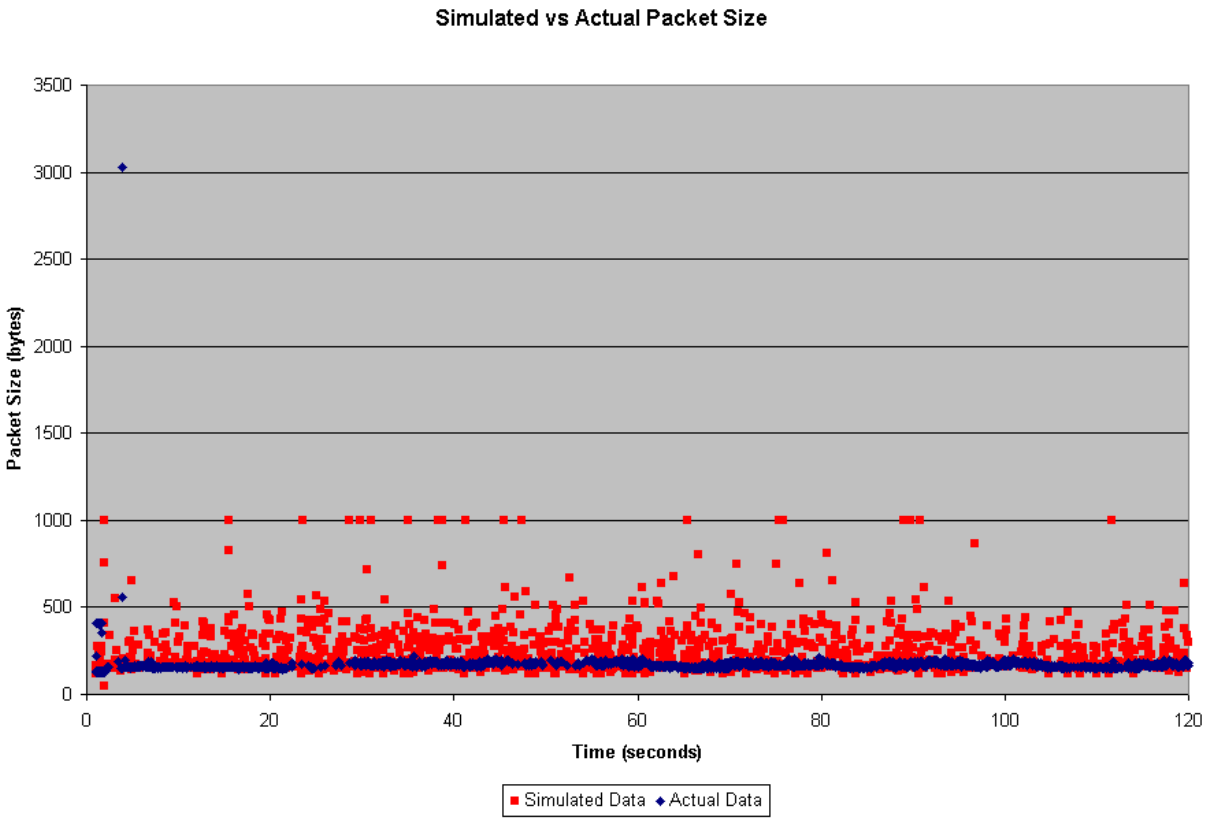
	2 players	4 players	6 players	8 players
Average bandwidth (bytes/second)	662.2995	2076.483	3437.764	4953.268
Standard deviation	106.4026	323.7356	495.3148	896.1799
Std dev/mean	0.160656	0.155906	0.144081	0.180927



	2 players	4 players	6 players	8 players
Average bandwidth	559.9632	2042.107	3486.258	4760.615
Standard deviation	110.6543	753.7073	1341.138	1699.715
Std dev/mean	0.19761	0.369083	0.384693	0.357037

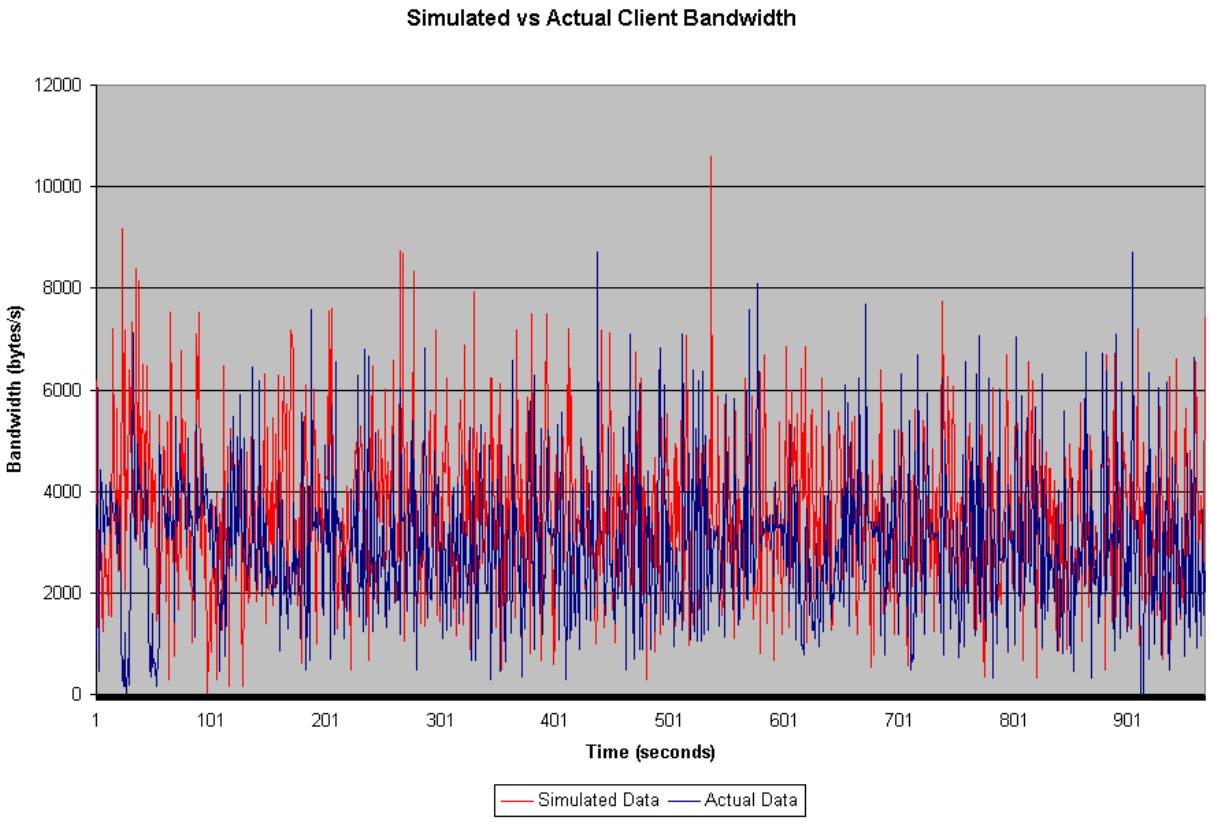
The amount of variance in the data was shown to increase with the number of players in section 4.1.3, but the simulated data behaves slightly differently. The amount of variance takes an immediate climb to a constant level between the 4, 6, and 8-player games, but remains very close to the actual values in the 2-player game. This shows that the simulations tend to become very inaccurate in terms of bandwidth distribution once the highest and lowest bandwidths in the real data become large enough to allow these levels of variance to occur.

5.2.2 – Counter-strike Client



Actual Data Average Packet Size:	170.13 bytes
Actual Data Standard Deviation:	65.52 bytes
Simulated Data Average Packet Size:	235.72 bytes
Simulated Data Standard Deviation:	147.93 bytes

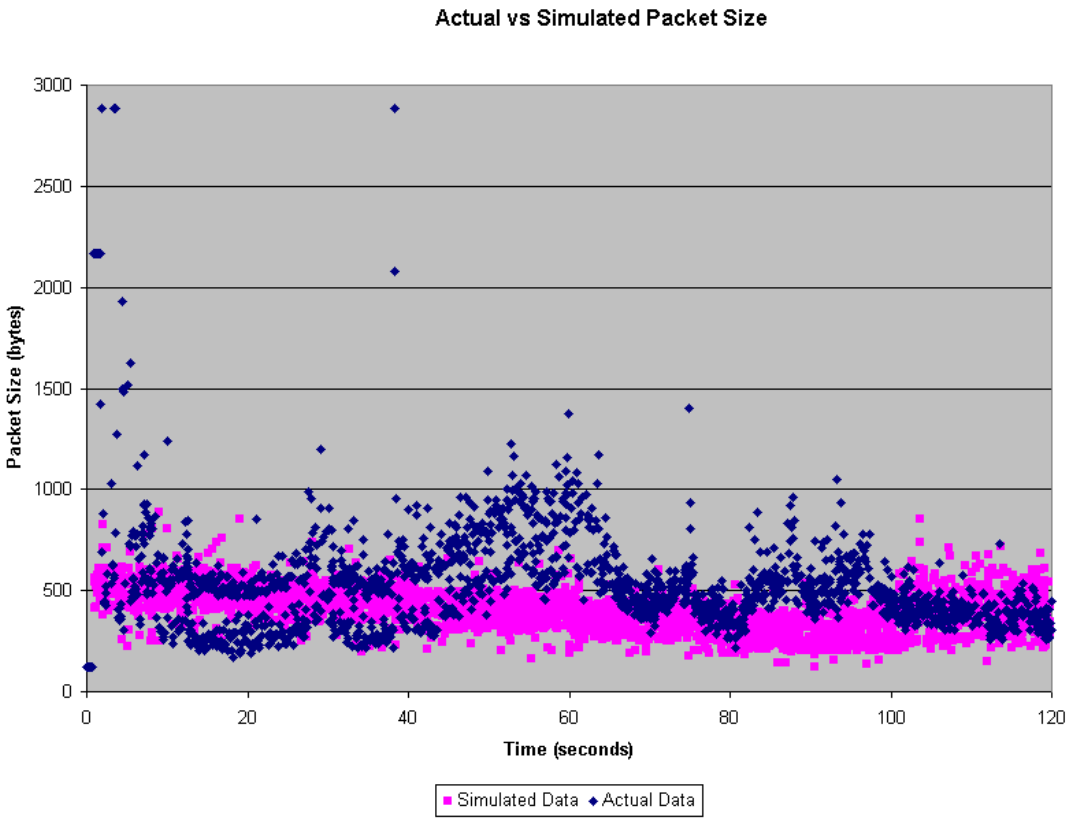
The Counter-strike client application was our first hard-coded NS app. It suffers from a number of flaws because the process for creating applications had not yet been fully developed at the time we made it. Obviously, the simulated data is a poor fit by packet size. The variance on the packets is far too large and the average size is almost 40% larger than it should be. These problems were found to be caused by some limitations of the JVM and a poor selection of a trace file for the template of this simulation. Our methodology for developing the application was flawed initially, and we did not perform verification testing early enough in the process to discover this error.



Actual Data Average Bandwidth:	2954.58
Actual Data Standard Deviation:	1450.12
Simulated Data Average Bandwidth:	3590.05
Simulated Data Standard Deviation:	1606.60

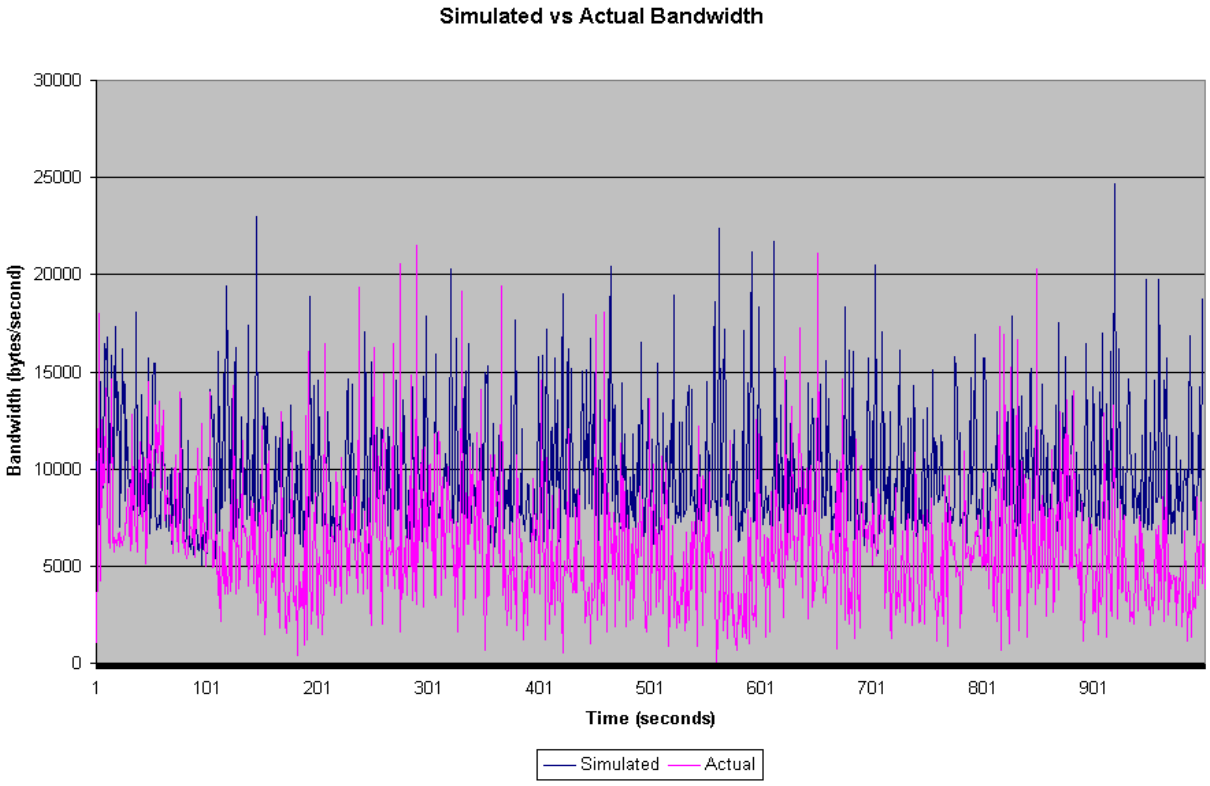
Due to the problems with packet size, the bandwidth usage is also significantly incorrect. However, the errors above should lead to a 30-40% increase in the bandwidth used rather than the 20% that is demonstrated here. This would indicate a significant error in the time component of the simulation. The Counter-strike client app is not totally beyond usefulness. A rebuild of the simulation seed data would make it fit much closer to the actual data.

5.2.3 – Counter-strike Server



Actual Data Average Packet Size:	528.25 bytes
Actual Data Standard Deviation:	256.61 bytes
Simulated Data Average Packet Size:	402.77 bytes
Simulated Data Standard Deviation:	102.06 bytes

The server was not yet completed at the end of our project, but we are including it for the sake of completeness. Several problems with the seed variables of the model are apparent looking at the data above. The initial timeInterval data values were a guess, and turned out to be well off from the average size. The default times between intervals also proved to be significantly off as is shown above. With some tweaking to the initial conditions and some work on the timing aspect, the server application could work well.



Actual Data Average Bandwidth:	5988.08
Actual Data Standard Deviation:	3285.25
Simulated Data Average Bandwidth:	9705.17
Simulated Data Standard Deviation:	3166.13

This is shown quite well with the above graph. Note that the simulated standard deviation is quite close the actual standard deviation, despite the large difference in average bandwidth. The simulation does a good job of accurately simulating the size of the bandwidth bursts the server creates as well as the lulls in data transmission. Once the packet size problem is corrected, the server would provide an accurate simulation of bandwidth consumption.

Despite the inaccuracies of the simulations, these games can still be simulated accurately. NS has a built-in trace mode that takes a file with sizes and times and sends them out in the order specified in the file. This allows a user to take the analysis presented above, generate a trace using that data, and get a more accurate simulation. We have also included several trace files with our MQP to facilitate this usage.

6 – Conclusions

The amount of research devoted to network games is lacking, even as the popularity of network games continues to grow. We intended to fill some of this knowledge gap with a study on how games behave over the Internet, and to provide a means of facilitating future research.

Our main tasks in this undertaking were to provide some meaningful analysis of network traffic generated by multiplayer games, and to simulate this traffic using a network simulator. While we were successful in meeting the former objective, there is still some amount of progress to be made on the latter.

In studying the behavior of multiplayer games over the Internet, we were able to construct simulations of two games with the intent that they would be solid representations of the games' traffic patterns. Our methodology proved true to our initial goals, as these games are simulated in NS, and the structure of our code is amicable to extension. However, the modules for these games require some amount of modification before they can be considered accurate representations of real traces.

There are a number of lessons that were learned in implementing our goals. It seems the key to understanding the traffic patterns of a game is to analyze enough data to be able to distinguish any factors that may affect the way they behave. Controlling as many factors as possible yields positive results, so it is important to compare game sessions played under similar conditions, as well as those that vary in aspects such as size, playing style, or network locality of players.

In addition, it is very important to build simulations early and analyze them thoroughly so as to leave time for tweaking it to meet specifications. More time is spent on configuring a game simulator, in our experience, than on building it.

Most importantly, it seems to be irrefutable that in order to simulate the traffic a game generates, one must first perform rigorous analysis on the data to be simulated. As there are a number of factors that can vary the behavior of any given game between sessions, they must be compared and contrasted before any simulation attempts can be trusted for accuracy.

As the level of attention devoted to the network aspect of games increases, we believe the need for simulating these games will grow as well. As games and other real-

time applications were not largely considered in the design of the Internet, they have had to adapt to the IP protocol. Most use a custom-built protocol on top of UDP in order to find a balance between advantages and disadvantages of TCP and UDP, but in the future, this may not always be necessary. It is possible that the Internet will begin to conform to games; that is, the development of protocols that better serve the behavior of games is likely to become a strategy for developers faced with increasingly network-intensive games. Simulation benefits this kind of research, as well. NS is already in use by research groups working on a variety of experimental protocols and routing schemes.

7 – Future Work

Despite the number of tasks we accomplished, there are always a number of opportunities for improvement. Choices were made throughout the project that required discarding options. We also planned to add more features than we had time for, so the addition of those items would be welcome. Finally, there was a rather large set of topics that while interesting and useful, were outside the scope of our particular project.

7.1 – Refinements

There were two tasks we were still attempting to complete as our project ended. While they were completed to some degree, we wanted to work a bit more on them so they would be as polished as the rest of the project. First, we could only perform limited analysis on game-app and its related classes in NS. Our first attempt at game-app gave simulations within 10% of expected mean, and with further refinements we were able to make it perform within 5% of expected mean. However, game-app creates simulated traffic with bandwidth variance that is much higher than in the real traces, most likely due to some problems with timing. Also, we did not have time to do rigorous testing with various bucket distributions. Some further work in this area should be performed before serious usage beyond the scope of our tests using game-app is done.

Second, we were able to design a basic structure for a Counter-strike server simulator and partially implement it. However, the functionality to determine when to send the next group of packets has not yet been fully implemented and little testing has been performed. Our early analysis on packet size distribution and burst size was promising, and with a little work, this could become fully functional.

7.2 – Additions

There were several areas that while within the scope of the project, we lacked the time or resources to pursue. For example, working on more games would take a small amount of time while at the same time being very worthwhile. By the end of the project, we had refined the analysis process of a game down to a few hours for some basic metrics and a few days for details. With most games, developing an NS application for

them would be limited to bucket generation and testing. With this refined process, it would be possible to compare and contrast several games within several genres with each other within a week's time and generate definitive simulations by genre rather than by game.

Writing and simulating TCP games is another interesting area of study. We made the decision early on to focus exclusively on UDP games due to the dramatic differences between TCP and UDP games. While the marketplace has overwhelmingly chosen UDP for its games, TCP might turn out to have promise if some modifications are made that reduce the severe problems a drop packet creates for games. Similarly, writing an NS app that handles TCP games would be a valuable addition to our work.

Finally, while the tool we wrote can produce a great deal of different useful types of output, it only accepts the output files of one packet sniffer, Commview, to generate this data. Adding modules for other popular sniffers would greatly increase the potential user base for the tool.

7.3 – Related Areas of Study

When we first started this project, we had two main choices for topics. We could either do in depth analysis on games as a whole, or do a brief overview of the games and write a NS addition to simulate them. However, this area still needs research and with some of our tools, this process could become much easier for future groups. A detailed traffic analysis of one or two games identifying what each packet does and the exact effects of packet loss and out of order packets could enable even better NS simulations to be written as well as demonstrating how such network events affect games.

Another promising path uses our NS additions to determine how a large number of game players on a network would affect congestion, router queuing, and packet loss. Research into this area might prove very valuable for both industry and academia. A detailed review in this area could change the way the games operate over networks and improve performance for game players and non-game players alike.

References

[Arm 01] Armitage, Grenville. "Lag Over 150 Milliseconds is Unacceptable." http://members.home.net/garmitage/things/quake3-latency-051701.html, May 2001.

Short treatise on user tolerance for game latency in id Software's Quake III. Conclusions were drawn based on noticed user responses to increasing levels of perceived latency and how latency factored into game selection.

[Ber 01] Bernier, Yahn W. "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization." Game Developers Conference http://www.gdconf.com/archives/proceedings/2001/bernier.doc, February 2001.

Very well written paper that describes the advantages, disadvantages, and details behind typical network gaming models. Discusses client-side prediction and lag compensation as viable means of improving gameplay.

[BT 01] Bettner, Paul and Terrano, Mark. "1500 Archers on a 28.8: Network Programming in *Age of Empires* and Beyond." Gamasutra. http://www.gamasutra.com/features/20010322/terrano_02.htm, March 2001.

A portrayal of some issues involved in creating a multiplayer system using an already constructed game engine. This was a successful endeavor, and a good resource for developing multiplayer games.

[Lin 99] Lincroft, Peter. "The Internet Sucks: Or, What I Learned Coding *X-Wing vs. Tie Fighter*." Gamasutra. http://www.gamasutra.com/features/19990903/lincroft_01.htm, September 1999.

An interesting tale of specific problems encountered in attempting to adapt a seemingly latency-intolerant game to multiplayer capabilities.

[Ng 97] Ng, Yu-Shen. "Designing Fast-Action Games for the Internet." Gamasutra. http://www.gamasutra.com/features/19970905/ng_01.htm, September 1997.

Extensive exploration of user-perceived latency tolerance, reducing perceived latency using a variety of methods, and conserving bandwidth. Very good resource for academic study; attempts to cover network issues across all genres.

[Pas 01] Pascal, Luban. "The Right Decision at the Right Time: Selecting the Right Features for a New Game Project." Gamasutra. http://www.gamasutra.com/features/20010926/luban_01.htm, September 2001.

Discusses the separate agendas of a typical game design team, including their motivations, areas of concentration, end-product goals, and priorities.

Appendix A – Structure of a Commview Packet Log

Each line represents one packet. This is an example of one packet.

Original packet:
#2002F000C01EA00397600045A4187C3005004261D9F0800450000414887000080112
29782D7E4F782D7E4E76987697D002D04567E140000E52E008017957879E8BE1F7A
F91348B66C42391E221851432E10982C2C400F9D00#

Start delimiter: #
Direction(0 pass, 1 in, 2 out): 2
Padding: 00
Minute: 2F
Padding: 00
Hour: 0C
Millisecond: 01EA
Padding: 00
Second: 39
Unknown: 76

Ethernet:
Destination MAC address: 00045A4187C3
Sender MAC address: 005004261D9F
Other ethernet stuff: 0800

IP: 4500
Datagram Size: 0041
Identification (packet number?): 4887
Flags?: 0000
TTL: 80
Protocol:11
Header Checksum: 2297
Sender IP address: 82D7E4F7
Destination IP address: 82D7E4E7

UDP:
Source Port: 6987
Destination Port: 697D
Length (header?): 002D
Checksum: 0456

Payload:
7E140000E52E008017957879E8BE1F7AF91348B66C42391E221851432E10982C2C40
0F9D00
End delimiter: #

Appendix B – Network Simulator Code

game-app.h

```
#ifndef __GAME_APP__
#define __GAME_APP__
#include "timer-handler.h"
#include "packet.h"
#include "app.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <stdlib.h>

#define SIZE_FILE "size.bkt"
#define TIME_FILE "time.bkt"
#define MAX_CONSEC_PKT 3

struct pkt_data
{
int iSize;
double dTimeDelta;
};

struct bucket
{
double value;
int packets;
};


class GameApp;


// Sender uses this timer to
// schedule next app data packet transmission time
class SendTimer : public TimerHandler {
 public:
        SendTimer(GameApp* t) : TimerHandler(), t_(t) {}
        inline virtual void expire(Event*);
 protected:
        GameApp* t_;
};
```

```
// Game Application Class Definition
class GameApp : public Application {
 public:
        GameApp();
        void send_game_pkt();  // called by SendTimer:expire (Sender)
        void send_ack_pkt(); // called by AckTimer:expire (Receiver)
 protected:
        int command(int argc, const char*const* argv);
        void start();       // Start sending data packets (Sender)
        void stop();        // Stop sending data packets (Sender)

        int totalSizePackets;
        int totalTimePackets;
        double elapsedTime_;
        int totalSent_;
        vector<bucket> sizes;
        vector<bucket> times;
        vector<pkt_data> trace;

        bucket selectBucket(int numPack
```
ets,vector<bucket> bucketList);
```
int readBucketFiles(char* sizeFileName, char* timeFileName);
```

game-app.cc

```
//
// Authors:   Josh Winslow and Dave Lapointe
// File:      game-app.cc
// Written:   11/13/01
//

#include "random.h"
#include "game-app.h"
#include "sys/time.h"


// GameApp OTcl linkage class
static class GameAppClass : public TclClass {
 public:
  GameAppClass() : TclClass("Application/Game") {}
  TclObject* create(int, const char*const*) {
    return (new GameApp);
  }
} class_app_game;


// When the send timer expires, call send_game_pkt()
void SendTimer::expire(Event*)
{
//  cout<<"Expired timer"<<endl;
  t_->send_game_pkt();

}


// Constructor (also initialize instances of timers)
GameApp::GameApp() : running_(0), snd_timer_(this)
{
        totalSizePackets = 0;
        totalTimePackets = 0;
}


bucket GameApp::selectBucket(int numPackets,vector<bucket> bucketList)
{

      int ran, index;
        index = 0;
        ran = (int)(Random::uniform(0,numPackets)+.5); // help with truncating
```

```
        // run through the buckets subracting the number of
        // "hits" in that bucket from the randomly selected
        // number until the generated number hits 0 or less
    while(ran>0)
        {
            ran -= bucketList[index].packets;
            index++;
    }
        index--;

    if(index>0)                         // Return the bucket value
         return bucketList[index];      // from the bucket we
                                        // selected above
    return bucketList[0];
}//end selectBucket


// trace files are of the format an integer for the size, a single
// character delimiter, space, and a double time until the next packet
// should be sent.
// I.E.
// 132, 0.0
// or
// 1452| 0.3432432
int GameApp::readTraceFile(char *szFilename)
{
    int totalPacketsCheck=0;
    int size=0;
    double time=0;
    pkt_data* temp;
    char delimiter;         // so cin works properly
    ifstream traceFile(szFilename);

    // check that the file handle is valid
    if(traceFile == NULL )
    {
        cout << "Invalid file handle... trace file NOT read" << endl;
        return -1;
    }

    while(traceFile >> size >> delimiter >> time)
    {
        if(traceFile.eof())
            break;

        temp = new pkt_data;
```

```
            temp->iSize = size;
            temp->dTimeDelta = time;
            trace.push_back(*temp);
        }
        return 0;
}


// bucket files are of the format: double the value of the bucket,
// single character delimiter, space, and an integer for the number of
// times that value appeared.  I.E.
// 132, 36614
// or
// 0.3300000000000409, 19
int GameApp::readBucketFiles(char* sizeFileName, char* timeFileName)
{
        int numPackets=0;
        int totalPacketsCheck=0;
        double size=0;
        double time=0;
        int i=0;
        bucket* temp;
        char delimiter;             // so cin works properly
        ifstream sizesFile(sizeFileName);
        ifstream timesFile(timeFileName);

        // check that the file handles are valid
        if((sizesFile == NULL) || (timesFile == NULL))
        {
            cout << "Invalid file handles... bucket files NOT read" << endl;
            exit(-1);
        }

        while(sizesFile >> size >> delimiter >> numPackets)
        {
                if(sizesFile.eof())
                        break;

            temp = new bucket;
            temp->value = size;
            temp->packets = numPackets;
            sizes.push_back(*temp);
            totalSizePackets += numPackets;
            i++;
        }
```

```
	i=0;

	while(timesFile >> time >> delimiter >> numPackets)
	{
		if(timesFile.eof())
			break;

		temp = new bucket;
		temp->value = time;
		temp->packets = numPackets;
		times.push_back(*temp);
		totalTimePackets += numPackets;
		i++;
	}

	return 0;
}


// OTcl command interpreter
int GameApp::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
          tcl.resultf("no such agent %s", argv[2]);
          return(TCL_ERROR);
      }
    }

  //add filename loading here.

  }

  return (Application::command(argc, argv));
}


void GameApp::init()
{
  // seed rng
  timeval temp;
  gettimeofday(&temp, NULL);
```

```
  Random::seed(temp.tv_sec);

  elapsedTime_ = 0;
  totalSent_ = 0;
  readBucketFiles(SIZE_FILE,TIME_FILE);
}


void GameApp::start()
{
  init();
  running_ = 1;
  send_game_pkt();
}


void GameApp::stop()
{
  running_ = 0;
}


// Send application data packet
void GameApp::send_game_pkt()
{
  if (running_) {
    int count = 0;
    double next_time_;
    bucket size;
    do
    {
      size = selectBucket(totalSizePackets,sizes);
      agent_->sendmsg(size.value);
      totalSent_ += (int)size.value;
      count++;
    }  while(((next_time_= next_snd_time()) ==  0) && (count <=
MAX_CONSEC_PKT));
    snd_timer_.resched(next_time_);
  }//end if
}


// Schedule next data packet transmission time
double GameApp::next_snd_time()
{
  bucket time = selectBucket(totalTimePackets,times);
```

```
  elapsedTime_ += time.value;
  return(time.value);
}


// Receive message from underlying agent
// We don't do anything with it, but it is necessary to declare one
void GameApp::recv_msg(int nbytes, const char *msg = 0)
{
}
```

starcraft-app.h

```
#ifndef __STARCRAFT_APP__
#define __STARCRAFT_APP__

#include "game-app.h"

class StarcraftApp;

// Starcraft Application Class Definition
class StarcraftApp : public GameApp {
 public:
        StarcraftApp();

 protected:
        int command(int argc, const char*const* argv);
        void init();
        void fillBuckets();
        void start();
        void stop();
        int gameSize_;          // corresponds to number of players in game
};

#endif
```

starcraft-app.cc

```
// Author:    Dave LaPointe
// File:      starcraft-app.cc
// Written:   12/02/2001

// THINGS TO DO:
// 1. ADD NUMBER OF PLAYERS OPTION


#include "starcraft-app.h"
#include "random.h"
#include "sys/time.h"

// StarcraftApp OTcl linkage class
static class StarcraftAppClass : public TclClass {
 public:
  StarcraftAppClass() : TclClass("Application/Game/Starcraft") {}
  TclObject* create(int, const char*const*) {
    return (new StarcraftApp);
  }
} class_app_starcraft;

// Constructor (also initialize instances of timers)
StarcraftApp::StarcraftApp():GameApp()
{
        bind("gameSize_", &gameSize_);
}

// OTcl command interpreter
int StarcraftApp::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
           tcl.resultf("no such agent %s", argv[2]);
           return(TCL_ERROR);
      }
    }
  }

  return (Application::command(argc, argv));
}
```

```
void StarcraftApp::start()
{
  init();
  running_ = 1;
  send_game_pkt();
}


void StarcraftApp::stop()
{
  running_ = 0;
}


void StarcraftApp::init()
{
  // seed rng
  timeval temp;
  gettimeofday(&temp, NULL);
  Random::seed(temp.tv_sec);

  elapsedTime_ = 0;
  totalSent_ = 0;
  fillBuckets();
}


void StarcraftApp::fillBuckets()
{
        This function contains four very long lists of time deltas and packet sizes, and has
been cropped to save space.
}
```

cstrike-app.h

```
#ifndef __CSTRIKE_APP__
#define __CSTRIKE_APP__
#include "game-app.h"

class CStrikeApp;


// Game Application Class Definition
class CStrikeApp : public GameApp {
 public:
        CStrikeApp();
 protected:
        int command(int argc, const char*const* argv);
        void init();
        void fillTimeBuckets();
        void fillSizeBuckets();
        void start();
        void stop();
};

#endif
```

cstrike-app.cc

```
// Author:    Josh Winslow
// File:      cstrike-app.cc
// Written:   11/28/01
//

#include "cstrike-app.h"
#include "sys/time.h"
#include "random.h"

// CStrikeApp OTcl linkage class
static class CStrikeAppClass : public TclClass {
 public:
  CStrikeAppClass() : TclClass("Application/Game/CStrike") {}
  TclObject* create(int, const char*const*) {
    return (new CStrikeApp);
  }
} class_app_cstrike;

// Constructor (also initialize instances of timers)
CStrikeApp::CStrikeApp():GameApp()
{
}
// OTcl command interpreter
int CStrikeApp::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
           tcl.resultf("no such agent %s", argv[2]);
           return(TCL_ERROR);
      }
    }
   //add filename loading here.
  }
  //call superclass?
  return (Application::command(argc, argv));
}

void CStrikeApp::start()
{
  init();
```

```
  running_ = 1;
  send_game_pkt();
}

void CStrikeApp::stop()
{
  running_ = 0;
}

void CStrikeApp::init()
{
  // seed rng
  timeval temp;
  gettimeofday(&temp, NULL);
  Random::seed(temp.tv_sec);

  elapsedTime_ = 0;
  totalSent_ = 0;
  fillSizeBuckets();
  fillTimeBuckets();
}

void CStrikeApp::fillTimeBuckets() {
```

This function also contains a very large amount of packet information, and has been cropped to save space.

```
}
```

cstrikeserv-app.h

```
#ifndef __CSTRIKESERV_APP__
#define __CSTRIKESERV_APP__
#include "random.h"
#include "app.h"
#include "packet.h"
#include "timer-handler.h"
#include <iostream>

#define TOTAL_INTERVALS 5

class CStrikeServApp;

struct timeInterval {
  double burstPct;
  double burstCoef;
  int minEffSize;
  int maxEffSize;
  double outlierPct;
  double outlierCoef;
};

class CSSSendTimer : public TimerHandler {
 public:
       CSSSendTimer(CStrikeServApp* t) : TimerHandler(), t_(t) {}
       inline virtual void expire(Event*);
 protected:
       CStrikeServApp* t_;
};

// CStrikeServApp Application Class Definition
class CStrikeServApp : public Application {
 public:
          CStrikeServApp();
          void send_css_pkt();

 protected:
       int curInterval;
          int gameState_;
          int command(int argc, const char*const* argv);
          void init();
          void start();
          void stop();
       timeInterval *tiaTimeIntervals;
          int running_;         // If 1 application is running
```

```
        double roundTime;
        CSSSendTimer css_snd_timer_;  // SendTimer
};

#endif
```

cstrikeserv-app.cc

```
// Author:    Josh Winslow
// File:      cstrikeserv-app.cc
// Written:   11/28/01
//

#include "cstrikeserv-app.h"

// CStrikeServApp OTcl linkage class
static class CStrikeServAppClass : public TclClass {
 public:
  CStrikeServAppClass() : TclClass("Application/CStrikeServ") {}
  TclObject* create(int, const char*const*) {
    return (new CStrikeServApp);
  }
} class_app_cstrikeserv;

void CSSSendTimer::expire(Event*)
{
  t_->send_css_pkt();

}

// Constructor
CStrikeServApp::CStrikeServApp() : running_(0), css_snd_timer_(this)
{
}
// OTcl command interpreter
int CStrikeServApp::command(int argc, const char*const* argv)
{
  Tcl& tcl = Tcl::instance();

  if (argc == 3) {
    if (strcmp(argv[1], "attach-agent") == 0) {
      agent_ = (Agent*) TclObject::lookup(argv[2]);
      if (agent_ == 0) {
            tcl.resultf("no such agent %s", argv[2]);
            return(TCL_ERROR);
      }
    }
   //add filename loading here.
  }
  //call superclass?
  return (Application::command(argc, argv));
}
```

```
void CStrikeServApp::start()
{
  init();
  running_ = 1;
  gameState_ = 0;
  send_css_pkt();
}

void CStrikeServApp::stop()
{
  running_ = 0;
}

void CStrikeServApp::init()
{
  tiaTimeIntervals = new timeInterval[TOTAL_INTERVALS];
  tiaTimeIntervals[0].burstPct = 5;
  tiaTimeIntervals[0].burstCoef = 6.00;
  tiaTimeIntervals[0].minEffSize = 400;
  tiaTimeIntervals[0].maxEffSize = 610;
  tiaTimeIntervals[0].outlierPct = 15;
  tiaTimeIntervals[0].outlierCoef = .5;

  tiaTimeIntervals[1].burstPct = 5;
  tiaTimeIntervals[1].burstCoef = 6.00;
  tiaTimeIntervals[1].minEffSize = 350;
  tiaTimeIntervals[1].maxEffSize = 550;
  tiaTimeIntervals[1].outlierPct = 15;
  tiaTimeIntervals[1].outlierCoef = .5;

  tiaTimeIntervals[2].burstPct = 5;
  tiaTimeIntervals[2].burstCoef = 6.00;
  tiaTimeIntervals[2].minEffSize = 300;
  tiaTimeIntervals[2].maxEffSize = 500;
  tiaTimeIntervals[2].outlierPct = 15;
  tiaTimeIntervals[2].outlierCoef = .5;

  tiaTimeIntervals[3].burstPct = 5;
  tiaTimeIntervals[3].burstCoef = 6.00;
  tiaTimeIntervals[3].minEffSize = 250;
  tiaTimeIntervals[3].maxEffSize = 450;
  tiaTimeIntervals[3].outlierPct = 15;
  tiaTimeIntervals[3].outlierCoef = .5;

  tiaTimeIntervals[4].burstPct = 5;
```

```
  tiaTimeIntervals[4].burstCoef = 6.00;
  tiaTimeIntervals[4].minEffSize = 200;
  tiaTimeIntervals[4].maxEffSize = 400;
  tiaTimeIntervals[4].outlierPct = 15;
  tiaTimeIntervals[4].outlierCoef = .5;

  curInterval = 0;
  roundTime = 0;


}
void CStrikeServApp::send_css_pkt()
{

  int effRange = tiaTimeIntervals[curInterval].maxEffSize-
                    tiaTimeIntervals[curInterval].minEffSize;
  int numPackets,pktSize;
  double rand;
  if(running_) {
    numPackets = (int)(Random::uniform(2)+1.5);
    rand = Random::uniform(100);
    if(rand<tiaTimeIntervals[curInterval].burstPct) {
      numPackets = (int)(numPackets*tiaTimeIntervals[curInterval].burstCoef);
    }//end if
    for(int i=0;i<numPackets;i++) {
          pktSize = (int)(Random::uniform(effRange)+.5);
          pktSize += tiaTimeIntervals[curInterval].minEffSize;
          rand = Random::uniform(100);
          if(rand<tiaTimeIntervals[curInterval].outlierPct) {
            rand = Random::uniform(0,1);
            if(rand>.5)
                  pktSize = pktSize +=
(int)(pktSize*Random::uniform(tiaTimeIntervals[curInterval].outlierCoef));
            else
                  pktSize = pktSize -=
(int)(pktSize*Random::uniform(tiaTimeIntervals[curInterval].outlierCoef));

          }//if
          agent_->sendmsg(pktSize);
    }//end for
    double next_time_ = .1;
    roundTime += next_time_;
    if( roundTime/(curInterval+1)>20 ) {
          if(curInterval<TOTAL_INTERVALS-1 ) {
            curInterval++;
          } else {
```

```
            curInterval=0;
            cout<<"Reset"<<endl;
           }
    }
    css_snd_timer_.resched(next_time_);
  }//end if(running_)
}//end send_css_pkt
```

Appendix C – Useful Perl Scripts

packet_concatenator.pl

Author: Dave LaPointe

```
#!/usr/bin/perl -w

use strict;

my($currentfile, $filename, $filter);

$filter = "none";

print "This script will concatenate all .ccf files in this directory into one file.\n";
print "This process can produce a very large file!  You have been warned.\nContinue?
(y/n) > ";

## process command line args and input
if(<> eq "y\n")
{
	## get packet filtering options
	while(!(($filter eq "tcp") || ($filter eq "udp")))
	{
		print "Select a filtering option:\n1) Filter out all tcp packets\n2) Filter out
all udp packets\n3) No filtering\nOption: ";

		$filter = <>;

		if($filter eq "1\n")
		{
			print "TCP packets will be removed\n";
			$filter = "tcp";
		}

		elsif($filter eq "2\n")
		{
			print "UDP packets will be removed\n";
			$filter = "udp";
		}

		elsif($filter eq "3\n")
		{
			$filter = "none";
```

```
                }

                else
                {
                        die "Unrecognized option: ARGV[0]";
                }
        }
}
else
{
        die ("Program aborted by user");
}

## create the big file
open(OUTFILE, ">allpackets");

## open every Commview file in current directory (start a loop)
while($filename = glob("*\.ccf"))
{
        $currentfile = "";

        print "processing $filename\n";

        ## open each file
        open(CMFILE, $filename) or die ("Open failed");

        ## read it in
        until(eof CMFILE)
        {
                $currentfile .= <CMFILE>;
        }

        ## check filter options
        if($filter eq "tcp")
        {
                ## get rid of tcp packets (spots 67 and 68 in file}
                $currentfile =~ s|^.{66}06.*$||gm;
        }
        elsif($filter eq "udp")
        {
                ## get rid of udp packets (spots 67 and 68 in file}
                $currentfile =~ s|^.{66}11.*$||gm;
        }

        ## remove excess newlines
        $currentfile =~ s|\n\n+|\n|g;
```

```
        ## concatenate
        print OUTFILE $currentfile;
}

## final comments
print "Files combined into file named \"allpackets\". If you choose to add a .ccf
extension, you should\n";
print "avoid redundancy by removing this file from the current directory before running
this script again.\n";
```

codegen.pl

Author: Dave LaPointe

```
#!/local/usr/bin/perl -w

# This script generates code to stuff the contents of a bucket file into GameApp bucket
structs

use strict;

# define some vars
my($sizesfilename, $timesfilename, $sizesline, @sizes, @sizenums, $timesline, @times,
@timenums, $i, $value, $numsizes, $numtimes, $TIMES, $SIZES, $OUTPUT,
$totaltimepackets, $totalsizepackets);

$numsizes = 0;
$numtimes = 0;
$totaltimepackets = 0;
$totalsizepackets = 0;

# get the bucket info
open($TIMES, "<time.bkt") or die('time.bkt not found\n');
open($SIZES, "<size.bkt") or die('size.bkt not found\n');

# open some output files
open($OUTPUT, ">code.txt") or die('error opening/creating output file');

# get the stuff
for($i = 0; $timesline = <$TIMES>; $i++)
{
	$timesline =~ m/^(.*?), (.*?)$/;
	$times[$i] = $1;
	$timenums[$i] = $2;
	$totaltimepackets += $timenums[$i];
	$numtimes++;
}

for($i = 0; $sizesline = <$SIZES>; $i++)
{
	$sizesline =~ m/^(.*?), (.*?)$/;
	$sizes[$i] = $1;
	$sizenums[$i] = $2;
	$totalsizepackets += $sizenums[$i];
	$numsizes++;
```

```
}

# make code
# time deltas array
print $OUTPUT 'double temptimes[] = {';

for($i = 0; $i < $numtimes; $i++)
{
	if($i == ($numtimes-1))
	{
		print $OUTPUT "$times[$i]\}\;\n";
	}

	else
	{
		print $OUTPUT "$times[$i],";
	}
}

# time packet number array
print $OUTPUT 'int temptimenums[] = {';

for($i = 0; $i < $numtimes; $i++)
{
	if($i == ($numtimes-1))
	{
		print $OUTPUT "$timenums[$i]\}\;\n";
	}

	else
	{
		print $OUTPUT "$timenums[$i],";
	}
}

# size array
print $OUTPUT 'int tempsizes[] = {';

for($i = 0; $i < $numsizes; $i++)
{
	if($i == ($numsizes-1))
	{
		print $OUTPUT "$sizes[$i]\}\;\n";
	}

	else
```

```
	{
		print $OUTPUT "$sizes[$i],";
	}
}

# size packet number array
print $OUTPUT 'int tempsizenums[] = {';

for($i = 0; $i < $numsizes; $i++)
{
	if($i == ($numsizes-1))
	{
		print $OUTPUT "$sizenums[$i]\}\;\n";
	}

	else
	{
		print $OUTPUT "$sizenums[$i],";
	}
}

# total packets for each
print $OUTPUT "totalTimePackets = $totaltimepackets\;\ntotalSizePackets =
$totalsizepackets\;\n";
print $OUTPUT "int timeArraySize = $numtimes\;\nint sizeArraySize = $numsizes\;\n";

# the loop to set it all up
print $OUTPUT <<EOD
bucket* temp;

for(int i = 0; i < timeArraySize; i++)
{
	temp = new bucket;
	temp->value = temptimes[i];
	temp->packets = temptimenums[i];
	times.push_back(*temp);
}

for(int i = 0; i < sizeArraySize; i++)
{
	temp = new bucket;
	temp->value = tempsizes[i];
	temp->packets = tempsizenums[i];
	sizes.push_back(*temp);

}EOD
```

results.pl

Author: Dave LaPointe

```
#!/local/usr/bin/perl -w

# This script parses out.tr for 2 NODE TOPOLOGIES THAT USE UDP
# records the size and time of each packet
# records bandwidth for each second in a separate file

use strict;

# define some vars
my($tracefile, $outfile, $bwfile, $line, $totalbytes, $time);

$time = 1;  # apps usually start at 1.0s
$totalbytes = 0;

# open trace file and output files
open($tracefile, "<out.tr") or die("out.tr not found\n");
open($outfile, ">results.txt") or die ("could not open\/create results.txt\n");
open($bwfile, ">bandwidth.txt") or die ("could not open\/create bandwidth.txt\n");

# handle it all in one loop to conserve memory
while($line = <$tracefile>)
{
        if($line =~ m/^\+ (.*?) . . udp (.*?) -.*?$/)
        {
                print $outfile "$1, $2\n";

                if($1 < ($time + 1))
                {
                        $totalbytes += $2;
                }

                else
                {
                        print $bwfile "$time, $totalbytes\n";
                        $time++;
                        $totalbytes = 0;
                }
        }
}

# clean up
```

```
close($tracefile);
close($outfile);
```