2006

Addressing Cheating and Workload Characterization in Online Games

Christopher Chambers
Portland State University

DISSERTATION APPROVAL

The abstract and dissertation of Chris Chambers for the Doctor of Philosophy in Computer Science were presented November 6, 2006, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

Wu-chang Feng, Chair

Wu-chi Feng

Bryant York

Nirupama Bulusu

Steve Bleiler
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:

Cynthia Brown, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Chris Chambers for the Doctor of Philosophy in Computer Science presented November 6, 2006.

Title: Addressing Cheating and Workload Characterization in On-Line Games

The Internet has enabled the popular pastime of playing video games to grow rapidly by connecting game players in disparate locations. However, with popularity have come the two challenges of hosting a large number of users and detecting cheating among users. For reasons of control, security, and ease of development, the most popular system for hosting on-line games is the client server architecture. This is also the most expensive and least scalable architecture for the game publisher, which drives hosting costs upwards with the success of the game. In addition to the expense of hosting, as a particular game grows more competitive and popular, the incentive to cheat for that game grows as well. All popular on-line games suffer from cheats in one form or another, and this cheating adversely affects game popularity and growth.

In this dissertation we follow a hypothetical game company (GameCorp) as it surmounts challenges involved in running an on-line game. We develop a characterization of gamer habits and game workloads from data sampled over a period

of years, and show the benefits and drawbacks of multiplexing online applications together in a single large server farm. We develop and evaluate a geographic redirection service for the public server architecture to match clients with servers. We show how the public server game architecture can be used to scalably host large persistent games such as massively multiplayer (MMO) games that previously used the client server architecture. Finally we develop a taxonomy for client cheating in on-line games to focus research efforts, and specifically treat one of the categories in detail: information exposure in peer-to-peer games.

The thesis of this dissertation is: a methodology for accurate usage modeling of server resources can improve workload management; public-server resources can be leveraged in new ways to serve multiplayer on-line games; and that information exposure in peer-to-peer on-line games is preventable or detectable with the adoption of cryptographic protocols.

ADDRESSING CHEATING AND WORKLOAD CHARACTERIZATION IN ON-LINE GAMES

by

CHRIS CHAMBERS

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
2006

Acknowledgments

I would like to thank Prof. Wu-chang Feng and Prof. Wu-chi Feng for their invaluable guidance and support at every stage of my progress. Surely the level of advising I was given was above and beyond what was required to nurture a graduate student.

I am also very grateful for the insightful comments and feedback given to me by Prof. York, Prof. Bulusu and Prof. Bleiler in the creation of this document and during my academic career.

Many thanks to my fellow office-mates and cohorts Ed Kaiser, Jim Snow, Jie Huang and the rest of the SYN department for the collaborative efforts achieved during lunch conversations, on-line game exploration and on the foosball table.

Finally my thanks go especially to my wife Evelyn, without whose unlimited support and patience I wouldn't have lasted a year.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

On-line gaming is a popular pastime around the world. A current chart of the growth of massively multiplayer online (MMO) games demonstrates population growth in an exponential phase [84]. While such a growth rate is unlikely to continue for long, surveys suggest that video games represent a modern-day generation gap, similar to the "rock and roll" gap of the 1960's: while 50% of Americans play video games, 75% of the gamers are under 40 years old [85]. We believe online gaming will be an increasingly popular activity over the next few decades as it spreads to new areas of the world and popular media devices such as console platforms and cell phones [1]. Furthermore, we believe games are valuable not only as a popular venue for entertainment, but also because of the contributions game developers can make towards computer science research. Efforts to make better computer games have driven advances in computer graphics visualization, networking and scientific computation [75, 19, 59].

While there is a lot of growth and potential capital at stake in the on-line gaming industry, there is also a great deal of risk. A decade ago it was not unheard-of for a game made by a few people in their spare time to become popular and successful. By contrast, today's on-line games have development budgets similar to Hollywood movies and the operating expenses of hosting a 24-hour a day service for potentially hundreds of thousands of users. The high development costs for a game do not guarantee a return; indeed, only a small percentage of published games thrive.

In this thesis, we take the point of view of a fictitious game publisher called GameCorp as it considers how to host a successful on-game while avoiding the huge development and maintenance costs and addressing the prevalence of cheating associated with modern on-line games.

1.1 Research Challenges

Two central issues associated with on-line games are hosting and cheating. In the hosting problem, the issue is one of scalability and popularity: how can a game architecture be built to support subscribers ranging across many orders of magnitude? The popularity of a game depends on influences outside a game architect's control, such as market forces. A hosting solution requires many tradeoffs in server resource allocation, game responsiveness in terms of user interactivity, and the amount of computation performed at the client. Adequately provisioning for a game is challenging as the host is caught between expensive hosting resources (servers, bandwidth, support), demanding users and the unknown initial popularity of a game.

Cheating is another challenge for on-line games. Clients are in control of their own machines, which are running a portion of the game code. By modifying the game however they see fit, cheaters are able to achieve numerous advantages over other players. These types of advantages range from fabulous wealth, to perfect aim, to being able to see through walls or outrun speeding bullets. While industry efforts to fix the problem have made some progress at preventing cheating, every popular on-line game suffers from cheats of one sort or another. As the stakes have become higher in these games, cheating has grown more prevalent and more debilitating. For persistent on-line games, this cheating amounts to essentially stealing money from other players or from the game company, and can quickly ruin a virtual world's economy. For any game, the presence of cheaters detracts from the play experience of others, decreasing the satisfaction of the cheat-free population, ultimately to the detriment of the game's popularity.

1.2 History

In order to understand on-line gaming, some historical perspective is useful. The history of on-line gaming is as brief as the history of computer networking. One of the earliest known examples of networked gaming was Spacewar, a two-player game written for the University of Illinois PLATO network in 1969 by Rick Blomme. The PLATO network utilized 512x512 access monochrome graphics capabilities and connected players at low latencies.

On-line gaming throughout the 1970's and 1980's typically involved networked games over the PLATO network, or variants of them ported to the UNIX platform. Some examples of these are MUDs, xtrek, and Zork. In the early 1990's the proliferation of Internet connectivity across universities and the popularity of Id Software's Doom, a first-person perspective shooting game, marked the rise of on-line gaming in popular culture. Doom was not designed to be played over the Internet, but rather on a local area network (LAN). However, demand for play was great enough that the game was retrofitted to be compatible with LAN emulator tools such as Kali that made play over the Internet possible. This game was nearly unplayable over a modem connection due to latency and reliability assumptions in the networking code. Id's next game, Quake (1996), was designed to use the Internet Protocol from the ground up and attempted to smooth client gameplay experience with client-side prediction. Quake was popular enough to hold tournaments for thousands of dollars of prizes and get millions of people to invest money in consumer graphics cards for games. On-line gaming has only gotten more popular since that time, and further driven advances in specialized networking, user-input, and even furniture devices designed to enhance the gaming experience. Today, a modern popular game played on-line can garner hundreds of thousands of players per day, and there are dozens of such games.

Academic research into on-line gaming covers a diverse set of research disciplines, such as artificial intelligence [9, 22] human-computer interaction [14], economics [27], computer graphics[7, 26, 63], security [6] and networking [32, 15, 13]. The overlap of networking research and games research is of particular interest to us. Networked gaming research has covered topics such as effects of latency on user performance, player characterization, traffic characterization, wireless gaming, hosting infrastructure for games and cheat detection and prevention.

1.3 Introducing GameCorp

In this thesis we take the perspective of GameCorp, an imaginary company in the business of publishing successful on-line games. GameCorp knows there is tremendous success to be had in the market. For instance, a single large on-line game, World of Warcraft, retains over 6 million subscribers paying monthly fees ($US 15 in the United States). However the market for on-line games is extremely competitive, with many games released to little acclaim or success, despite enormous development and initial launch costs, and substantial recurring maintenance and content development costs. In addition to the usual challenges of creating and selling a good product, GameCorp faces three special difficulties. First, the on-line gamer is an infrequently studied entity whose behaviors are relatively unquantified. GameCorp resolves to gain more knowledge about the average gamer by studying the network-observable behaviors and challenges faced by average gamers. Second, the architecture used to host modern on-line games is expensive and scales poorly. GameCorp resolves to develop an alternate architecture for hosting games that scales. Third, modern on-line games are plagued by a bewildering variety of cheats. GameCorp resolves to address cheating in on-line games.

Broadly, it is our intention to show techniques by which the state of the art in hosting games and preventing cheating can be advanced. We view this as important both for the continued success of on-line gaming as well as for the shared advances between on-line games and other applications. Interactive on-line applications of many sorts, such as military or disaster simulations, distance learning, and interactive storytelling share overlap with the networked aspects of on-line games, and technologies and results garnered from advances in gaming can benefit these other applications as well.

1.4 Thesis Overview

GameCorp is creating a compelling game to be hosted and played on-line, and would like to release it. However, there are two central challenges GameCorp must address: how to cheaply host the game and how to ensure there are no cheats that will ruin the game. We summarize GameCorp's investigation by chapter.

- Chapter 2 addresses the characterization of gamers and game server workloads, with the intention to determine ways of decreasing hosting costs. We evaluate player characteristics such as attention span and loyalty to a server as well as game server characteristics such as load variation and load periodicities. We evaluate and reject the hypothesis that multiple client server games can be efficiently hosted on the same hardware.

- Chapter 3 explores a promising alternative to the client server hosting model: *public server*. While the public server model allows for user hosting and user content, thereby alleviating the publisher's workload, it does not allow for a large-scale gameplay experience and it burdens users with a server selection task. We present a design allowing for large-scale gameplay in the public server model, as well as a geography-based redirection service to address the server selection problem.

- Chapter 4 provides an overview of cheating in on-line games and introduces our classification of all client-side cheats into four distinct categories. We then treat a specific category (information exposure) in greater depth and present the *protected real-time strategy* protocol for decreasing information exposure in peer-to-peer games. We show the performance characteristics of protected RTS via game trace driven simulation.

- Chapter 5 summarizes our results and outlines future research.

Chapter 2

Characterizing Game Workloads

2.1 Introduction

On-line gaming is an increasingly popular form of entertainment on the Internet, with the on-line market predicted to be worth over $5 billion dollars in 2008 [25]. As an example of a popular, money-making game, EverQuest [30] has over 450,000 subscribers paying a monthly fee and purchasing expansions. Unfortunately for game companies such as GameCorp, the success of a game is highly unpredictable. To make matters worse, there are substantial costs in developing and hosting on-line games. As a result, such companies are increasingly exploring shared, on-line hosting platforms such as on-demand computing infrastructure provided by companies such as IBM and HP [46, 48, 86, 39, 47, 62, 78, 81]. In this chapter GameCorp undertakes a study of the efficacy of such an approach.

In order to judge the feasibility of large-scale game multiplexing, it is important for

GameCorp to understand how gamers and game workloads behave. Understanding the behavior of players, the predictability of workloads, and the potential for resource sharing between applications allows infrastructure to be tailored to the needs of games. While there has been a substantial amount of work characterizing web and peer-to-peer users and workloads [21, 40], there is very little known about game players and workloads.

In order to provide insight into such issues, this chapter examines several large traces of aggregate player populations of popular games as well as the individual player population of a busy game server. We present a detailed analysis of on-line game players and workloads that targets several key areas which are important to game and hosting providers including:

- *How easy is it to satisfy gamers?*: One of the key issues in providing a successful game is to understand how players connect to servers and how long they play on them. By understanding what players are willing to put up with, game and hosting companies can tailor their infrastructure and content to maximize player satisfaction. For example, one of the challenges with using on-demand computing infrastructure for games is the latency associated with re-purposing a server. It would thus be useful to characterize how patient game players are in connecting to a game before deploying such infrastructure. To this end, we characterize individual player behavior of an extremely popular Counter-Strike game server over a long period of time. Our results show that gamers are an extremely difficult set of users to satisfy and that unless game servers are properly set up and provisioned, gamers quickly choose to go elsewhere.

- *How predictable are game workloads?* Another problem in hosting on-line games is determining the amount of hardware and network bandwidth that is required. Hosting a game is an expensive proposition, costing the game provider more than 30% of the subscription fees in just hardware and bandwidth per month [68]. Hosting is made all the more difficult by variations of popularity as the game moves through its life cycle. Game companies face the provisioning problem both in determining the amount of resources to provide at launch time and in allocating spare resources to support dynamic usage spikes and subscriber growth. Characterizing the diversity and predictability of game workloads allows companies to more accurately provision resources. To this end, we examine the real-time aggregate game player population of more than 550 on-line games. Our results show that game popularity follows a distinct power law distribution making the provisioning of resources at launch-time extremely difficult. However, as games mature, their aggregate populations do become predictable, allowing game and hosting companies to more easily allocate resources to meet demand.

- *Can infrastructure be shared amongst game and other interactive applications?* With the advent of commercial on-demand computing infrastructure, it is becoming possible to statistically multiplex server resources across a range of diverse applications, thus reducing the overall hardware costs required to run them. In order for such shared infrastructure to provide any savings, peak usage of applications must not coincide. To characterize the amount of sharing benefit that is available, we examine the usage behavior of a number of popular on-line games and compare them against each other and against the usage behavior of several large distributed web sites. As on-demand infrastructure is distributed, we also examine the client load of a number of servers based on geographic region. Our results show that usage behavior of interactive applications follows strict, geographically-determined, time-of-day patterns with limited opportunities for resource sharing.

Section 2.2 describes the methodology behind our study. Section 2.3 analyzes properties of individual gamers. Section 2.4 describes trends of on-line gaming in aggregate. Section 2.5 evaluates the potential for multiplexing games and web traffic together, and Section 2.7 discusses our conclusions.

`cs.mshmro.com` trace	
Start time	Tue Apr 1 2003
End time	Mon May 31 2004
Total connections	2,886,992
Total unique players	493,889

GameSpy trace	
Start time	Fri Nov 1 2002
End time	Fri Dec 31 2004
Total games	550
Total player time	337,765 years

Steam CDN trace	
Start time	Mon Sep 27 2004
End time	Mon Apr 8 2005
Content transferred	6,193 TB
Average transfer rate	3.14 Gbs

Table 2.1: Data sets

2.2 Methodology

The study of on-line game usage is typically limited due to the proprietary nature of the industry. To overcome this, we have collected several unique data sets that allow us to analyze properties that have not been possible previously. These data sets include the following:

Individual player data: In order to study the behavior of individual players playing a representative on-line game, we examined the activity of one of the busiest and longest running Counter-Strike servers in the country located at `cs.mshmro.com` [66, 33]. Counter-Strike (a Half-Life modification) is currently the dominant on-line game with the largest service footprint of any game at 35,000 servers and over 4.5

billion player minutes per month [3]. Of all of the active Counter-Strike servers, `cs.mshmro.com` is among the busiest 20 servers as ranked by ServerSpy [79]. The server averages more than 40,000 connections per week, has hosted more than 400,000 unique players in year 2004, and has logged more than 60 player years in activity since its launch in August 2001. Table 2.1 describes the trace collected from the server.

GameSpy aggregate player population data: One problem with measuring on-line game usage is the limited access to game server hosting data. Game companies typically keep the access and usage behavior of their players confidential. There are two factors that enable the measurement of aggregate game player populations, however: (1) on-line games use a centralized authentication server to keep track of the players that are playing and (2) information on overall player numbers per game is usually exported publicly. Several game portal services collect such player numbers over a large number of games and report the information in real-time. Among these services is the GameSpy network, which provides real-time player population data on individual games in a structured format that can readily collected and analyzed [36]. Currently, there are over 550 on-line games that are being tracked across various genres including first-person shooter games (FPS), massively multi-player on-line role-playing games (MMORPG), real-time strategy games (RTS), card and board games, and sports games. To study on-line game population behavior, we have collected a data feed from GameSpy for more than two years since November 2002. Our redundant collection facility periodically samples the GameSpy data every 10 minutes. Note that the availability of the data is sensitive to many factors, including service outages at the portal and our own outages. These outages have been manually removed from the data analysis. Table 2.1 describes the data set which includes over 50 million measurements and represents more than 300,000 years of player time spent on games over the course of a two year period.

Content-distribution networks: One of the common features of on-line games is their ability to dynamically update themselves. To support this feature, many games employ custom, game-specific, content distribution networks that deliver new game content and software patches to clients when needed. One such network is Steam [87], a multi-purpose, content-distribution network run by the Valve corporation which is used to distribute run-time security modules as well as client and server software patches for Half-Life and its mods such as Counter-Strike and Day of Defeat. The network consistently delivers several Gbps of content spread across over 100 servers. In order to analyze the resource usage of Steam, we have collected its data feed over a 6 month period, a duration that has seen Steam deliver more than 6 petabytes of data. Table 2.1 describes the trace collected.

2.3 Gamers As Individuals

It is important for game providers to understand the usage behavior of its players in order to adequately address their needs. In order to study player characteristics, we analyze the trace of `cs.mshmro.com` to track individual gamers throughout their play cycle. Specifically, we track gamers attempting to connect to the server, gamers playing on the server, and the likelihood of a gamer returning to the server. We first demonstrate that gamers are difficult to please. In particular, they 1) have no tolerance for busy servers, often connecting once while the server is busy and never reconnecting again for the entire trace, 2) have very specific gameplay needs and if those needs are not met in the first few minutes of play, their likelihood of continuing to play at the server drops off dramatically, and 3) they often have no loyalty or sense of community tied to a specific server and do not return after playing a handful of times. For those that do return often, we also demonstrate that their session times show a marked decline and their session interarrival times show a marked increase just as they are ready to quit playing on the server altogether.

2.3.1 Gamers Are Impatient When Connecting

Quantifying the patience of on-line gamers is important for adequate server provisioning. For some Internet applications, such as web-browsing, users are known to be impatient [12]. For others, such as peer-to-peer services such as *Kazaa*, users

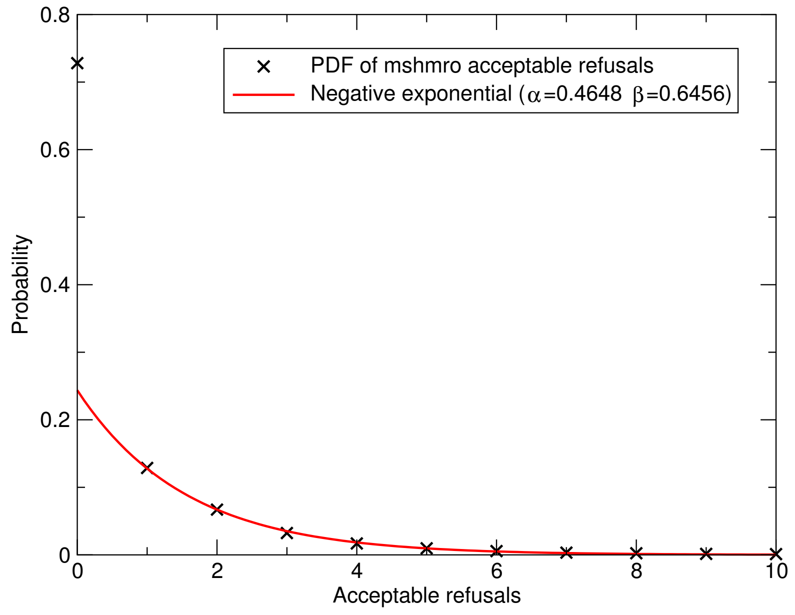
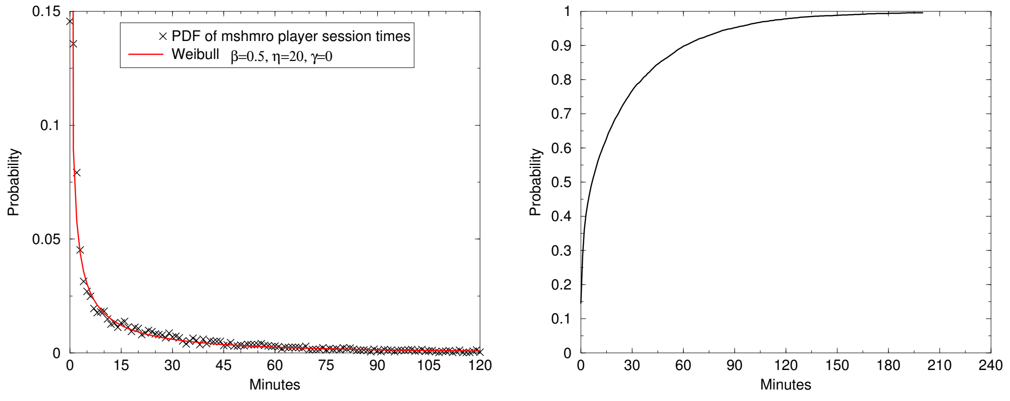


Figure 2.1: Player impatience based on acceptable refusal ratio

are very patient [40].

Our trace of `cs.mshmro.com` records successful connections as well as connection attempts, when players connect to the server and are refused service. The latter is extremely common; every day, the server turns away thousands of people. Browsing the trace, it is not unusual to see the same player reconnect to the server several times in a row, waiting for a spot on the server to free up. We operate on the assumption that a player's willingness to reconnect to the same busy server repeatedly is an indication of their patience.

In order to quantify player patience we calculate for each player their total number of gaming sessions on the server and their total number of failed attempts to play,



(a) PDF (b) CDF

Figure 2.2: Session time results for `cs.mshmro.com` trace

and compute the ratio as an indicator of the number of acceptable refusals per player. Figure 2.1 shows the probability distribution of acceptable refusals per player. As the figure shows 73% of the players are unwilling to reconnect to the server even once. One of the reasons players do not reconnect is that game clients have a "Quick Start" mechanism that many players use. The mechanism works by downloading a list of candidate servers from the master server and cycling through them one by one until a successful session is established. Thus, such clients may not lack patience, but rather are automatically redirected elsewhere. For the rest of the players, however, 13% are willing to reconnect one time on average with the percentage sharply decreasing with successive refusals. Aside from the first data point, the rest of the graph represents a client's patience in connecting to our busy server and, not surprisingly, can be fit very closely with a negative exponential distribution. As Figure 2.1 shows, a negative exponential distribution with parameters $\alpha = 0.4648$ and $\beta = 0.6456$ fits the data with a correlation coefficient of 0.999. Players, therefore, exhibit a remarkable degree of impatience with busy game servers.

2.3.2 Gamers Have Short Attention Spans

Using the same trace, we extracted the total session time of each player session contained in the trace. Figure 2.2 plots the session time distributions of the trace in unit increments of a minute. The figure shows, quite surprisingly, that a significant number of players play only for a short time before disconnecting and that the number of players that play for longer periods of time drops sharply as time increases. Note that in contrast to heavy-tailed distributions reported for most source models for Internet traffic; the session ON time for game players is not heavy-tailed. To further illustrate this, Figure 2.2(b) shows the cumulative density function for the session times of the trace. As the figure shows, more than 99% of all sessions last less than 2 hours.

Unlike the player patience data, session times can not be fitted with a simple negative exponential distribution. However, the data can be closely matched to a Weibull distribution, a more general distribution that is often used to model lifetime distributions in reliability engineering [74]. Since quitting the game can be viewed as an attention “failure” on the part of the player, the Weibull distribution is well-suited for this application. The generalized Weibull distribution has three parameters β, η, and γ and is shown below.

$$f(T) = \frac{\beta}{\eta}(\frac{T-\gamma}{\eta})^{\beta-1}e^{-(\frac{T-\gamma}{\eta})^{\beta}}$$

In this form, β is a shape parameter or slope of the distribution, η is a scale parameter, and γ is a location parameter. As the location of the distribution is at the origin, γ is set to zero, giving us the two-parameter form for the Weibull PDF.

$$f(T) = \frac{\beta}{\eta}(\frac{T}{\eta})^{\beta-1}e^{-(\frac{T}{\eta})^{\beta}}$$

Using a probability plotting method [74], we estimated the shape (β) and scale (η) parameters of the session time PDF. As Figure 2.2(a) shows, a Weibull distribution with $\beta = 0.5$, $\eta = 20$, and $\gamma = 0$ closely fits the PDF of measured session times for the trace.

This result is in contrast to previous studies that have fit a negative exponential distribution to session-times of multiplayer games [44]. Unlike the Weibull distribution which has independent scale and shape parameters, the shape of the negative exponential distribution is completely determined by λ, the failure rate. Due to the memory-less property of the negative exponential distribution, this rate is assumed to be constant. Figure 2.3 shows the failure rate for individual session durations over the trace. As the figure shows, the failure rate is *higher* for flows of shorter duration, thus making it difficult to accurately fit it to a negative exponential distribution. While it is difficult to pinpoint the exact reason for this, it could be attributed to the fact that Counter-Strike servers are notoriously heterogeneous. Counter-Strike happens to be one of the most heavily modified on-line games with support for a myriad of add-on features [41, 4]. Short flows could correspond to players browsing the server's features, a characteristic not predominantly found in other games. As with player patience, it may be possible to fit a negative exponential for longer session times. As part of future work, we hope examine this as well as characterize session duration distributions across a larger cross-section of games to see how distributions vary between games and game genres.

2.3.3 Gamers Are Not Loyal

Public-server games such as Half-life provide users with a large choice of servers located all around the world. Gamers can switch between servers as often as they like. Some reasons to continue playing on the same server are simplicity, a known low-latency connection, preference for server options, or a sense of community. It is natural to wonder whether servers continue to serve the same group of clients and to what extent these reasons or others keep clients at a specific server.

Our trace contains the connection records for each client via their unique player identification number (WONID). We quantify loyalty to the server by counting

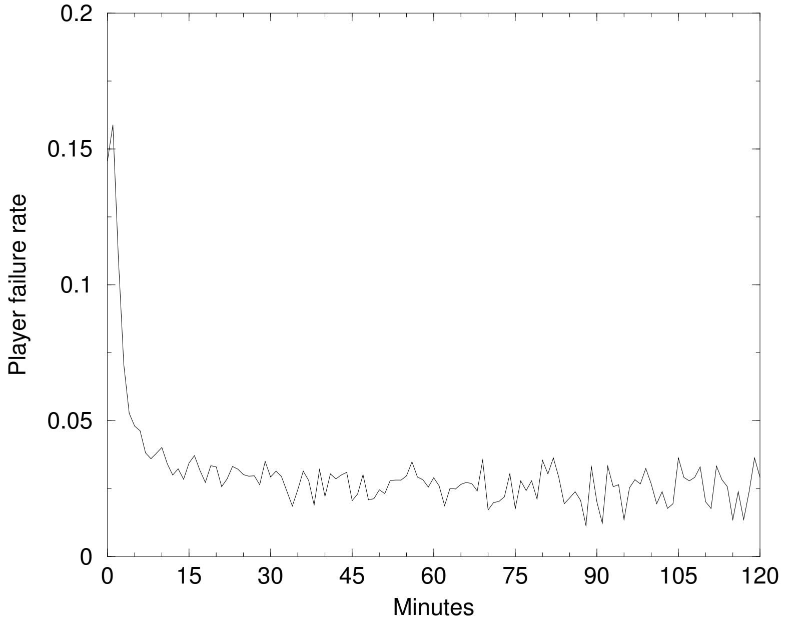


Figure 2.3: Player failure rates for individual session times for `cs.mshmro.com` trace

the number of times a player returns to play after a successful playing session. Figure 2.4 shows, on a logarithmic scale, the cumulative distribution of additional game sessions per player for players who returned at least once to the server. As the figure shows, 42% of the players in our trace returned to play only once and 81% played less than 10 times. On the other hand, the top 1% of loyal gamers return to play many thousands of times (hence the logarithmic scale). It appears that the majority of clients have very little loyalty to public servers, and only a small fraction have grown strongly attached. We hypothesize that, due to a large population of servers to choose from (over 30,000), clients rarely select the same

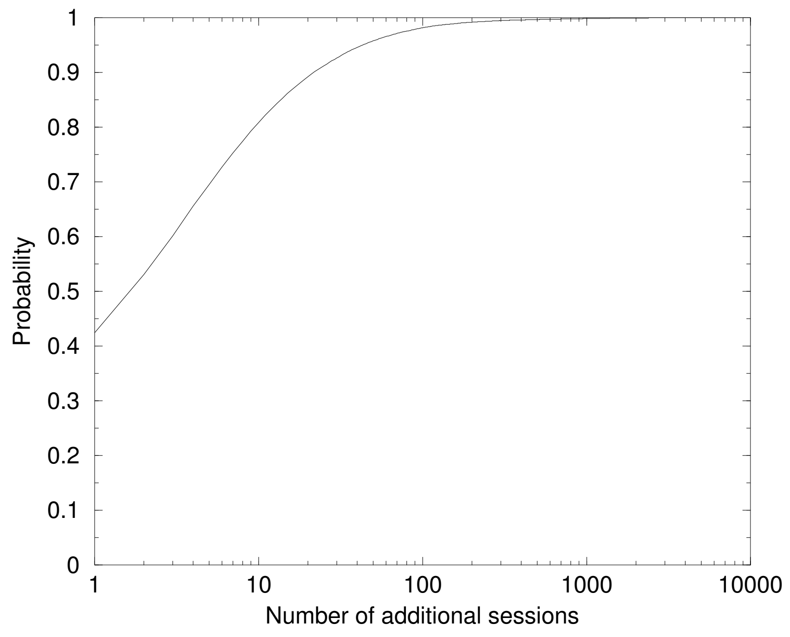
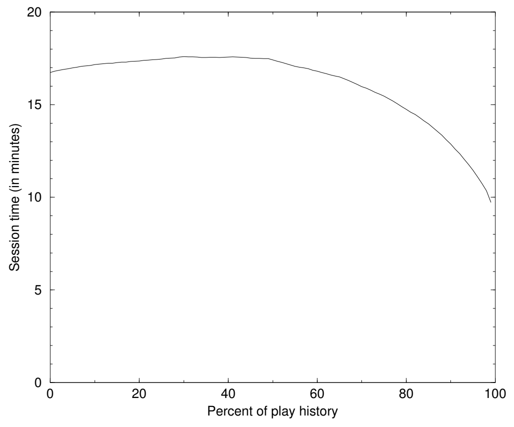


Figure 2.4: CDF of sessions per player on the server

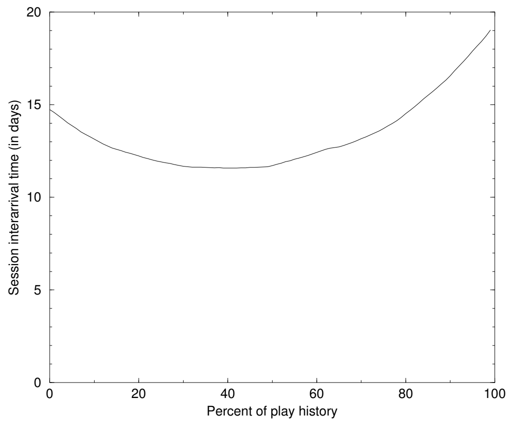
server twice.

2.3.4 Gamers Reveal When They Lose Interest

Players of a game have some discretion about how frequently they play a game and for how long. Players often lose interest in a game and cease playing altogether at some point. Before that happens, however, there may be noticeable indications that their interest is waning. Such indications are extremely useful to game providers who can detect waning interest and react to it on a macroscopic level with new content or on a per-player basis via customized incentives for continued play.



(a) Average session time



(b) Average session interarrivals

Figure 2.5: Player behavior throughout their playing careers

We determine the average player interest curve by calculating each player's sequence of play sessions from their first session to their last recorded session. This is a player's *play history.* Since each player may progress through his or her game interest at a different rate, we normalize each of these data sets based on the duration each player is active on the server. We then examine the average session times and session interarrival times of all players throughout their playing careers. Figure 2.5(a) shows that player session times are relatively constant halfway through their play history and fall off to just more than 50% of the initial session time before the player loses interest completely. Figure 2.5(b) shows that the time between player sessions is minimized before the halfway point and increases steeply until the player's interest has fully waned. We conclude that player session times and session-interarrival times can be used as an early indicator of peaking player

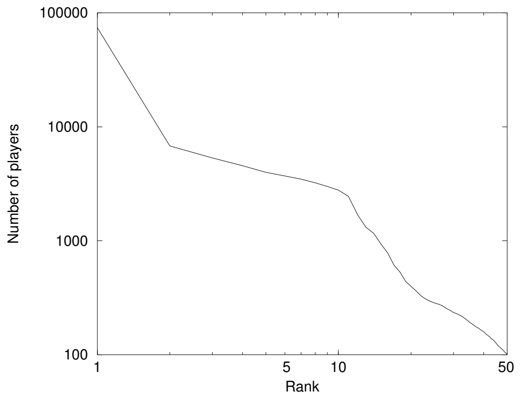
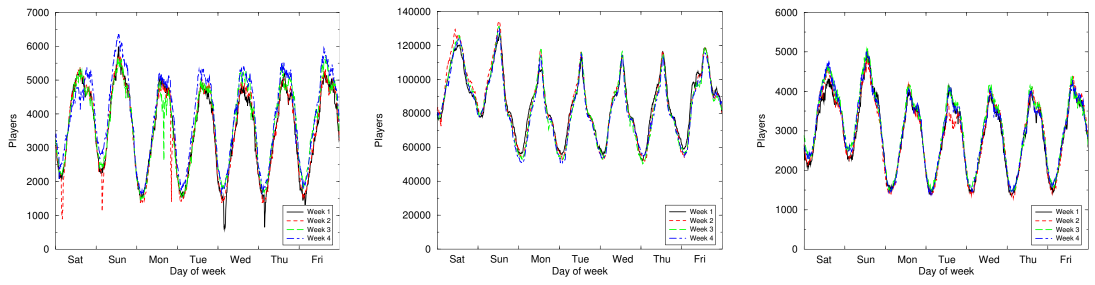


Figure 2.6: Game popularity distribution averaged over nine months (log scale)

interest and that game publishers should use these measurements to trigger the delivery of new content or incentives for the individual player.

2.4 Game Populations

As shown in Section 2.3, under-provisioning resources for a game can quickly drive gamers away. Over-provisioning, on the other hand, can be costly. We look at two facets of gaming integral to successful game provisioning: overall game popularity and predicting game workloads. We show that (1) there are, and will be, very few extremely popular games, and (2) game workloads are periodic and predictable over short-term intervals.



(a) America's Army (b) Half-Life (c) Neverwinter Nights

Figure 2.7: Player load for three popular games over a 4-week period

2.4.1 Game Popularity Follows a Power-law

To determine the distribution on-line game popularity, we analyzed the GameSpy data set described in Section 2.2. By averaging the number of players per game over the trace, we ranked each game based on its popularity. Of the games, we consider only the top 50 games, as the remaining games averaged a minimal number of players throughout the trace. Figure 2.6 shows the popularity data on a log-log scale. As the figure shows, this distribution is very heavily skewed in favor of the most popular games, with the first ranked game having over ten times the number of players of the next most popular. This distribution of popularity is most similar to a power-law distribution. Power-law distributions are of the form $y = ax^{\lambda}$ and occur in a number of places including the frequency of words in the English language, the popularity of web pages, and the population of cities. An intuition for these distributions is that whenever choices are made between many options, and each choice affects other choices, the choices tend to pile up on a few popular selections. A perfect power-law distribution would graph as a straight line on a logarithmic scale in both the x and y axis. The relatively straight line (correlation coefficient -0.98 for a simple linear regression) demonstrates that the GameSpy data does follow a power law distribution. This distribution has an interesting, albeit unfortunate, implication for provisioning server resources for on-line games: the host must plan for several orders of magnitude of change in popularity (and therefore resources) in either direction. As a result, this indicates that on-demand infrastructure can significantly reduce the costs and risks of launching and hosting on-line games.

2.4.2 Game Workloads Have Varying Degrees of Predictability

Accurately predicting game workloads allows game hosting providers to allocate the appropriate amount of resources for a game. In order to determine whether this is feasible, we analyze the GameSpy trace for different sets of games. Specifically, we investigate whether any simple trends or patterns can be used to accurately predict the game workload, whether the workload is stable and if so, over what time scale.

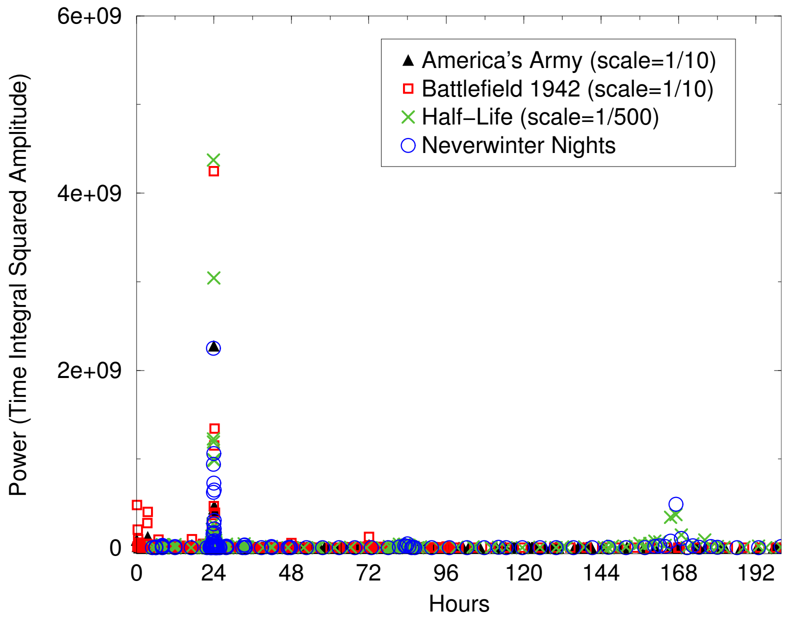


Figure 2.8: FFT of the player load from four games over one year.

Game Workloads Exhibit Predictable Daily and Weekly Changes

Intuitively, it is reasonable to assume that usage is strongly tied to the daily and weekly activities of players. Figure 2.7 shows the global player population of four consecutive weeks starting from 3/1/2003 for three popular games: America's Army, Half-Life, and Neverwinter Nights. As expected, the figure shows that the workload has regular daily cycles and that over this one month period the workload does not vary significantly from week-to-week. In fact, for all three games, the trends as well as the maximum and minimum points match up at identical points in time during the week. We observe similar results over other parts of the year with the only anomalies caused by service outages and by holidays. To further

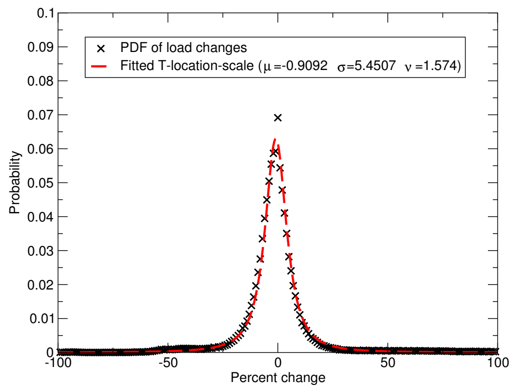


Figure 2.9: Instantaneous week-to-week PDF of percent load changes for the top 5 most popular games of 2004

demonstrate the cyclical nature of gaming workloads, we take one year's worth of game server load samples across a variety of games and plot the Fast Fourier Transform (FFT) of the data. The FFTs have been scaled so that they can be plotted together. As Figure 2.8 shows, the FFT contains strong peaks at the 24-hour cycle for each of the games. There is also a significant peak at the 168-hour (one week) cycle for two of the games as well. This corresponds to an increase in player usage on the weekends during some parts of the year. Papagiannaki et. al use wavelet multiresolution analysis (MRA) on another long-term data series [72], and model their series as a 12-hour and 24-hour cycle plus a trend. We were unable to apply this technique however, due to the reliance of wavelet MRA on resolutions that are factors of two apart. The difference between our two cycles is seven.

In order to quantify the week-to-week variation of game workloads, Figures 2.9,2.10,2.11 show the distribution of week-to-week load changes of the top 5 most popular

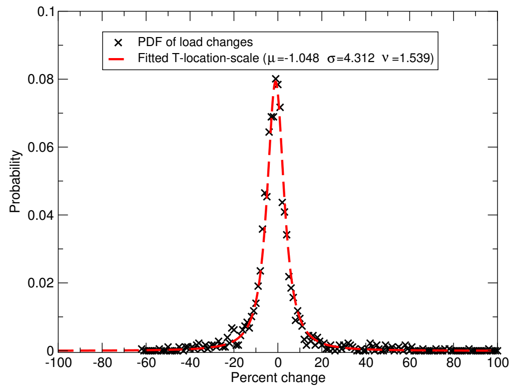


Figure 2.10: Mean week-to-week PDF of percent load changes for the top 5 most popular games of 2004

games during 2004: Half-Life, Battlefield 1942, Medal of Honor: Allied Assault, America's Army, and Neverwinter Nights. Figure 2.9 plots the distribution of instantaneous load changes between identical points in time of consecutive weeks, while Figure 2.10 plots the change in average daily load between the same day of the week of consecutive weeks. Finally, Figure 2.11 plots changes in maximum daily load between the same day of the week of consecutive weeks. The figures fit a 't' location-scale distribution, which has three parameters, a scale parameter $\sigma > 0$, a location parameter μ, and a shape parameter $\nu > 0$. The density function for this distribution is as follows:

$$f(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sigma\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}(\frac{\nu + (\frac{x-\mu}{\sigma})^2}{\nu})^{-\frac{v+1}{2}}$$

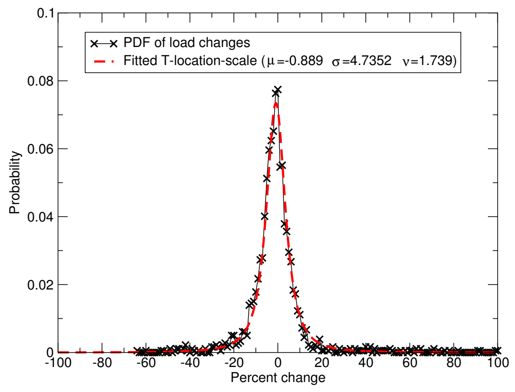
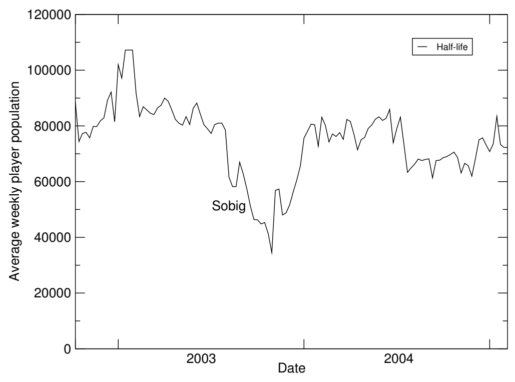


Figure 2.11: Max week-to-week PDF of percent load changes for the top 5 most popular games of 2004

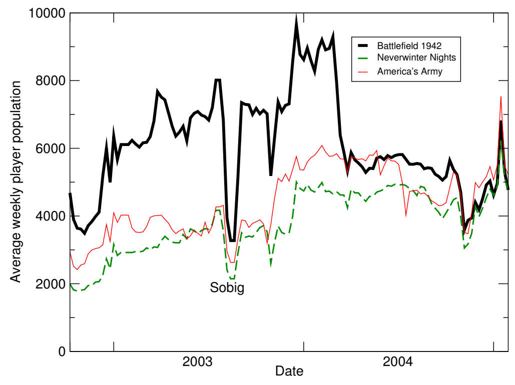
Note that if x is ‘t’ location-scale distributed, $\frac{x-\mu}{\sigma}$ is Student’s ‘t’ distributed with ν degrees of freedom. As illustrated in Figures 2.9,2.10,2.11 we find a very good fit for all the three plots. Based on this observation, we draw two main conclusions with regard to resource usage:

- As the figures show, almost all week-to-week load variations are under 10% of the previous week’s workload. Such behavior makes it relatively easy for game and infrastructure providers to provision and predict resource usage on a weekly basis.

- Further, the above distribution fitting of load variations indicates that it is feasible to model the week-to-week load variations using such standard distributions. We are exploring the feasibility of online parameter estimations for using this model in the resource provisioning.



(a) Half-Life



(b) Other games

Figure 2.12: Population trends for Half-life and other games after daily and weekly cycles are removed

Game Workloads Exhibit Unpredictable Long-term Fluctuations

While the daily and weekly cycles in server load are clear, the duration of our trace allows us to examine longer term cycles. We examine the trend of the most popular game, Half-life, as well as three games of similar popularity over the period of just over two years. We compute the trend as the moving average of the data with a window size of one week. Figure 2.12 shows the trends of the respective games. The underlying trend of these games does not reveal periodicities on a monthly timescale, and the limits of our trace prevent us from drawing any strong conclusions about annual cycles. There are several points in trace where the games appear to be synchronized, but the explanation for the concurrent peaks or valleys is not necessarily predictable. We observe peaks in all games near the Christmas season, but, for example, all four games experience a drop during the unpredictable weeks of the *Sobig* virus [16].

2.5 Impact on Infrastructure

With the movement toward hosted game services [65, 29] as well as on-demand computing infrastructure for games such as Butterfly.net [10], there has been a great deal of interest in reducing the cost of running game servers by sharing server resources dynamically across multiple games and applications. We explore two likely scenarios: hosting multiple games on the same servers, and hosting web sites along with game servers. In addition, we study the usage behavior of a content-distribution network for supporting games. Our results show that there are significant challenges in multiplexing interactive applications on the same server infrastructure and that only limited opportunities for reducing peak resource usage exist.

2.5.1 Game Workloads are Synchronized

There are two ways games can be multiplexed with each other. One way would be to coarsely and statically assign physical servers to particular games based on the popularity of the game. Results from Section 2.4 clearly show that this can provide a lot of benefit for game companies. Another way would be to dynamically re-allocate servers based on instantaneous demand for a particular game. An implicit

Game	Average number of players
Half-Life	80324
America's Army	5791
Battlefield 1942	5402
Neverwinter Nights	4579

Table 2.2: Mean player populations for week of May 23, 2004

assumption that gives value to the latter method is that different games have usage patterns that are substantially different. Thus, rather than have each game provision server resources based on the peak usage of their game, server resources would be provisioned for the global peak.

In order to investigate the extent to which different games can be multiplexed with each other, we examined the aggregate player populations of four popular games that span several genres. The games examined included FPS games (Half-Life, Battlefield 1942, and America's Army), as well as an MMORPG (Neverwinter Nights). Player populations of these games were collected over a one week period (Sunday May 23, 2004 to Saturday May 29, 2004) from the GameSpy trace. In order to compare the games directly, independent of their popularity, each game's population data was normalized by the mean population for that particular game during the week. Table 2.2 lists the mean player populations for the four games examined. Figure 2.13 plots the normalized player loads for the four games during the one week period. As the figure shows, player populations fluctuate significantly based on the time of day from lows close to half of the mean to peaks close to twice

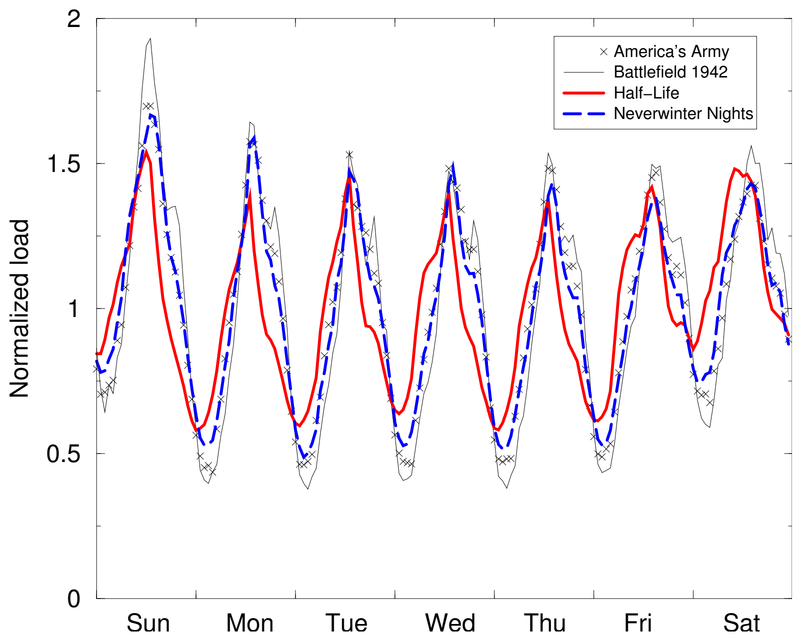


Figure 2.13: Aggregate normalized load across four popular games for week of May 23, 2004

the mean. In addition, populations across games have peaks in close proximity to each other, making it difficult to achieve significant statistical multiplexing gain between different games. Finally, as indicated in the FFTs from Figure 2.8, games show slight peaks on the weekends with slightly more players on-line than during the week.

2.5.2 Games and Interactive Application Workloads are Synchronized

While Section 2.5.1 shows the difficulty in obtaining statistical multiplexing gain between different games, on-demand computing infrastructure could still be useful

North American cereal manufacturer	
Start time	Mon Aug 13 2001
End time	Sun Aug 19 2001
Total requests	10,368,896
Content transferred	59.6 GB

North American credit card company	
Start time	Tue Aug 14 2001
End time	Mon Aug 20 2001
Total requests	112,590,195
Content transferred	366.4 GB

International beverage manufacturer	
Start time	Tue Aug 14 2001
End time	Sat Aug 18 2001
Total requests	11,932,946
Geographically resolvable	11,829,429
Content transferred	51.1 GB

Table 2.3: Web site logs for week of August 13, 2001

for multiplexing between other applications such as web servers. In order to examine this, we obtained web server logs over a week for three commercial sites. The sites included those for a North American cereal manufacturer, a North American credit card company, and an international beverage manufacturer. Table 2.3 describes the traces of the web servers, all from the week of August 13, 2001. The servers themselves were located in geographically distributed data centers and the individual logs from each site were aggregated and sorted into a single log file. Using these traces, we plotted the normalized load for the web server against the normalized global aggregate load of Half-Life during the same week in August 2004.

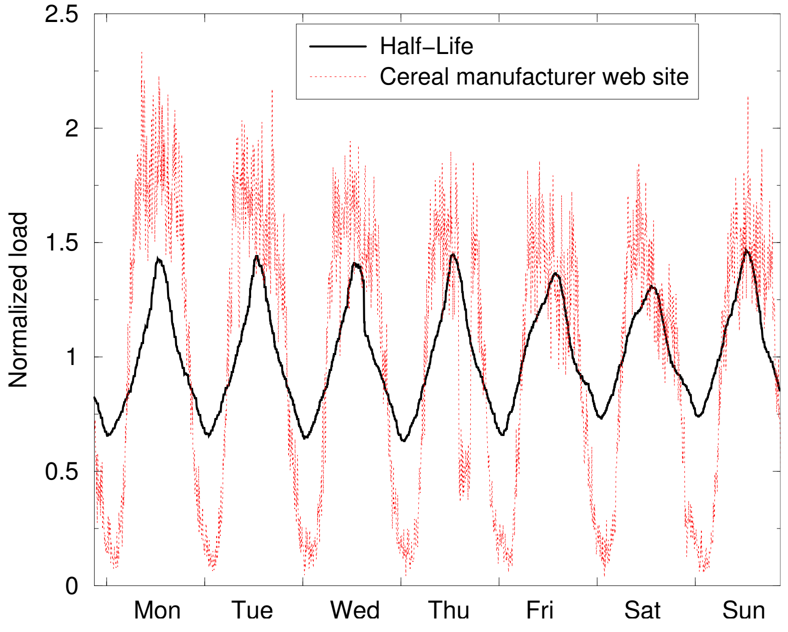


Figure 2.14: Aggregate normalized load between Half-Life and North American cereal manufacturer website

As Figures 2.14,2.15,2.16 show, workloads for web and on-line games share similar daily periodic peaks. This particular week of game traffic does not have a strong weekend rise (perhaps due to being from the summer), but the web traffic does slump during the weekends as Figures 2.14 and 2.15 show. Interestingly, Half-life shows considerably less variance than the North American websites, but similar variance to the international beverage manufacturer website. Intuitively, it makes sense that applications and web sites with global usage patterns are more consistently busy and have less daily variance. Due to the international popularity of Half-Life, its usage pattern is quite similar to that of the international beverage

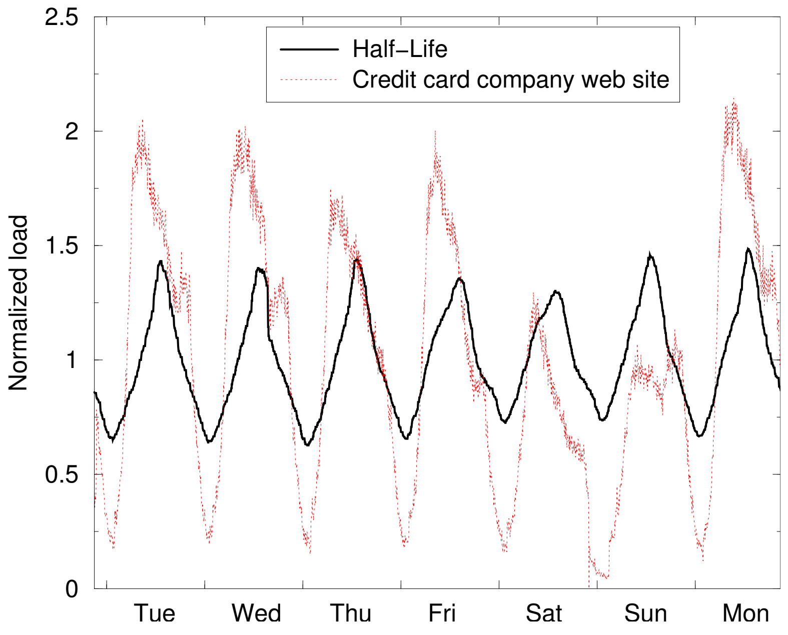


Figure 2.15: Aggregate normalized load between Half-Life and North American credit card company

company's web site. Overall, these results indicate that infrastructure sharing between applications during the week will have a somewhat limited benefit with some potential for multiplexing gain during the weekends and during the "off hours" for geocentric applications.

2.5.3 Games Exhibit Strong Diurnal Geographic Patterns

One of the salient features of globally distributed, on-demand computing infrastructure is that it can easily shift resources geographically close to where the demand is coming from. Intuitively, it makes sense that a predictable, diurnal pattern drives global resource consumption and hence, the provisioning of server

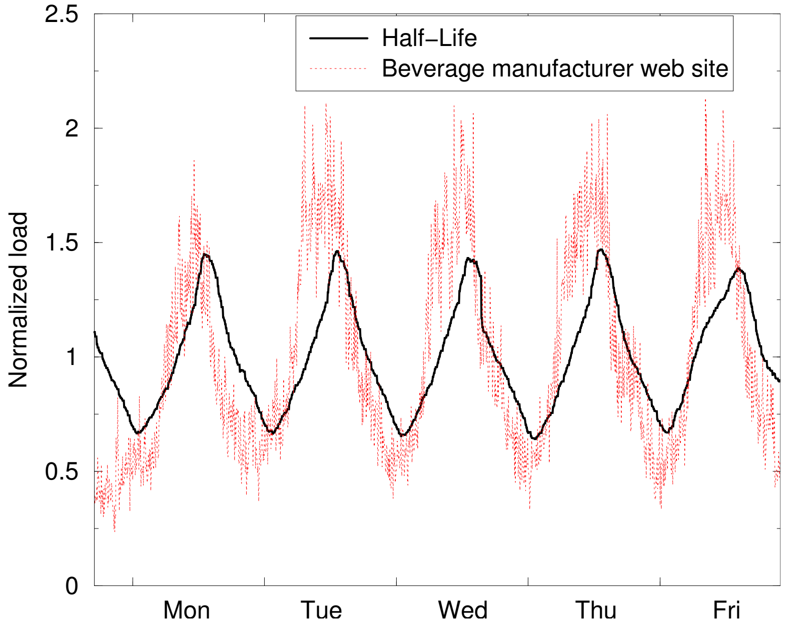


Figure 2.16: Aggregate normalized load between Half-Life and International beverage manufacturer

resources. This is especially the case for applications that require human participants such as games. To study this phenomenon, we examined a one-week period of `cs.mshmro.com` (Sunday May 23, 2004 to Saturday May 29, 2004). Using this log and a commercial geographic IP address mapping tool [38], the location of each player connecting was resolved. As Table 2.4 shows, a significant portion of the load is from outside of North America. Using the resolved connections, the per-continent load normalized by the mean connection arrival rate was plotted. As Figure 2.17 shows, each continent shows a predictable, diurnal pattern of activity with the only difference being a time-zone shift. It is interesting to note that in

Total connections	71,253
Geographically resolvable	30,226
From North America	9,414
From Asia	9,814
From Europe	8,788
From other continents	2,210

Table 2.4: Connection data for `cs.mshmro.com` for week of May 23, 2004

contrast to the Half-Life aggregate load and international beverage company web site load shown in Figure 2.16, the per-continent load of `cs.mshmro.com` exhibits a large variance similar to the North American web site loads shown in Figures 2.14 and 2.15. We hypothesize that when the usage patterns of international servers and services are broken out into individual regions, the resulting load variances are similar to those of regional servers such as the cereal manufacturer and the credit card company.

To test this hypothesis, we compared the per-continent load between `cs.mshmro.com` and the international beverage company web server trace [1]. Figure 2.18 shows the per-continent, normalized load of the game and web server for North America and Europe. The load from other continents shows similar results. As expected, the per-continent load fluctuations and variance are similar to those found in the two regional web sites. The figure also shows that usage of both applications are

[1]Note that a much larger percentage of the IP addresses in the beverage company trace is resolvable. This is due to the fact that the trace (and the set of IP addresses in it) is much older, giving services such as GeoBytes more time to identify their locations

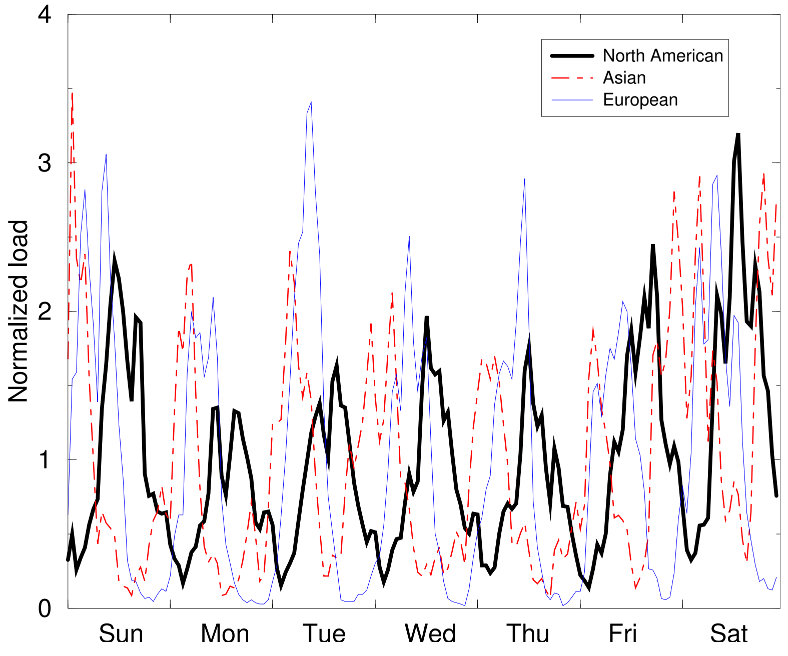
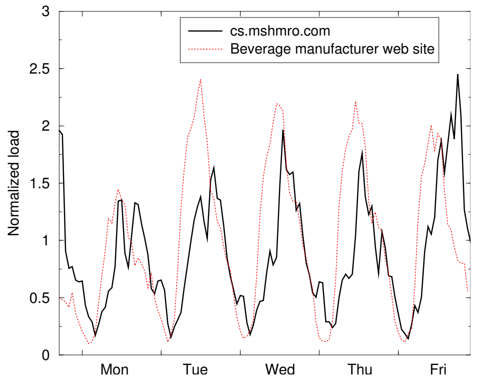


Figure 2.17: Aggregate normalized load per-continent for `cs.mshmro.com`

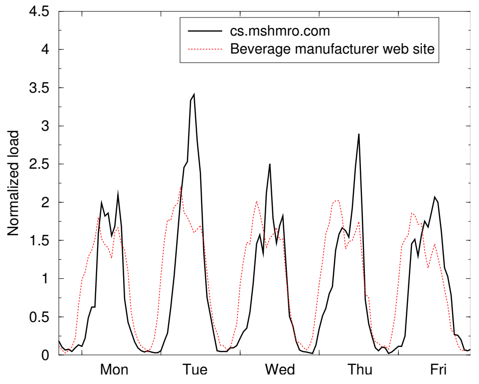
highly synchronized when broken down into geographic regions. The degree of synchronization thus limits the benefits that geographically distributed, on-demand computing infrastructure has on interactive applications such as games and web.

2.6 Game Updates Significantly Impact Resource Usage

The infrastructure required to host on-line games must also account for the mutability of the games over time. Software patches to fix bugs, prevent cheats, and deliver new content to end-users are an expected component of many on-line games. These patches can vary greatly in size, from a few bytes to several gigabytes. Understanding the impact of these patches on hosting, and adequately



(a) North America



(b) Europe

Figure 2.18: Normalized load for `cs.mshmro.com` and the international beverage company website

provisioning for them is an important part of supporting on-line games. We use the trace of the Steam content delivery network to examine this aspect of games. Our Steam trace includes the initial download of the popular FPS game *Half-Life 2* as well as a number of sizable content updates for both clients and servers.

The Steam network is utilized for both player authentication and content distribution. Players are authenticated to Steam for each game session, via the download of an authentication module. Content is distributed to players (and servers) via Steam at irregular intervals and irregular sizes. These two functions are not distinguished in the data set we have collected. However, we can differentiate them by utilizing the GameSpy dataset, which tracks player load, by assuming that player load and game authentication are linearly correlated.

As a way of validating that the Steam data and the GameSpy data are tracking the same thing (i.e. player load), we consider a week without a Steam update. Figure 2.19 shows a scatter plot of Steam data (in megabits per second) versus GameSpy data (in players), and the least-squares fit line. The correlation coefficient for this week is 0.86, indicating a roughly linear relationship. We attribute the inexact nature of the correspondence to small changes in the size of the authentication module and sampling error.

We use the GameSpy dataset to subtract away the authentication data from Steam and focus on the bandwidth requirements of a patch. Figure 2.20 shows a two week period of Steam activity, with a single patch occurring three days into the period. Also graphed is the authentication data component, computed from the GameSpy dataset with a ratio of players to megabits/second of 1 to 0.0291. By integrating these two signals and subtracting, we estimate the patch burden on Steam for this patch to be 129.7 terabytes, which is 30% of that week's total load including authentication.

We use this same methodology on four patches delivered during our trace, and chart the bandwidth impact of the patches over a two-week period in Figure 2.21. Three anomalies deserve explanation: patch *p3* is cut short of the full two week period analysis because of the release of *p5*, patch *p2* shows a rise in bandwidth after one week due to erroneous player data from GameSpy, and (according to Steam's press releases) the two weeks of patch *p7* contain numerous patches. One question to

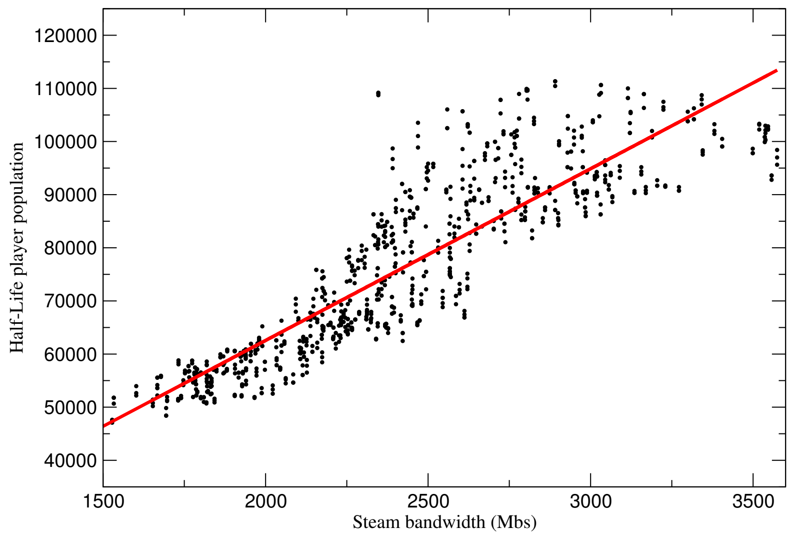


Figure 2.19: Half-Life player population versus Steam CDN usage

address is how long it takes to deliver a patch: the cumulative distribution function (CDF) of the patch delivery data in Figure 2.22 shows that 80% of the load occurs in the first 72 hours for the three single-patch traces, whereas the various patches in trace *p7* are delivered throughout a two-week period.

Our observations on patch distribution bring up several issues. We believe content delivery for games is a significant burden that must be provisioned for, as it can greatly increase the hosting bandwidth requirement. At this point, however, it is unclear what the optimal strategy would be for delivery and scheduling. Our initial observations are that to avoid the stacking effect seen in Figure 2.22, content should be spaced for delivery such that the bulk of each patch is delivered before the next patch begins. Further, if minimizing the combined content and authentication load

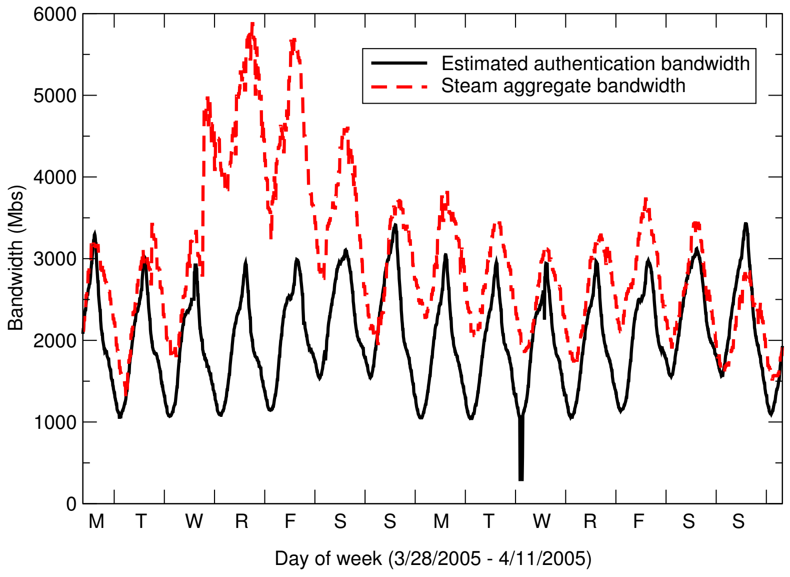


Figure 2.20: Steam bandwidth during a patch release

is a goal, then patches should be released at the lowest peak in the weekly and daily cycle. For example, a patch released Monday evening may potentially miss the daily afternoon peak as well as the weekend peak. As part of future work, we plan on examining the proper scheduling of patches based on measured game workloads.

2.7 Conclusions

On-line gaming is an increasingly popular form of entertainment on the Internet. Unfortunately, effectively hosting on-line games is a difficult, expensive proposition made more onerous by the lack of workload models for games or known characteristics of gamers. Due to the unpredictable nature of the popularity of a game,

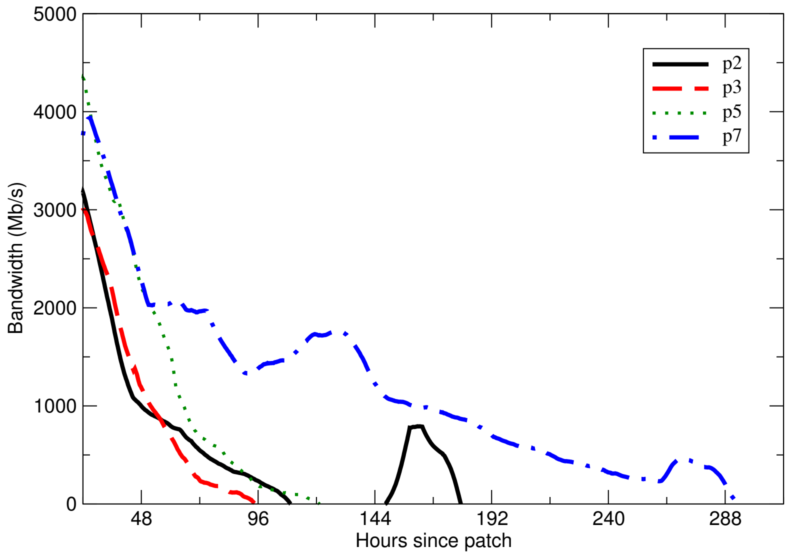


Figure 2.21: Excess bandwidth consumed by users downloading patches via Steam

combined with the high barrier to entry for hosting, a number of academic and industry projects have focused on providing a shared on-demand infrastructure to solve the hosting problem.

To understand the benefits of such infrastructure, this chapter presents a comprehensive analysis of on-line game players and game usage data collected from a number of unique sources. Our results show that gamers are difficult to satisfy throughout the gameplay process: they are likely to leave and never return if they can't connect, they are likely to leave within the first few minutes if they don't enjoy the server's characteristics, and they are unlikely to become loyal to a server. In addition, game popularity follows a power-law distribution, with a small number of games having orders of magnitude more players than the rest. This makes

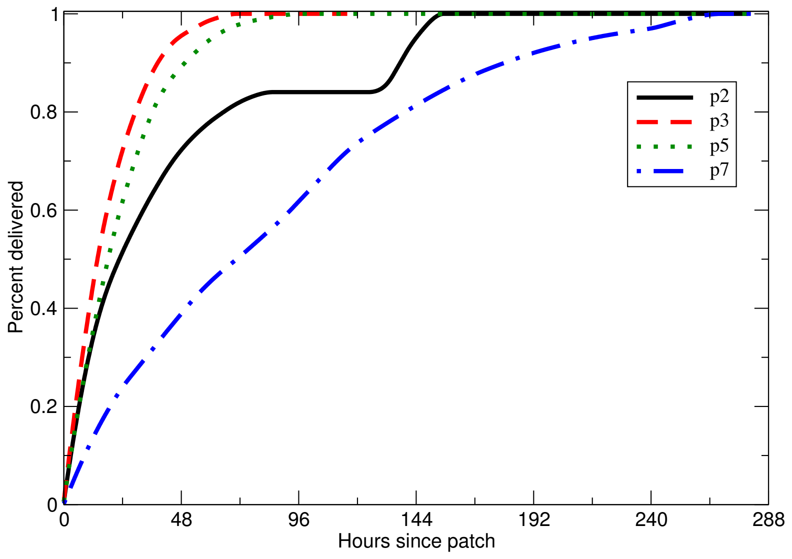


Figure 2.22: Cumulative distribution function of patch data.

resource provisioning very difficult for the initial release of a game when popularity has not been established and provides a promising area where shared hosting can provide benefit. Although initial provisioning is difficult, our results also show that once established, game workloads are relatively stable from week to week, allowing game providers to more easily allocate resources to meet demand. In addition, we determine that game workloads are synchronized amongst themselves and other interactive applications and that they follow strong diurnal, geographic patterns. Such synchronization makes it difficult to obtain statistical multiplexing gain between games and other interactive applications when using shared infrastructure. Finally, we show that game software updates provide a significant burden on game hosting and must be scheduled and planned for accordingly.

These conclusions paint a grim picture for the GameCorp goal of hosting an on-line game, as a game host must substantially overprovision their servers to cope with the daily peak of game traffic, as well as provision for the unknown initial game popularity, knowing that underprovisioning will result in frustrated gamers likely to quit. GameCorp takes these results as evidence that an alternate architecture for hosting should be explored.

Chapter 3

Public Server Games

In this chapter we describe the attractions and drawbacks of the public server architecture as compared to the more popular client-server architecture. The public server model scales more easily by relying on user-supplied hosting and user-generated content, but requires users to perform a difficult server selection task and does not allow for authenticated persistent content across public servers. To make the public server model more appealing, we introduce a geographic redirection service to address the server selection problem and a design for a public server MMO that allows for persistent content across public servers.

3.1 Introduction

In Chapter 2, GameCorp performed a measurement-based characterization of games, players and game workloads with an aim towards decreasing hosting costs by hosting many games on the same centralized server farm. Unfortunately, the study concluded that multiplexing gain across games or other interactive applications was unlikely to be of much benefit, due to synchronized workloads. In this chapter we investigate what can be done to leverage the public server architecture to address hosting costs. We first provide an overview of alternate architectures and game popularity in order to further motivate the hosting challenge, and then present the two problems we address in this chapter: server selection and persistent content on public servers.

On-line games can be broken down into one of three types of network architectures. In *client/server*, the game publishers operate game servers that host the game. The clients perform all communication with the company-controlled server. This architecture is relatively well-controlled compared to the other two types: peer-to-peer and public server. In *peer-to-peer*, there is no central server, or alternately, one of the peers playing the game is also the server for the game. In *public server* games, the code to run a game server is widely distributed, and anyone who likes may run a game server, or shut off their server on a whim.

On-line games of all architectures are enormously popular pastimes throughout the world. As of 2006, the dominant first-person shooter (FPS) game Half-life averages over 100,000 players concurrent players at all times, or over 9,000 player years played per month [87]. Casual games such as those played over MSN Zone [64] post similar concurrency numbers, and massively multi-player on-line (MMO) games such as World of Warcraft claim over 6 million subscribers paying monthly fees [90]. Figure 3.1 shows the steep growth in subscribers to MMO's since 1998 as compiled by Bruce Woodcock [84]. The games with the three largest contributions are Lineage, Lineage 2, and World of Warcraft.

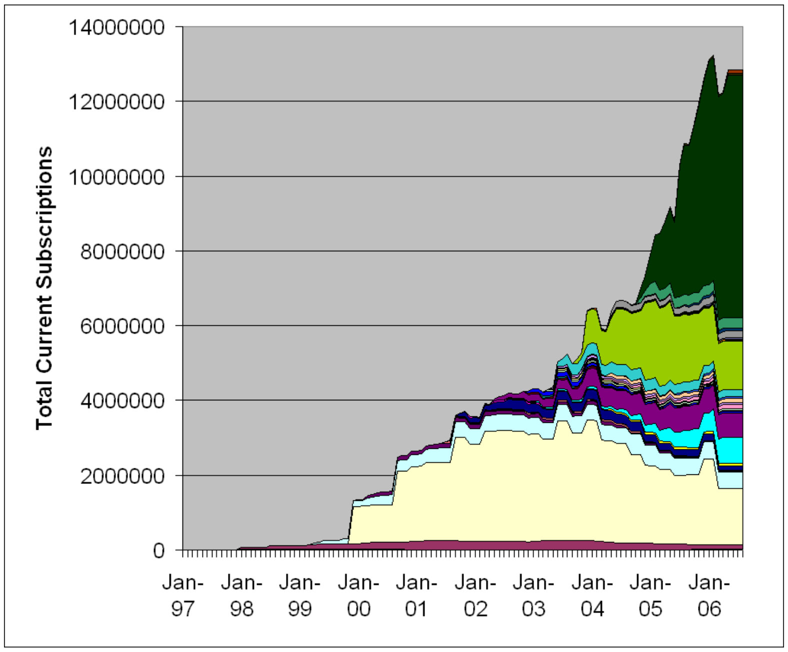


Figure 3.1: MMO subscriber growth over time by game

As the prospective host of a popular on-line game, GameCorp can expect to pay many costs. Direct costs include bandwidth, power and the amortized cost of the server machine itself. Indirect costs include continued content development, player support, as well as information technology support such as operating system maintenance and network maintenance. As the number of players increases, or their level of demands increase, these costs increase as well. An extreme example of difficulty in hosting is a popular MMO game. The commitment required to introduce a new MMO game is daunting; a popular game must have support at the server level for millions of simultaneous transactions, with server response-time needing to be as rapid as possible for a satisfying customer experience. The servers must be up 24 hours a day to support the global customer base. A large staff of game support staff must be constantly present to handle voracious customer needs, such as technical support, in-game errors, or reports of abuse. New content must be released for the persistent world regularly, meaning that it must be in development all the time. The supporting infrastructure required for a massive game to succeed is made all the more onerous by the fact that the on-line game market is extremely competitive and many games fail to generate revenue and are canceled.

Because of the scalability issues, large recurring costs and risk associated with hosting, GameCorp would like to investigate alternate architectures to mitigate these factors. The public server architecture is especially attractive to GameCorp because it offloads the burdensome requirement of hosting the game logic, dispute arbitration and network bandwidth to the public at large. Public server games such as Half-life and Neverwinter Nights [8] have demonstrated that users are willing to host games themselves, with their own servers and maintenance efforts. For example Half-life typically has around three thousand full servers, three thousand partially full servers, and eighteen thousand idle servers. These servers are all maintained by the Half-life gaming community, without any resources from the game company.

Furthermore, public server is an ideal architecture to allow for user generated content, as the users are running both the servers and the client themselves. In addition to granting players the ability to run game servers, game developers typically allow players to modify the game servers in specified ways, creating *mods* that add additional content to the game. Users are very willing to create new content for games that they can serve. For instance the number of mods created for Neverwinter Nights (over four thousand) dwarfs the game content expansion created by the game developers (seven).

The public server architecture simultaneously eliminates the costly hosting expense of an on-line game and also provides an avenue for greatly extending the life-cycle of the game via user-created content. However, the public server architecture has some problems. First, there is no persistent data shared across public servers, and the number of players that can be hosted on a given user's computer is typically small, for example from ten or twenty for a first-person shooter, to around one hundred for the lower processing requirements of a text-based MUD. This means that people desiring a massively multi-player on-line experience will likely be less interested in a public server game. In addition to scalability issues, with tens of thousands of servers to choose from, finding a server with the desired gameplay can be challenging. Players must solve this problem amidst a changing set of servers each time they want to play the game.

GameCorp would like to build a public server architecture that addresses these two drawbacks of (1) small-scale gameplay without persistent content and (2) difficulty finding a server to play on, and use it for the next big on-line game. In Section 3.2 we address finding a good server in the public server architecture, and in Section 3.3 we address the issues of persistent content and “massiveness” in the public server architecture.

3.2 Public-server Games and Geographic Redirection

3.2.1 Server Selection and Overflow Connections for FPS Games

In the public server model, the standard technique for a client to find a suitable server to play on is to first download a list of all of the currently registered servers from the central registry. An automated process then contacts each one of them (or sometimes just a random subset) and retrieves important information about the server, such as how many players are playing, what the latency is to the server, and what map is being played. The player can then sort this list locally by any criteria and connect to their chosen server. Unfortunately, such mechanisms do not scale, do not work effectively, and consume a significant amount of bandwidth for a popular game with hundreds of thousands of players and tens of thousands of servers.

In this section we address the issue of server selection in the public server setting. We build on the work of the Internet measurement community whose efforts have made it possible to locate an IP address geographically in the world as well as estimate the latency between two arbitrary IP addresses [57, 52, 70]. One problem closely related to ours is that of server selection for content distribution networks [58, 80]. The key difference between the server selection problem for public-server games and the problem for content distribution is the unstable set of servers for public-server games. In the content distribution problem, the set of distribution nodes is typically under the provider's control or known to be stable. Our focus is on the FPS game of Counter-Strike due to its popularity, the large number of deployed servers, and the fact that we have access to an extremely popular server for the game. Some background on FPS games is helpful for this discussion. The typical FPS game is a user controlling an avatar on a given map (such as a building), exploring the map to find enemy player and engaging in a gun shoot-out. The games are small-scale, with typically under twenty people playing on a given public server. For FPS games such as Counter-strike, latency is a strong determinant in user satisfaction [5, 44, 42, 43]. Because of this, it is imperative that clients can easily find and connect to servers that are close to them. However latency is not a client's only concern; they may also care about the map, server

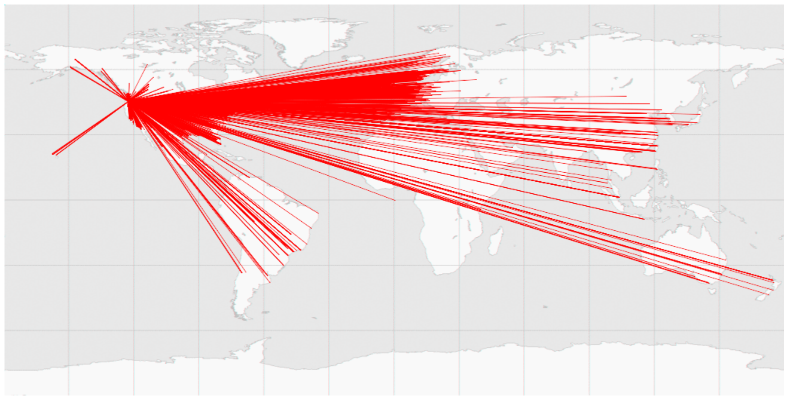


Figure 3.2: Overflow connections on `cs.mshmro.com` 6/17/03-6/19/03

rules, or other individual server characteristics. These client concerns make the server selection problem for public-server games unique as compared to content distribution or anycast. The server selection problem for a client is therefore to find a server that meets the client's needs (latency and otherwise) in a scalable fashion. In order to address this problem, we have designed and implemented a centralized geographic redirection service to connect players and servers.

To demonstrate the efficacy of our approach, we modified the Counter-Strike game server for an extremely popular gaming destination, `cs.mshmro.com`, so that it transparently redirects players based on their geographic locations. For reasons that are not completely clear, some public servers are enormously popular while others languish. Some explanations include network positioning, word of mouth reputation, server rules, or something biased in the black-box algorithm used by Half-Life's "Quick Start" button. `cs.mshmro.com` is forced to turn away over 2000

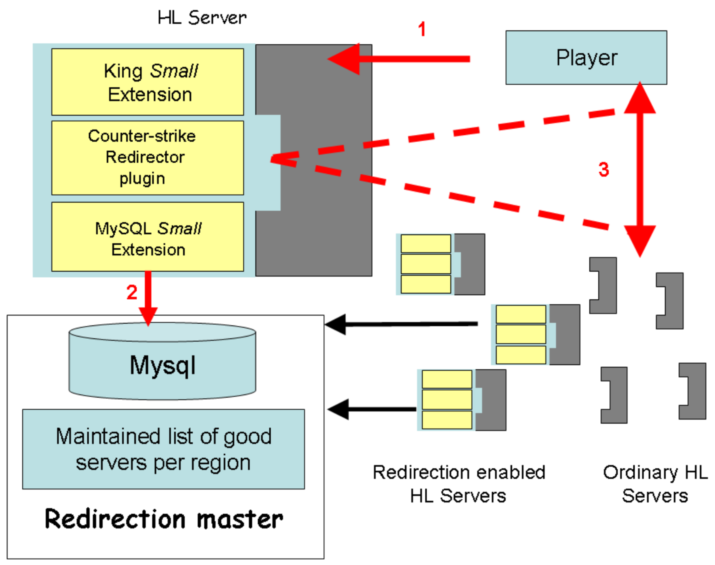


Figure 3.3: Overview of redirection service architecture

people per day simply due to being full. Figure 3.2 shows a map of the world with a line drawn between `cs.mshmro.com` and every client who tried to connect but found the server full during a 48 hour period (there are 5400 lines). Ideally, full game servers would be able to redirect potential clients to other servers, perhaps to servers even better suited to the client needs.

3.2.2 Methodology

Our service provides a scalable, centralized redirection architecture by which servers with high load can, instead of dropping clients when full, redirect the clients to a server that meets client needs, as well as improving the overall connectivity of the network. The management of where these clients go is handled by a redirection master, a service that can be located anywhere on the Internet. We envision the application of this service to any public-server game, such as Quake, Half-Life, or Neverwinter Nights, as well as any network service that runs on a large number of geographically distributed servers (i.e. a geographic “anycast”).

Figure 3.3 provides an overview of our architecture, and the three-step process players go through to be directed to a good service. In (1), players connect to a Counter-strike server running our redirection plugin. In (2), the Counter-strike server contacts the redirection master, submitting the player’s IP address, and receiving a good server for that player in return. In (3), the Counter-strike server reconnects the player to the good server.

In order to match clients with servers, geographic positioning information is used to locate a server that is in the same region (say, continent) as the client. As seen in Figure 3.2, clients often select servers very far from home, even when there are similar servers nearby. Our geopositioning information is obtained from a commercial tool [38], which has a success rate in obtaining GPS data from client IP addresses of over 60% for Counter-Strike traffic to our server [34]. Redirection is not performed on addresses that cannot be mapped. The mapping tool itself is being updated continuously, presumably increasing its success rate and thereby allowing for broader participation in redirection over time. In addition to geographic data, we need to know which servers are currently running, which ones are usable, and where they are located.

Many current public-server games have a master server which tracks all of the game servers, for licensing purposes as well as to aid players in finding a server. A plethora of tools exist to enable gamers to contact this centralized registry and download the list of game servers. We use QStat [73], which can also contact a server and retrieve its characteristics. We define a server as “good” when it is up, not full, and has game rules that match the game rules we define as important. For this study, we only considered one game rule to be critical, which was the rule allowing play without a password. Surprisingly, around 40% of all randomly selected Counter-Strike servers are protected with a private password. Our goal is to redirect players who cannot play on a server to a likely candidate server in their own region of the world. In addition to a ruleset we also partition the world into a distinct set of geographic regions, with the goal being to maintain a list of valid servers for each such region. Our service can allow for more game rules to be considered at the cost of maintaining more valid servers per region.

The redirection master periodically retrieves a listing of all registered servers and performs geographic lookups on each of them, categorizing each server into a particular region of the world, and storing this information in a database. This is done daily, and serves to capture a rough view of the available registered servers for the day. This process can take up to an hour.

In addition to being time consuming, polling the servers once a day gives us little assurance that these servers are still up several hours later and tells us nothing about how full the server is at any given moment. To address these issues, we frequently (every 5 minutes) select a subset of servers from our large list to be redirect target servers, 5 per region, and verify them to be good. These are the servers to which clients are redirected to when our server is full. The frequent verification of this list gives us some measure of confidence that the servers are still good, although it does not establish any guarantees.

Reducing the list of servers to just a few select redirect servers updated every few minutes has several benefits. First, it reduces the amount of polling and processing required by the redirect master. Second, it allows the service to fill up the redirect servers, giving players on those servers close to the same game play that they would have experienced on the original server. Finally, it allows for scalability of the service; by measuring the number of redirection requests in the last few time periods, provisioning of adequate redirect servers for the future can be performed.

As an added optimization we utilize an Internet tomography tool called King, which computes latency between two arbitrary IP addresses on the Internet by using recursive DNS lookups [53]. Using King we establish the approximate latency between the client and each of the redirect servers in the client's region, and direct them to the best server.

Name	# regions	Date range	Redirects
$e1$	7	6/17/03 - 6/19/03	7173
$e2$	50	10/30/03 - 11/11/03	7303

Table 3.1: Redirection experiments

Overall, our system centralizes the work of collating and continually verifying the list of redirect servers per region. In order for a game server to participate in client redirection, all that is required is the installation of a small server plug-in which contacts the redirect master and receives a redirect server given the client IP address.

3.2.3 Evaluation

In order to evaluate our scheme, we utilized two Linux servers, one running Half-Life's Counter-Strike mod on `cs.mshmro.com` and the redirect plug-in, and the another running the redirect master. The game server's redirection plug-in was written in Small [4], and the management of the redirect master was implemented as a set of perl modules storing data in a MySQL database (the same database with the geographic IP lookup).

We perform two experiments, detailed in Table 3.1. Both experiments redirect around seven thousand clients over a period of a few days, and represent an attempt to test the same redirection service with varying region size. To illustrate the redirection process, Figure 3.4 graphically shows the locations to which refused

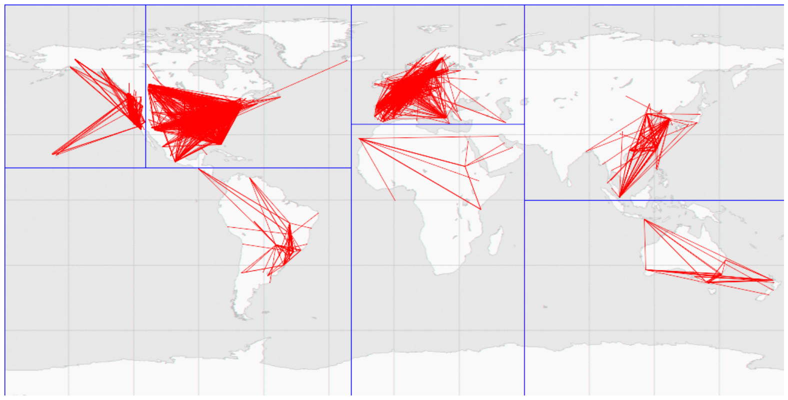


Figure 3.4: Redirected connections on cs.mshmro.com 6/16/03-6/17/03

players were routed over experiment $e1$, as well as explicitly delineating the regions. To establish the relative merits of our redirector we evaluate it on two criteria: distance and latency.

Distance

One metric used to determine the effectiveness of this redirection service is the savings in overall network efficiency: kilobit-miles. As shown in [32], a typical Counter-Strike player utilizes 56kbps of network traffic. If a player was playing over a two mile link, and was rerouted to a server one mile closer, the network would save 56 kilobits per second of traffic over a one mile link, or 56 kilobit-miles. Since converting between kilobit-miles and miles is a constant conversion for Counter-Strike traffic, we simply measure miles saved. It is important to note that these are extra miles of links which gameplay traffic no longer has to traverse, and furthermore these savings have a dimension in time for as long as the player plays. While miles saved and network latency saved are only roughly related, it is expected that with the continued build-out of Internet infrastructure (in terms of exchange points and last-mile links), the correlation will increase.

By logging each client's location, c, and the location of the server, s, to which they were redirected, and with the knowledge of the server location at o, the number of miles saved can be calculated by computing the great circle distance between c and o, and subtracting the distance between c and s. Using the radius of the Earth r, we compute the geometric formula for the distance between two points $(\delta_1, \phi_1, \delta_2, \phi_2)$ on the globe as

$$d(\delta_1, \phi_1, \delta_2, \phi_2) = 2rsin^{-1}\sqrt{\begin{array}{l} sin^2(\frac{\delta_1-\delta_2}{2})+ \\ cos\delta_1 cos\delta_2 sin^2(\frac{\phi_1-\phi_2}{2}) \end{array}}$$

During experiment $e1$, the redirector found good servers for 7173 clients, saving over 15.5 million miles, or an average of 2203 miles per redirected client. Experiment $e2$ saved on average 3570 miles for its 7303 clients.

The distribution of distance saved per client over both experiments is shown in Figure 3.5. There appear to be two distinct peaks in the distribution: one between zero and four thousand kilometers, and one at 8000 kilometers. The former corresponds to redirection within the United States, and the latter is players being

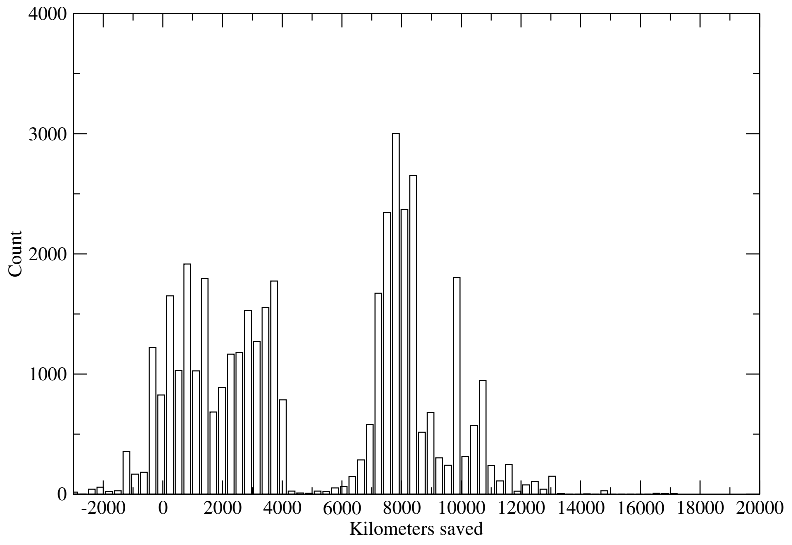


Figure 3.5: Distribution of distance savings in kilometers

redirected across the Pacific or Atlantic Ocean. Also notable is the small percentage of clients whose latency was negatively impacted by redirection. As a sanity check for our experiments, we investigate the redirects which negatively impacted client latency as measured by King in experiment $e1$. Figure 3.6 shows that these occur predominately in areas close to the server location as expected.

Latency

To better capture the relationship between geographic miles saved and network latency, we again use the King tool [53]. For simplicity, we focus on experiment $e1$ where the regions (continents) have well-known names. For this experiment, King was able to determine the latency between 61% of the redirected clients and servers. Using King we compute the latency between each redirected player and

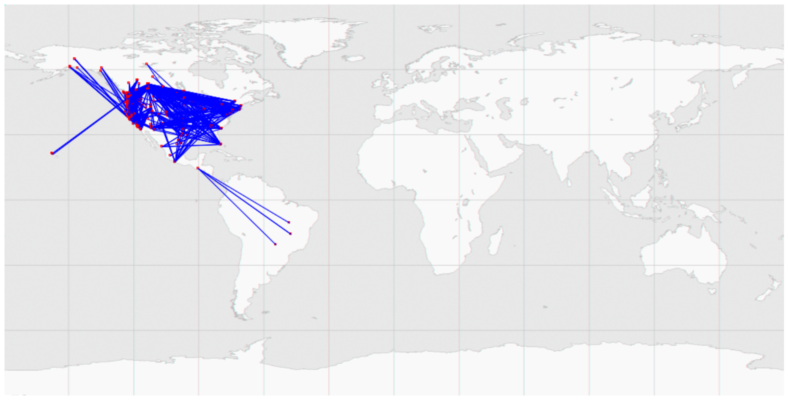


Figure 3.6: Bad redirects for experiments $e1$ and $e2$

Continent	Distance Sample Size	Latency Sample Size	Latency (ms)	Distance (mi)
Europe	1914	787	46.97	4456.18
N. America (West)	1689	1215	-10.01	-1.42
N. America (East)	2614	1901	13.38	1017.07
Asia	574	223	75.23	4981.10
Australia	77	69	-15.77	5889.95
S. America	180	107	153.14	5421.81
ALL	7069	4318	19.39	2203.22

Table 3.2: Average latency and distance reduction for redirected players for experiment $e1$

our server and the latency between the player and the server they were redirected to. Table 3.2 summarizes the latency and distance reductions of redirected players for the server, as well as the available sample sizes.

The latency savings per player are modest to small for players in nearby regions and relatively large for players in faraway continents. While the bulk of clients are affected favorably by redirection, a fraction are very adversely affected. The majority of clients come from Europe and North America. As the table shows, the clients in regions furthest away from the server (Europe, Asia) benefit most from redirection, whereas clients in the same region as the server receive little (if any) latency savings. Indeed, while European clients save on average 47 ms of latency and 4456 miles, clients in North America West frequently get redirected to servers (slightly) further away than our own.

The correlation between distance and time saved is inexact. Typically, as millions of meters are saved, so are tens of milliseconds. In Figure 3.7 we show a scatter plot of distance versus latency, and the linear regression line. The graph shows the continent gap between 4 and 6 million meters and a large number of outlaying data points. We attribute these outliers to clients who are geographically distant but extremely well connected to other continents, errors from the King tool, or errors from our geographic database.

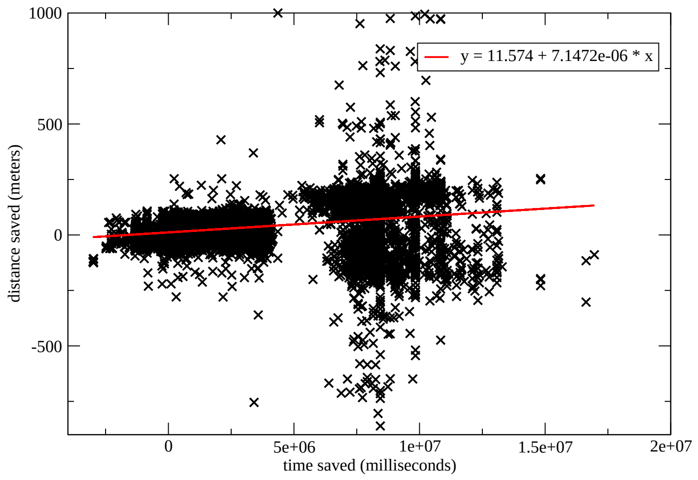


Figure 3.7: Plot of distance versus latency for both experiments

Analysis

Next we contrast the algorithmic complexity the status quo client connection process versus of our redirection scheme in terms of network connections over time. The simpler case is the status quo: if we let h be time in hours, c the number of clients connecting per hour, and s be the total number of servers, then the status quo has chs connections from clients probing servers, as each client must in the worst case probe all servers. Once a good server is located, the client must connect, adding another ch network connections, for a total of $chs + ch$, or $O(csh)$.

For the redirection service we consider service maintenance and client connections separately. Our service must maintain a list of good servers per region. There are two portions to this maintenance: daily scanning, and continual updates. Daily scanning involves $\frac{sh}{24}$ probes. For continual updates, let the number of probes per hour per region be u and the number of regions r. Then the total maintenance probes is uhr. What is the value of u? In the worst case, we must probe all the servers in each region to find good servers, and $u = \frac{ks}{r}$ for some constant k. In our implemented system $k = 12$, as regions were maintained 12 times an hour. Thus the worst-case hourly maintenance portion of the system requires ksh probes.

The second source of network probes comes from clients connecting to the service. They make one network connection, the service makes j connections on their behalf where j is the constant number of servers maintained per region, and they are sent to the appropriate server for a total of $j + 3$ connections. Therefore client connections amount to $ch(j+3)$ connections. This gives a total worst-case complexity for the service of $O(\frac{sh}{24}+ksh+chj+3ch)$, or, rearranging terms, $O((\frac{1}{24}+k)sh+(3+j)ch)$ which is equal to $O(sh + ch)$. This is a substantial improvement over $O(csh)$ for large numbers of hours, servers, and clients.

Moreover, the expected number of connections is substantially smaller. While the worst-case number of maintenance probes per hour per region was $\frac{ks}{r}$, the observed expected case is $2kj$: on average 2 probes per good server. This is due to the large number of empty servers and server stability. This brings the expected total number of connections to $O(\frac{sh}{24}+2kj+chj+3ch)$. While this does not impact the algorithmic complexity (still $O(sh + ch)$), it removes the constant factor of k from the server term, an improvement of hundreds of thousands of connections per hour in the real world.

3.2.4 Conclusion

We have presented a redirection service for game servers that improves the public server experience for clients by addressing the problem of finding a good server without incurring the time or efficiency costs of probing all available public servers. Our technique is to centralize the polling process, divide the world into geographic regions and direct clients to nearby servers with low latency, as established with a third-party latency measurement tool, King. Unique to this service is the ability to connect clients to servers that meet specific server rule criteria.

The redirection service has certain limitations. It groups players and servers by geographic region, which means that (1) players from different regions will never play together, and (2) the service may perform poorly for regions where network latency and geographic distance between clients and servers are especially negatively correlated. A limitation of our evaluation is that we also rely on the King tool heavily in verifying the benefits of redirection. Our results using King with regards to likelihood of a valid lookup, as well as consistency of latency from lookup to lookup are somewhat contradictory with the results shown in the King paper.

We would like to perform a more rigorous analysis of the usefulness of King with respect to the gaming population.

3.3 Public Server Games and Persistent Content

3.3.1 Introduction

In Chapter 2 we have discussed the challenge of hosting an on-line game in terms of resource provisioning, and in this chapter we have presented a promising alternative to client/server hosting for GameCorp: the public server model, where the users provide hosting resources for the game. Two drawbacks of the public server architecture are the difficulty of getting clients and servers connected (addressed in the previous section) and the typically less "massive" experience of playing the game on a small server. This second limitation is closely tied to the issue of persistent content sharing between servers, as a large number of servers trusting each other and acting together as a game host can present the illusion of massive world. Indeed this is typically how commercial MMO's are architected. In this section we present an architecture for hosting a MMO game with persistent content in a public server setting.

A public server system has two benefits that we would like to bring to GameCorp. The first is that it shifts some of the burden of hosting onto the game players. The second is user-generated content. In addition to the challenge of hosting a game,

Game	User Content	Architecture	Persistent
Typical MMO		Client-server	X
Typical FPS	X	Public server	
Typical RTS		Peer-to-peer	
Second Life	X	Client-server	X
Neverwinter Nights	X	Public server	
PS MMO	X	Public server	X

Table 3.3: Game architectures

persistent on-line games require an enormous amount of content generation to keep subscribers playing. If the game is lacking in novel activities or progression, avid gamers will become bored and unsubscribe. In stark contrast to a linear single-player game that can be mastered in dozens of hours, publishers would like MMO players to be able to enjoy their game indefinitely, regardless of how much time they put into the game. Thus, continuing content development is critical to the longevity of a MMO. Unfortunately, novel content is typically developed by the publisher at a slower rate than it can be played through, resulting in bored gamers. Some games, such as Second Life and public-server games like Half-life are designed to allow user-generated content or mods that extend the lifetime of the game considerably.

Table 3.3 summarizes the state of on-line gaming with respect to user-generated content, architecture and persistence. Only Second Life allows for user generated content along with a persistent world, but it takes place in a client-server architecture. Our solution to the hosting and content creation challenges posed by MMO's is to move to a public-server architecture and allow users to generate content. We call this architecture Public Server MMO (PSMMO). The intended goal of PSMMO is to inexpensively scale hosting resources and content generation with the number of users playing the game. This PSMMO architecture introduces some fundamental challenges that this section addresses: (1) trust and authentication (2) content creation and (3) content distribution and exchange. We address these issues using a combination of incentives and public-key cryptography.

The rest of the section is outlined as follows: In Section 3.3.2 we discuss work related to ours, in Section 3.3.3 we discuss user resources, in Section 3.3.4 we present our PSMMO design, and in Section 3.3.5 and Section 3.3.6 we share our conclusions.

3.3.2 Related work

A number of solutions have been proposed in recent years to address the problem of hosting MMO's. One solution is to dynamically host games in an on-demand fashion and take advantage of economies of scale and differences in gaming popularity in a centralized or grid-based fashion [46, 81]. We believe these efforts to be synergistic with our effort to harness user resources.

Another approach is to use clients to form a P2P network responsible for gameplay computation as well as storage [51, 45, 55]. Our solution is not P2P, but rather public server, which incurs certain trade-offs. While P2P networks are an attractive solution due to their scalability properties, they typically introduce increased latency for multi-hop tasks such as peer routing. Additionally some game players are unable or less able to participate in a P2P network due to firewalls and discrepancies between upload and download speeds in home networks.

3.3.3 User Resources

Quantity

The merits of our design rest on the willingness of gamers to contribute their resources and creative energy to the betterment of a compelling game. There is some empirical evidence that this willingness exists for popular public server games. Figure 3.8 shows the cumulative distribution function (CDF) of the percentage full of all Half-life 2 servers as polled every 10 minutes from 05/24/2006 to 05/29/2006. This figure shows that even though Half-life 2 is an enormously popular game with over 100,000 concurrent players at any moment, the user-contributed server resources are 70% idle. This represents over 18,000 idle Half-life servers.

In addition to being willing to contribute server resources, players are also keen to contribute game content. In a public server game the players typically have access

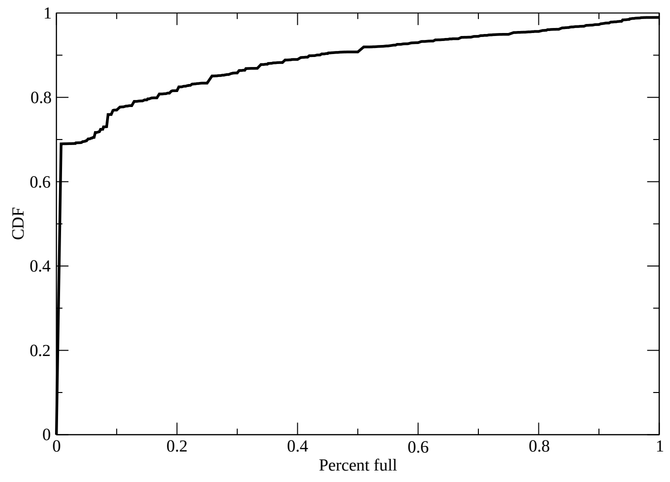


Figure 3.8: CDF of public server utilization for Half-life 2. 70% of all servers are empty.

to the art for the game in addition to the server binaries, and so publishers often allow players to easily modify and extend the art and gameplay. As examples of user interest in content creation, the developers for Half-life and Neverwinter Nights have released 6 and 7 official gameplay additions and variants, respectively. Their user bases however have created at least 492 and 4372 gameplay additions [89, 50]. As another example, Second Life is a MMO that has minimal developer content; most of the gameplay is emergent behavior generated by user behavior and user-driven content creation. Linden Labs estimates that of the 80,000 aggregate hours per day users spend in Second Life, 25% of user time is spent on content creation [67]. One concern is that users generate too much content and that it may be challenging to locate the high-quality additions. While a content rating system would help to solve this problem, we believe that the social nature of persistent on-line games will cause gamers to gravitate towards compelling content without any publisher-sponsored rating system.

Quality

User resources may be plentiful but their overall utility to game popularity is less easily quantified. Regarding user hosting, the computational requirements for hosting MMOs are not well known due to the closed nature of successful industry games. While users are willing to contribute servers for games with dozens of clients such as Neverwinter Nights, it is unknown what sorts of gameplay sacrifices would be required to allow user machines to host compelling MMO gameplay. We do not address this issue in this paper and instead assume that any desired gameplay can be hosted in some way by user-contributed server resources. We also do not address public server reliability or response time fairness to clients, and instead assume that whatever service a given client requires is replicated in depth, as is the case for Half-life players searching for popular varieties of gameplay.

Regarding quality of content we note that user content can be extremely popular: the Counter-strike and Capture the Flag modifications for Half-life and Quake have been more successful than any publisher-generated content.

As successful on-line games can be profitable and very important to users, there are certain legal and ethical challenges inherent in harnessing user resources for profit, such as intellectual property rights and server liabilities. We believe these issues are important, but we also believe that users enjoy contributing building blocks to their gaming world, and assume that some legal or monetary resolution for these issues can be achieved that enables the harnessing of user resources for scalable persistent worlds.

3.3.4 Design

We preface our design discussion with a more in-depth description of the tasks and motivation involved in playing a MMO. The generic case of MMO gameplay involves controlling a single avatar with a set of abilities and performing tasks in world that advance the power, possessions and abilities of the character. These gameplay tasks can vary widely based on the genre of the game, from rescuing hostages to competitive fighting to killing monsters. Successful completion of the tasks generates rewards that slowly advance the state of the character. While a new character begins the game with only a few abilities or possessions, as a reward for hundreds or thousands of hours of playing the game the character typically has dozens of abilities and hundreds of possessions.

From the perspective of a gameplay host, these various aspects of persistence (abilities, possessions, and levels) are all alike in that they grant the player additional gameplay effects. Because of this, and in order to maintain generality, we refer to any persistent advancement a character can achieve as that player receiving *loot*. The substantial investment in time played and tasks completed typically means that players are very attached to their loot, and very interested in how to get better loot.

Because acquisition of loot is a primary motivator for persistent on-line games, we choose to focus our design on loot instead of gameplay. This is not to downplay the importance of gameplay, but rather to allow the publisher and community complete freedom to create whatever sort of game they would like and maintain a valid reward structure. As important as loot is to individual players, the assurance that a player's loot was earned fairly is of critical importance to the community's confidence in the virtual economy and the lifetime of the game.

The three key participants in our design are the clients, public servers, and publisher. Figure 3.9 shows each of the three participants and their general role in the architecture: the publisher handles player authentication, billing, global gameplay functions such as chat, and loot distribution to servers, who handle gameplay interactions with clients. In the rest of this section we present our architecture by focusing on the three challenges it strives to meet: authentication, persistent content, and trading.

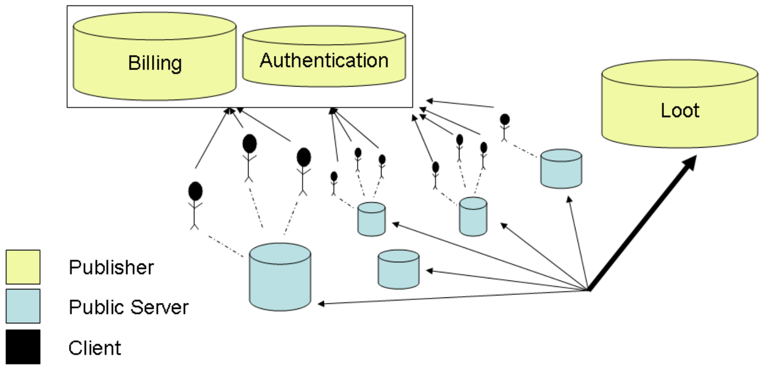


Figure 3.9: Participants in PS MMO

Authentication

One central challenge in a public server MMO is authentication and trust. Since clients are paying a subscription, the loot server must be able to authenticate clients. All participants must be able to verify loot as authentic and trust that a given client is allowed to possess it. To meet these needs, we generate the following pairs of keys: each client i keeps a private key cl_priv_i, with public key cl_pub_i stored by the publisher. The publisher advertises a master loot key $loot\_pub$ but keeps $loot\_priv$ secret. Finally, the publisher keeps a key pair $bind\_pub_i, bind\_priv_i$ for each client i, advertising the public bind key. Generally, cl_priv_i is used to authenticate the client to the authentication server, $loot\_priv$ to sign loot, and $bind\_priv_i$ to bind loot to a given player.

A different but related authentication problem arises from our incentive-based loot distribution model, which grants loot based on the number of player-minutes accumulated per server. The loot server needs to verify that player-minutes are not being granted to a server without a player actually present on the server. One could imagine, for instance, a player and public server colluding to accumulate player-minutes every minute of the day even when the player was away from the computer. We authenticate player-minutes with the use of periodic CAPTCHA tests that are known to be challenging for computers but easy for humans [88]. While CAPTCHA design is outside the scope of this work, we believe the goal of the tests should have an additional component aside from differentiating humans and computers: differentiating gamers from other humans. A game world has a unique environment and set of rules; it should be relatively easy to place some of this context into the CAPTCHA, for example with an image from the game. By binding the player to the domain of our game we can deter work-arounds such as CAPTCHA farms or CAPTCHA redirects.

Persistent Content

A second challenge for a public server MMO is persistent content creation and security. We first discuss persistent content creation and distribution. Our design is that the user community and publisher create content of two sorts: gameplay content such as the environment with which the player interacts and the available actions, and persistent content such as items or abilities that are intrinsic to the player and modify their appearance and gameplay. The community and publisher have complete freedom to design gameplay content, but persistent content must be reviewed and balanced by the publisher before being admitted to the game.

Persistent content is issued to the public servers according to accumulated authenticated player minutes logged at the server. The public server receiving loot dispenses the loot according to its gameplay rules, which could vary widely. For example the gameplay could require players to compete against each other in a tournament, with all the combined authenticated player minutes going towards a prize for the winner. The server could distribute loot according to the defeat of computer-controlled scripted encounters, or randomly, or only to certain people. While the potential for abuse is clear in such a system, we believe that as users can vote with their attention for different servers, they will tend to gravitate towards fair servers with compelling gameplay. This is the case for other public server communities such as Half-life where servers that allow cheating are eventually abandoned.

The security of distributed content is ensured with the bind keys $(bind\_priv_i, bind\_pub_i)$ and the master loot keys $(loot\_pub, loot\_priv)$. A given piece of loot is signed with $loot\_priv$ and signed again with $bind\_priv_i$ for client i, and then given to the client.

The client can present this loot at any public server or to any client for verification. No other client j can forge loot without knowledge of $loot\_priv$ or $bind\_priv_j$. This enables each client to store its own loot locally. One drawback of this design is that a player can never trade or lose an item. We will describe a way to relax this constraint later.

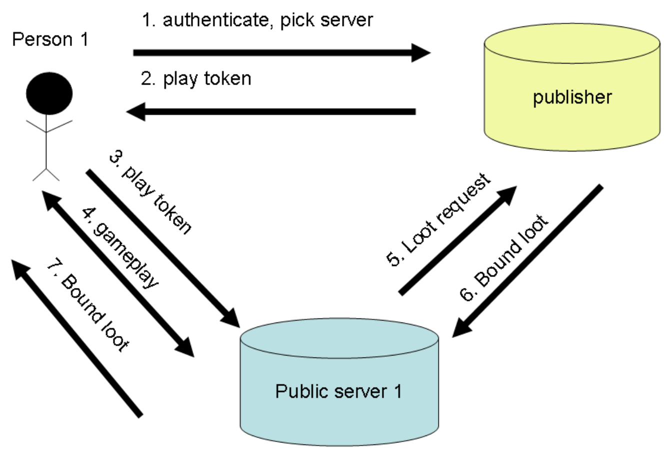


Figure 3.10: Player interactions with public server and publisher during normal gameplay

Figure 3.10 shows an overview of the player interactions with the publisher and public server during normal gameplay. The player is initially authenticated with the publisher as a subscribed gamer and receives a play token good for play on the desired public server for a certain period of time. That public server's credit is recorded by the publisher. Using the play token, the player authenticates itself to the public server and plays the game. When loot is distributed, the public server requests the item from the publisher, who signs it and debits the public server's account. The public server then gives the player the item.

Trading

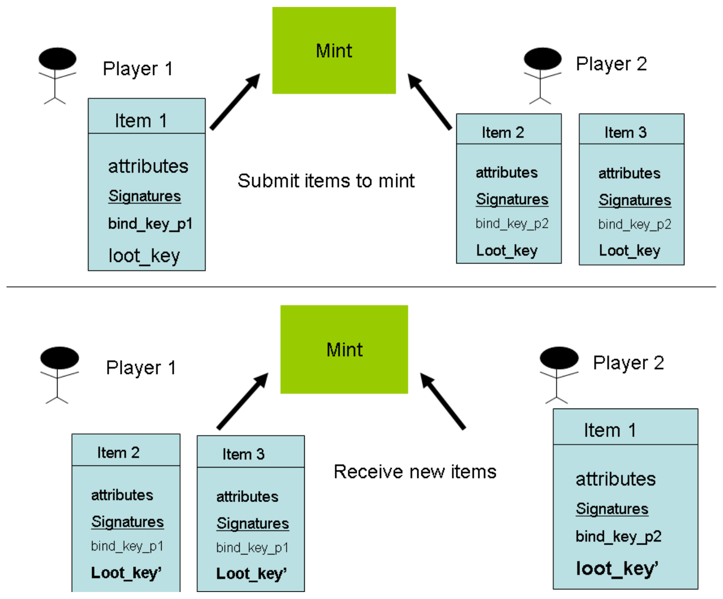


Figure 3.11: Example of how trading alters signatures on items

The trading of items is a backbone of many persistent worlds, but prohibited in our design so far. We would like to allow trading in a way that does not permit item duplication. The core of the problem is that once an item is traded between parties it should not be possessed by both parties. As PSMMO uses cryptographic signatures to indicate item authenticity and ownership, we need a way to invalidate ownership. In public key cryptography this problem is called certificate revocation and the typical solution is a certificate revocation list (CRL): a list kept by an authority listing invalid certificates. For PSMMO, this solution requires (1) frequently checking the CRL for freshness and (2) maintaining a CRL for all items for all players over the history of the game. We propose to avoid the scalability problems associated with such a list by establishing a globally synchronized trading session in which all items in the game are re-issued according to a new $loot\_priv, loot\_pub$ master loot key pair. The publisher would host a global market for persistent items, with players able to bid on items or establish trades in the time period preceding the trading session. No actual trading would occur however, until the trading session, during which each player would have his items re-issued according to whatever trades had been agreed upon. Figure 3.11 shows a pair of players with their items before and after the trading session, where they each possess the other's items. As the loot key has changed, the old items are no longer valid.

Our scheme requires the periodic re-signing of all items in the game to a new master loot key. What is the computational cost of that task? To scale trading with the number of subscribers, the publisher must possess resources capable of performing the signing task for all users in a given window. The free cryptography library $crypto++$ reports signature and signature verification times of 4.75ms and 0.18ms respectively for the RSA1024 public-key cryptography scheme on a

2.1GHz Pentium 4 [69]. Assuming each user has 100 items, and each user's items are processed (verified and signed) once, the Pentium 4 would then process 90,947 users in a 12 hour period. This gives a simplistic overview of the cost of the signing task, but it should be noted that re-issuing each item can be performed in parallel and can be performed lazily upon authentication. Computation time for signing can be further reduced with dedicated hardware support.

3.3.5 Discussion

As our design can accommodate a variety of persistent games, it may be helpful to present an example game, called *Smite*. In this imaginary game each player controls a fantasy character in a universe whose gameplay consists primarily of smiting monsters, collecting treasure and learning new abilities. In the beginning of the game, players have a default set of abilities that let them smite the lowest level monsters. The publisher has set up a loot server, authentication server, and possibly some other globally servers to address global functions such as chat, trade or movement between servers. Users have set up *Smite* gameplay servers of all sorts representing different parts of the world or different activities in the world, and catering to different levels of player. In order to advance in *Smite*, a player must convince a server to issue loot to the player, by completing gameplay tasks on the server. The server may choose to guarantee that every monster on the server drops loot when killed; in this case the server will only display a monster when it has enough authenticated player minutes to give loot to the monster as well. Or the server may choose to spawn monsters and determine if they have loot or not when they are defeated. Certain servers may grant loot to players who pass a gameplay challenge such as platform hopping, or to players who beat other players in a head-to-head challenge such as chess. The restrictions imposed on user created *Smite* servers are that they can only issue approved loot, and they can only issue loot in accordance to cost in player-minutes.

The focus of our design is on working persistent content into the public server model in a way that allows authentication of players and items, and the intended benefit is decreased hosting costs and content creation costs. A first limitation of our model is the use of clients to store persistent data. In this situation, the burden of backups, sharing and synchronizing data between different locations is on the client. We see two other primary sources of limitation: abuses of the unmonitored channel between clients and public servers, and scalability of the publisher's central authentication role.

Regarding hosting scalability, it should be noted that the publisher's hosting and content costs do increase with the number of users as the publisher must orchestrate whatever global gameplay tasks exist, such as the trading market or global chat functions. Similarly, the content balance review process that controls loot admission into the world becomes more laborious with the number of users. Thus one limitation of our design is that we merely reduce the load on the publisher. Regarding client and public server collusion, we believe our system incentives work against widespread abuse. The incentives in our system are (1) people who contribute servers want them to be utilized by players and (2) players want to acquire loot and to have fun. As an example of abuse, a hacked public server could have special rules granting the server administrators powers or special non-authentic loot. However a persistent social world such as a MMO comes implicit with a social reputation system, and in the long run we believe players will tend to avoid cheating servers. Similarly, servers and clients could collude to receive loot without performing any meaningful gameplay tasks (i.e. clients log into an empty room, answer periodic authentication queries, and eventually leave with loot), but we believe players will instead gravitate towards servers with compelling content. As a final example, servers could allow non-authenticated players to play on their server, although they would lose the incentive of gaining credit with the loot server. This leads us to one form of abuse that players could have incentives to gravitate towards: a free service that was not subscription based, but rather ran on user-contributed hardware and simply copied the approved content from the paid service game as it came out. As the bulk of the artistic content is available to clients and the server binaries are also available in our model, we do not believe there is a simple solution to this problem. However as these copycat publishers become popular they also become easier to locate and shut down legally. Furthermore, the cost of taking the publisher's role is not trivial even if the copycats do not have to perform the content balance.

3.3.6 Conclusion

Current MMOs are extremely popular but are costly to host and require an enormous amount of ongoing content creation to keep subscribers happy. The public server architecture offers an alternative that harnesses user resources to host and author content. We focus our design on the management of persistent content (loot) across public servers. The challenges of the public server architecture we address are authentication, persistent content creation and distribution, and game balance.

Our design uses a central authority (the publisher) and is incentive-based. Players want better items, abilities, and other forms of persistent upgrades to their character, while servers want popularity and to distribute valuable upgrades. The key mechanism for both of these incentives is that loot is distributed from the publisher to user-run public servers based on accumulated player-minutes logged at that server. Once issued, loot is stored on the client and cannot be forged as it is signed with cryptographic keys.

Our incentive-based design has certain limitations, such as requiring clients to store and backup their own persistent content and allowing for collusion between players and servers for short-term benefits. More broadly, our design ignores the substantial difference between the high level of performance and reliability of modern centralized MMOs and the more modest hosting characteristics of a single public server. We assume the overabundance of public servers can be used to form a similar high reliability and performance system for MMO gameplay. Future work would provide a design for this, in addition to meeting other challenges such as a system for exchanging players between servers according to game rules, a reputation system for public servers, and a more elegant solution for trading authentic content other than completely re-issuing all content.

Within these limitations, the PSMMO model is a cost-effective architecture for GameCorp to consider when launching the next persistent on-line game, as it has the advantage of harnessing user resources to effectively scale. Games are becoming more popular, resource intensive, and expensive to author and maintain; we believe user-generated content and user hosting will allow games to flourish in the years to come.

Chapter 4

Cheating in On-line Games

4.1 Introduction

One issue GameCorp must concern itself with is cheating in on-line games. Cheating in games of all sorts has existed as long as games have had rules. Cheating is not always a bad thing. By violating the rules of a game, cheaters can add a handicap to an unfair match-up, test the observational power of their opponents, or spice up an otherwise dull experience. In society at large, cheating at a game with little at stake (such as solitaire) is generally viewed as more acceptable than cheating at games with a great deal at stake (such as casino gambling or political elections).

Cheating in video games is also well-grounded in tradition. Most single-player games are released with "cheat codes" that allow players to bypass difficult content or gain abilities that the game was not originally designed around, or simply add interest to a game that has been otherwise fully explored. However, cheating in computer games has become more of an issue in the past decade as the Internet has enabled people in disparate locations to compete or cooperate together in the same game. As opponents have become more anonymous and remote, the focus on competitive gameplay over camaraderie or relaxation has increased. Indeed, there now exist tournaments with thousands of dollars at stake for competitive players of on-line games.

Recent years have seen an increased interest in the research community in addressing the problem of cheating in on-line games [91, 6, 9, 20, 56]. Several differing taxonomies of cheating have been proposed to categorize the large body of existing cheats [61, 54, 24, 92]. Some research has addressed the issue of ordering events in a peer-to-peer game [37] as well as in a centralized architecture [13, 71], while other work has focused on preventing lookahead cheats and hiding secrets in client-server games [6, 9, 20, 60]. As work in this area is just beginning, many forms of cheating are left unaddressed.

We believe cheating to be an impediment to the success of any on-line game, especially if it is a game based upon competition or takes place in a persistent world where time and skill are rewarded with advancement. Individuals are motivated to write and use cheats for their own advancement and in-game rewards, while companies are motivated to write and use cheats to take advantage of the virtual markets associated with on-line games. While there is substantial money at stake in terms of sales and player subscriptions, cheating has been difficult to prevent. Indeed, cheats exist for every popular on-line game. We believe players must have a reasonably sound and cheat-free gaming experience before they will be satisfied. In this chapter we present a survey of cheats in on-line games and introduce the concept of a game's *control path*: the flow of game commands and information from the player to the server and back. We show how each client-side control path cheat can be decomposed into one of three central issues: information exposure, protocol manipulation, and game abstraction. We then address the area of information exposure in RTS games in detail, and present a *protected RTS protocol* that detects information exposure cheats in peer-to-peer games.

4.2 A Survey of Cheats

Cheating in on-line games is a major concern [23]. It is prevalent, and according to two media surveys [31, 77] is the number-one problem facing on-line games. The impact of cheating is two-fold. First, cheating negatively impacts the popularity and longevity of games. There have been no comprehensive studies about the psychological impacts of cheating on the game industry, but anecdotal evidence suggests that it greatly deters repeat customers (the honest ones suspect that everyone else is a cheater) and thereby harms industry growth. For games such as MMO's in which the legitimate players must play the game for thousands of hours to acquire certain virtual goods, the idea that anyone else could acquire those same goods but bypass the man-years of effort is infuriating to honest gamers.

Secondly, the financial impact of cheats, in many cases, comes out of the pockets of those who do not cheat. On-line games with a persistent state, in which players can advance their status over time, inevitably create virtual economies. In these economies (realized over eBay[28] or customized game-specific markets [49]), players exchange real-world currency for virtual-world items and characters. Players who cheat have an easier time acquiring items and can then sell the items to those who do not cheat.

It is important to have a clear categorization of cheats in order to structure cheating dialogue and research, as well as plan responses. While some taxonomies have been proposed by the research community, no taxonomy's categories are disjoint; that is, for each taxonomy a given category overlaps with other categories such that a single cheat may fall into two categories. For example, Yan [92] notes two categories *Exploiting a Bug or Loophole* and *Abusing the Game Procedure* that have considerable conceptual overlap, as any abuse of the game procedure can be considered a loophole with the given definitions. We believe a set of disjoint categories for cheats is an important first step in addressing cheating. Establishing a set of well-defined non-ambiguous categories for cheating in games can be considered the first step towards an automatic identification and labeling process for cheats.

Previous taxonomies have addressed all games (on-line or single player) and all forms of cheating. We choose to focus instead on what special forms of cheating exist for clients of on-line games, apart from other on-line applications such as web browsing, and apart from other games not played on-line such as dodge-ball. While cheating can occur at the client or server level (or in the middle of game traffic), we focus on client cheating. Client cheating is of greater importance as clients outnumber servers and servers for games are frequently trusted authorities who also handle client financial and authentication information. The cheats we focus on are those performed by the client on the *control path* of the game. Intuitively, the control path is the flow of game commands and information from the player to the server and back. More formally, we define a generic on-line game G as consisting of a sequence of control path messages $m_1, m_2, ..., m_n$ where m_t is chosen from the following control path message options:

1. game receives user input message ui_t
2. game writes network output message no_t
3. game receives network input message ni_t
4. game writes display output message do_t

Games are complex computer programs and they perform many more actions than these such as reading from random number generators or writing to persistent storage. Those actions, however, do not directly affect the user's perception of the game state or the state of game peers until they are used in a control path message, such as a write to the display or to the network. We limit our focus to these four actions as they define the information boundaries between the user, the game, and the network as shown in Figure 4.1 We define a *control path cheat* to be *any control path message not achievable by the publisher software and its unmodified software dependencies that gives the player an advantage.* The somewhat cumbersome definition is intended to classify cheats along the control path together regardless of what level of the software hierarchy they are performed at. For example, if a certain control path message gives a player an advantage, we wish it categorized as a cheat whether it was generated by altering the client's random number generator, or by altering the game's binary, or some operation at another layer. Given the definitions above, we can categorize on-line game cheats into four disjoint classes: *Information Exposure, Abstraction, Protocol Manipulation,* and *Out-of-path.*

At this point we first note two important limitations of our definitions. First, anything achievable by the publisher software is not classified as a control path cheat, and therefore we categorize any software bugs or design errors as out-of-path. For example, if during the course of normal gameplay players can create a duplicate of an item by dropping the item and then picking it up rapidly, we

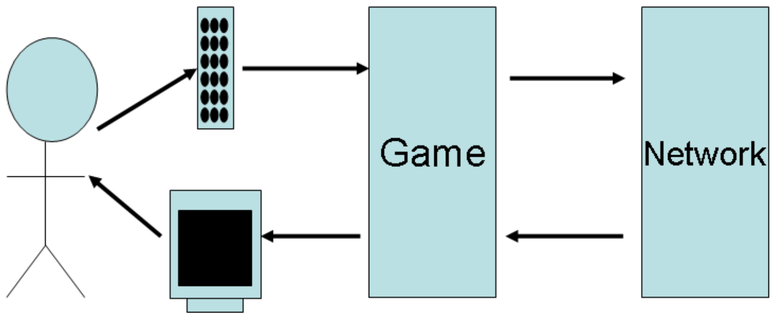


Figure 4.1: Control path for a generic on-line game

categorize this as an out-of-path design flaw. If on the other hand, an item can be duplicated via a sequence of messages that cannot be generated via the game interface, we classify this as protocol manipulation. Second, we classify effects, not original causes; for example, if a cheat writes a message to the network giving the player an advantage, we do not further classify what caused the write, which could have come from a modification of a game data file, a bug introduced into a random number generator or any other source. This means that there may be several different implementations of the same cheat.

4.2.1 Information Exposure

A cheat falls into this category when it reveals to the players information which they could not have access to, but is available on their machine. In our control path model, we define information exposure as a *display of information not available to the user*. For example, in RTS games, the "fog of war" is supposed to hide areas of the world which have not been scouted out by player-controlled units. A *maphack* cheat removes the fog of war, revealing the entire map. In FPS games, players must explore a three-dimensional map, trying to find the enemy and kill them. Part of the game is this exploration process, but various cheats such as the *wallhack* circumvent this by making all of the walls semi-transparent, revealing all players. Other examples of cheats which expose the game's supposedly secret information to players include removing "blinding" effects such as smoke or flash-bang grenades, revealing the contents of treasure chests before opening them, or revealing a deck of cards. Information exposure cheats operate at the level of the knowledge of the game state.

4.2.2 Game Abstraction

Cheats in this category are those that have abstracted the game away to something simpler, and allow the users of the cheat to play this much simpler game against their opponents (who must play the difficult game). We divide game abstraction further into two subcategories: Abstraction of Input, defined as *writing user input or gameplay messages decreasing user interactivity*, and Abstraction of Output, defined as *displaying refined information to the user to guide input.* Abstraction of input cheats are often referred to as "botting". In FPS games, an especially egregious use of this is the "aimbot" which removes the task of aiming, and causes all of the cheater's shots to hit the enemy. This is not done by altering network messages to read "I hit you" (another kind of cheat) but rather by reading the location of enemies from the game and solving a simple formula to compute the correct trajectory from the velocity and position data. A RTS example would be a "macro" which handles the management of large armies in single clicks, when it would take a fair player many clicks. An MMO example from Diablo 2 is the "get all" cheat, wherein when a monster's treasure falls on the ground, the cheat instantaneously picks up all of the loot before other players can react.

Abstraction of output cheats, on the other hand, do not perform user tasks, but rather filter game information available to the user into a more useful form. Card counting cheats or cheats that parse and remember large amounts of screen information and guide the user's inputs fall into this category, as do cheats that replace game models or textures, or highlight the best actions for users to select at a given moment in the game. Abstraction of input and output, while seemingly very different subcategories can be seen as extremely similar when the abstraction of output is for example voice synthesis guiding the user's input.

4.2.3 Protocol Cheats

These cheats use the protocols for communicating with the server and other players that were defined by the game designers, but take advantage of weaknesses in those protocols for the cheater's benefit. We define protocol cheats as *writing game protocol messages not generable by user actions to exploit weaknesses in the protocol*. For example, Diablo had a network protocol which allowed network messages of the form "I did X damage to you", and a cheat came out which simply sent messages to other players doing damage to them until they were dead, even when the players were nowhere near each other. In persistent on-line games the most common version of Protocol Cheating is "item duping", where a rare or powerful item is copied thousands of times via a sequence of messages exploiting a non-transactional game protocol. In RTS games, these cheats can take the form of manipulating messages and network timing to create extra resources or disconnect the other player from the game. A more subtle protocol cheat is the "speed hack", where a cheater speeds up the rate of sending messages to the server, enabling their avatar to move more quickly or fire more frequently than normal. In a gambling example the cheat might take the form of taking the pot before winning the hand. Protocol cheats operate at the level of the game's outputs to the server or other clients.

4.2.4 Out-of-path Cheats

Cheats fall into this category when they are not on the control path from client to network and back. Due to our definition of control path cheats, we consider any game actions performable by unmodified publisher software as intended design. This leaves many game bugs and loopholes such as game map flaws or extremely powerful combinations of game abilities not labeled as control path cheats. Even item duplication can be considered to be an out-of-path cheat if it can be performed via the unmodified publisher software. We believe these are serious issues, and ones that can yield interesting research questions and answers, but we believe the challenge is in detecting those bugs and design flaws before the game is shipped, at which point such exploits become simply part of the game. Other notable issues that are not on the control path include those found in all games and those found in all on-line applications. Cheats common to all games include collusion with other players, the dealer or game authority, leaving the game early, or lying about the results of the game. Problems common to all on-line applications include spoofing and authentication, denial of service and operating system security issues such as buffer overflows. These are important issues but their scope is beyond that of on-line games.

4.2.5 Discussion

Our three core categories are disjoint from each other in that while a set of user actions can be composed of several atomic actions within each cheating category, no atomic action can belong to two categories. This is the case because the cheating

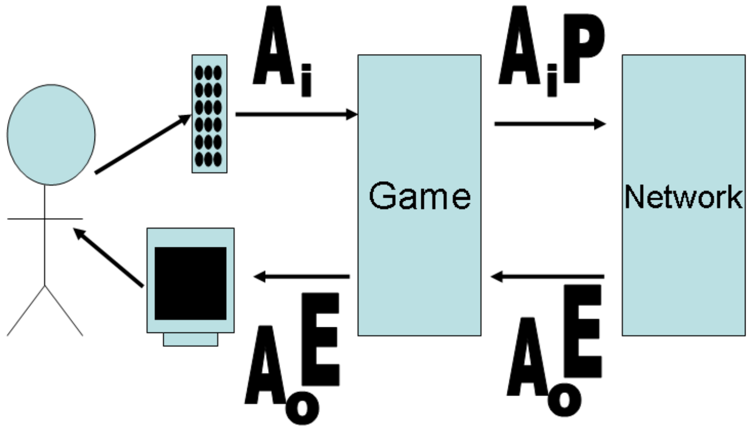


Figure 4.2: Abstraction of Input (A_I), Abstraction of Output (A_O), Protocol Manipulation (P) and Information Exposure (E) each occur at certain points in the control path

action must occur somewhere on the path from the user's interaction with the game to the game's output to the user, and the three categories operate at different points along that path. As Figure 4.2 shows, Information Exposure operates between the path from the server to the game, Game Abstraction between the user and the game, and Protocol Manipulation between the game and the server. While Abstraction and Protocol Manipulation both send messages to the server, they are distinguished in that the Protocol Manipulation messages are not normal gameplay messages.

We have designed our categories to be disjoint so that a given cheating message will only belong to a single category. We believe this to be an important property for a categorization, as it allows for the categories and terms to be used unambiguously in discussion. There are, however, some non-intuitive boundary conditions. We can

Genre	Information Exposure	Abstraction of Input	Abstraction of Output	Protocol Manipulation
FPS	wallhack	aimbot	texture replacement	rejoin hack
RTS	maphack	macro	show best action	resource creation
MMO	chest hack	botting	voice suggestions	item duping
Poker	reveal deck	botting	show odds	take pot

Table 4.1: Examples of cheats in each category for some genres of games

have, for example, a single piece of software performing the same action repeatedly but resulting in two different categories of cheats. Imagine a cheat that operates on game message no_i on the link between the game and the network and injects message no_{i+1}, an exact copy, after a delay of 50ms. If the user-generable actions for the game allow a message every 50ms, a user clicking once every 100ms will never violate game protocol, and the extra writes on the control path will be categorized as abstractions of input. If on the other hand, the user clicks more frequently, there will be violations of the user-generable actions categorized as protocol manipulations. While non-intuitive, we believe this is justified as there are two forms of cheating occurring; user interactivity is being decreased via the bot, and also the actions are being sent more quickly than allowed by the game client.

4.2.6 Dealing with Cheats

We believe the three categories of control path cheats to be important areas for GameCorp to consider in its game design, as well as for researchers to focus their efforts on. Table 4.1 shows a list of cheats and their categorization based on likely implementation. In this section we discuss what can be done to address cheats of each category.

Protocol Manipulation cheats are typically fixed by altering the faulty protocol. The research challenges in this area of cheating are in automatically finding protocol errors and proving protocol security. In recent years the research areas of programming languages [24] and security [6] have begun addressing this challenge. Information Exposure cheats can be addressed to some degree by game design; in the popular client-server architecture, the server can hide much of the information given to the client. However, performance limitations often require a certain amount of extra information to be sent to clients. For example, in FPS games the server typically sends the location of all of the players in the game instead of just the players visible from a given player's viewpoint. If network and server performance were limitless, the server could render each player's viewpoint directly and conceal global player locations. The information exposure issue is even more challenging in peer-to-peer architecture games where restricting sent information creates opportunities for other cheats. We treat this issue more fully in Section 4.3.

Abstraction cheats are problematic to prevent for a game played over a network. At some level, the outputs to a game are always visible and the inputs are always programmable, either via software or in the extreme case by a robot monitoring the computer's output devices and operating the computer's input devices. Abstraction cheats are also challenging to detect. While differentiating a computer-aided game player from a human game player sounds superficially similar to the Turing test designed to distinguish a human from a computer on the basis of ability with natural language, we do not believe on-line games can be used as an accurate Turing test. The Turing test focuses on natural language, a skill difficult to teach a computer, whereas games are designed for fun and are often not especially challenging for computers. Furthermore, while the Turing test need merely distinguish between human and computer, game abstraction cheats can achieve a middle ground where a computer and human operate together. While game abstraction may not be preventable or detectable in general, we believe that two techniques can be used to minimize game abstraction cheating: (1) reverse-Turing tests (CAPTCHAs [88, 76]) designed to test for the presence of a human and (2) trusted hardware [2]. CAPTCHAs, based on difficult artificial intelligence problems, demonstrate that the solver is not a completely automated program. Trusted hardware allows game hosts to believe that the control path from the gamer's input device to the server is tamper-proof, and that if game abstraction cheating is going on, it is occurring at the level of robotics.

4.3 Information Exposure in Peer-to-Peer Games

In Section 4.2 we introduce a top-level categorization of cheats, including the Information Exposure cheat in which cheaters gain access to information available on their computer but not intended to be revealed. In this section we consider solutions to this cheat in the context of RTS games. We first show how cheating can be detected in the simple peer-to-peer game of on-line Battleship with cryptographic bit commitment, and then demonstrate a region-based bit commitment scheme called *protected RTS* that decreases information exposure in RTS games.

4.3.1 Background on RTS games

In RTS games, each player acts as a general commanding a set of units in a battle against another player. Units gather resources, fight each other, or explore the map. Central to RTS games is the concept of the *fog of war*: player A cannot see player B's unit x unless a unit controlled by player A observes x. Each unit has a scouting radius and any enemy unit within this radius is revealed to the player. The player's vision is comprised of the union of the vision of each of his units, and everything outside of that area is in the fog of war. This work focuses on *maphacks*, a form of information exposure cheating in RTS games where one player runs a modified version of the game that eliminates the fog of war and displays the entire game state, including the other player's units and move choices, thereby gaining an extremely large advantage in the game. To our knowledge, this cheat exists for every popular peer-to-peer RTS game.

The maphack cheat is important due to the strategic underpinnings of the game. For an RTS game, players typically have a selection of many units to command, and the games are generally balanced with a "rock, paper, scissors" scheme: one kind of unit is strong against another kind, but weak against a third. Typically the games are finished when a player concedes or loses all units. In this context a maphack, by removing the fog of war for one player, confers an unfair advantage on the user.

Because of the large number of units involved per player and the financial impact of hosting client/server games, RTS games are typically played via a peer-to-peer architecture. Maphacks are prevalent in RTS games because the players exchange only user input information over the network. Each player's computer simulates the complete game individually. This technique of distributed simulation prevents many other forms of cheating by placing no trust in the other players. For example, players cannot fabricate units that they did not legally build. However distributed simulation leaves the complete game state on each computer, leaving the game open to maphacks.

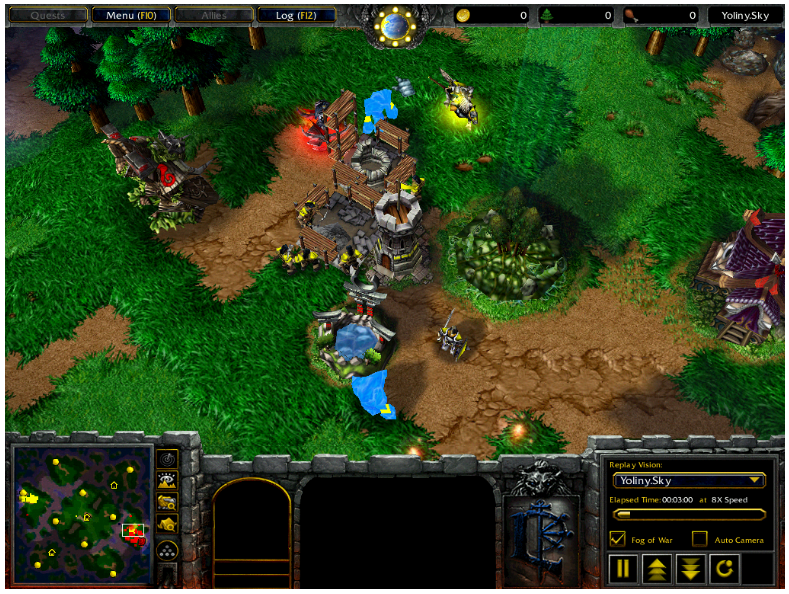


Figure 4.3: Example RTS game (Warcraft 3) interface. The map in the lower left corner shows the player's units and viewable area.

4.3.2 Related Work and Solutions

The solution to information exposure cheating is to make that information inaccessible. In the client-server architecture, the server can readily prevent information exposure by sending each client limited information. This may not always be practical, however, due to limitations in the processing power of the server or in the network performance. Another solution to information exposure is encryption. This can prevent eavesdropping, but given that the game eventually decrypts the information, cannot prevent all forms of exposure. An even harder problem is dealing with information exposure in the peer-to-peer architecture, where there is no central server to cull information between parties. In general the challenge is to develop a protocol that allows secrets that are unknown to one or both parties, but not allow either party control over the secrets. This topic is addressed in the cryptography field as the mental poker problem [82, 17, 18, 11, 93], and solutions to it rely on cryptographic primitives such as bit-commitment, symmetric encryption and zero-knowledge proofs. While these cryptographic protocols for mental poker address the sharing and hiding of secrets between peers, they are not directly applicable to the unit location secrets of map-based games such as RTS games.

In the field of networked gaming research, recent efforts to classify and categorize cheating in on-line games [54, 61, 91] discuss the problem of maphacks specifically, but not solutions. Baughman *et al* apply bit-commitment to secrets in on-line games in the context of dead-reckoned games and peer-to-peer games [6]. They introduce a scheme called AS that prevents look-ahead cheats by requiring players to commit to their moves in advance of revealing them. They also use a zero-knowledge proof to determine if two players occupy the same general region (cell) of space without revealing location information. Given the small cell size required for RTS games and the large number of units, this technique would scale exponentially and is infeasible in this context. Our work builds upon their work by using bit commitment to hide secrets, but focuses on the challenges of RTS games. Buro addresses the issue of maphacks in RTS games by presenting a client-server architecture (ORTS) to perform visibility culling for each player [9]. ORTS does not meet our goal of a peer-to-peer architecture and instead requires server resources for each game played. With hundreds of thousands of RTS games played on-line per day this solution has scalability issues we wish to avoid.

4.3.3 Protected RTS

Overview

At a high level, our scheme for securing RTS games from maphacks alters the network protocol from exchanging perfect information about what the other player is doing to exchanging information based on what region each player can see. We call the region a player can see his *viewable area*. We propose to utilize distributed simulation for actions within each player's viewable area, but to hide all other actions. We then secure these other actions from cheats by using bit-commitment and post-game verification, a technique we discuss first in the simple case of the Battleship game. Of special interest to us is the fact that in our scheme, each player knows the other player's viewable area.

We evaluate protected RTS on two criteria: *network impact* and *reduction in information exposure*. By exchanging viewable areas, players no longer know the entire game state, but they do know something about the other player's units. As detailed below, we quantify this reduction in knowledge as an increase in information uncertainty. We then evaluate protected RTS in a user-independent fashion by creating a model of a generic RTS game and simulating the increased uncertainty and information loss as we vary the unit density on the map and size of the viewable area. We demonstrate a substantial reduction in the total information available and increased uncertainty of unit position. We then perform a user trace-driven evaluation of protected RTS using actual gameplay traces to establish realistic performance characteristics for a particular RTS (Warcraft III). In this trace-driven evaluation we find that protected RTS substantially reduces information exposure and greatly increases bandwidth usage, but that the bandwidth usage still falls within a usable range.

Preventing Cheating in Battleship

A basic building-block of modern cryptography is bit commitment: a party's ability to make a choice without revealing it and then, at a later date, reveal the choice. Central to the concept is that the committer cannot change his choice after making it, and that others cannot determine the choice before it has been revealed. We first demonstrate how bit-commitment can secure the simple game of *Battleship* and then we apply it to the more complicated case of RTS games.

In Battleship, each player has five ships placed on a grid. Players take turns calling out a single grid position and telling each other whether the shot was a hit or miss. A player wins when all positions on the other's ships are hit. Without bit-commitment, Battleship is easy to cheat at, especially in an environment such as a networked game. The kind of cheating depends on whether or not you know the other player's ship positions. It is assumed that this information would not be intentionally displayed to the user, but the reality of today's cheating-heavy environment is that if the information is available on a person's computer, someone will write a program to reveal it.

If player p_1 knows where player p_2's ships are, p_1 can easily cheat by calling out a sequence of shots that hit. If, on the other hand, p_1 does not know where p_2's ships are, p_2 can cheat by telling p_1 that all shots are misses. Player p_1 would never be able to verify that player p_2 was cheating.

One simple technique to secure Battleship is to use bit commitment. Each player p_i picks a secret s_i and a set of initial ship positions sp_i. Each player then sends $h(s_i, sp_i)$ to the other player where h is a cryptographic hash function. Each player must take the other's word when they declare if each shot missed or hit, but at the end of the game, players exchange (s_i, sp_i). They can verify these against the initial hash, then verify each of the given answers as correct.

Note that the game is not secured in the sense that it is impossible to cheat, but rather each can verify that the other did not cheat. This is the approach we would like to take with RTS games as well.

Preventing Cheating in an RTS game

Our goal is to secure RTS games such that information exposure cheats will be detected. Detection of cheaters is an adequate goal for on-line games because of the high level of control held over players. Players are typically authenticated via a code on their purchased copy of the game to a central server before beginning a peer-to-peer on-line game. This gives the hosting company the ability to globally ban known cheaters from playing.

Cheating in RTS games presents more challenges than cheating in Battleship. Battleship has a few static secrets: the ship locations. RTS games have dynamic sets of units, each of which has a dynamic location. Some of the enemies secrets are supposed to be known, and some are not, based on a player's viewable region. RTS games thus represent a class of applications in which the secrets are too numerous and dynamic to secure with conventional cryptographic approaches such as bit commitment, and are linked together spatially.

Our scheme is designed to minimize network traffic while concealing as much information as possible about the enemy without permitting cheating. While the protocol generalizes to a multi-player peer-to-peer game, we confine our discussion to the simpler two player game for this example. Our scheme is as follows:

Initial exchange: Each player P_i generates an initial game state gs_i according to the game rules. Each player generates a secret s_i and sends $h(s_i, gs_i)$ along with the player's viewable area v to the other player.

In-game exchanges: For each time slice player P_i performs the following:

1. Send viewable area v
2. Receive opponent's viewable area v'
3. If current move m is in v', send it clear-text
4. Otherwise, send $h(m, s_i)$
5. If P_i's units u just entered v', send them clear-text

Post-game exchange and verification: After the game is completed, each player P_i sends s_i as well as a log of all the moves m for which they sent hashes $h(m, s_i)$. Then each player simulates the game with complete knowledge of all moves and checks the validity of each sent hash, viewable area and unit.

Using this protocol, players can lie about their viewable area, their hashed move, and what units they control. In the post-game exchange and verification, these lies will be detected. For this process we believe that the Baughman *et al* [6] definition of a *logger service* for each client to record secret moves is adequate. Verifying proper gameplay is beyond the scope of this work, but we assume it is possible given the moves, hashes, and gameplay engine.

Viewable Areas

The network impact of sending the viewable area could be very large, depending on its accuracy and representation. The two extremes of representation for a viewable area are a vectorized representation of units and radii, or a rasterized representation. As the representation more accurately depicts the location of the individual units (as in a vectorized representation), the amount of uncertainty about where the opponents units are decreases. We want to increase uncertainty and minimize bandwidth overhead, so we believe a rasterized viewable area is appropriate for RTS games.

We create our rasterized viewable areas from the actual viewable area v of area s^2 by mapping v onto a raster r of k^2 bits where bit zero indicates that a raster element is not viewable and bit one indicates that an element is viewable. For each element of our raster $r(x, y)$, let $r(x, y) = 1$ if $(\lfloor xs/k \rfloor, \lfloor ys/k \rfloor) \in v$, and $r(x, y) = 0$ otherwise. As a small raster increases uncertainty and decreases network impact, for our experiments we use a ratio of s to k of 64:1, and we vary the unit density by varying the number of units instead of changing the size of the raster.

Non-repudiation

The protocol as presented is sufficient for a player to know if a game was played fairly at the verification step. To meet the larger goal of proving to another party that cheating took place, each player must have a public and private key pair. The natural place to store the public keys would be at the authentication server for the game. To alter the protocol to allow for provable cheating each player P_i must send an additional message during the in-game exchange: a signed, dated cryptographic hash of the player's message $(v, m|h(m, s_i), u)$ for that time slice. By cryptographically signing each message sent with a player's private key, players can achieve non-repudiation; a player can prove that another player cheated if and only if cheating actually took place. This technique enables the central authentication server to ban cheaters, forcing them to buy another copy of the game to play again.

4.3.4 Evaluation

We first evaluate protected RTS in a user-independent fashion; that is, we ignore the play characteristics of users in terms of clustering of units and player interaction, and instead explore the effectiveness of protected RTS under varying sizes of maps and number of units. Specifically, we evaluate the impact of the bit-commitment scheme on three characteristics: the uncertainty it adds, the quantity of information it loses, and the added cost in bandwidth it incurs. We model our experiments after the map sizes, unit numbers and proportions used in Warcraft 3.

Uncertainty

We wish to quantify the amount of information concealed by sending viewable regions instead of unit locations. One general measure of information is Shannon's uncertainty[83], which measures the disorder and unpredictability contained in a random variable. Shannon uncertainty is defined on random variable x with n possible values over probability distribution $p(x)$ as

$$H(x) = -\sum_{i=1}^{n} p_i log(p_i) \tag{4.1}$$

In order to demonstrate the usage of this formula, imagine a networked game where a machine transmits a series of k symbols chosen from set (A, B) to the player. How much information was transmitted to the player, and how much was redundant? This depends on the probability distribution of the symbols. For example, if the player receives a series of symbols predominately A's, the player's uncertainty should be small about what the next symbol is. If we calculate the uncertainty for the distribution of symbols $(A = 0.99, B = 0.01)$, we get the small value of 0.0808. Suppose, on the other hand, the machine transmits symbols A and B with equal probability. We would like this to represent maximal uncertainty, and if we calculate the uncertainty of that equal two-symbol distribution $(A = 0.5, B = 0.5)$ we get one, the maximum uncertainty for a series of two symbols.

Shannon uncertainty represents the average number of bits required to encode the series of symbols with the most space-efficient encoding. Thus the theoretical best encoding for our $(A = 0.99, B = 0.01)$ distribution of symbols would only require an average of 0.0808 bits per symbol. One can imagine such an encoding has many short strings of bits representing long strings of A's, averaging $-log(0.99) = 0.0145$ bits, and a longer string of bits representing B requiring $-log(0.01) = 6.6439$ bits. We can see that on a per-symbol basis, receiving a B represents a larger change in uncertainty than receiving an A, and this matches the intuition that the B's contain more information. Our use of uncertainty as applied to RTS games does not relate to encoding efficiency directly, but rather to this change in uncertainty. We apply Equation 4.1 to our protected RTS scheme as follows. We represent the unit location information in the unprotected version of the game as a raster containing white pixels for unit locations, and black pixels otherwise. We represent the protected version of the game as the same raster, but replacing the units with viewable areas of a given radius. We then calculate the average uncertainty in both cases, and compare the difference. This difference measures the uncertainty we have added to the raster. As in our uncertainty examples above, we have only two symbols in our data (white and black). We evaluate the uncertainty impact of varying the number of units and the view radius of each unit as outlined in Table 4.3.4.

Experiment	Map area	View Radius	Avg. Num Units
Warcraft 3	11200^2	860	100
vary-num	640^2	49	*vary(1-100)*
vary-rad	640^2	*vary(1-100)*	6
quant	640^2	49	*vary(1-100)*
overlap	640^2	49	*vary(1-100)*

Table 4.2: Data on experiments performed to quantify uncertainty and information loss

Shannon uncertainty does not directly correlate to the amount of gameplay information hidden (for example, it does not capture the hidden unit types), but it is a useful comparison as it is completely separate from the meaning of the information transmitted. While we could model every facet of a specific RTS game in terms of information hidden or transmitted (unit level, items carried, attributes), this would bind our analysis more tightly to that specific game. Instead we focus only on the uncertainty introduced in unit location.

Figure 4.4 shows the amount of uncertainty gained as compared to the uncertainty in the maphacked version of the game. Experiment *vary-num* varies the number of units from one to 100 and leaves the radius fixed at the map proportions of Warcraft 3. Each point represents the average of 50 uncertainty calculations with x randomly distributed units. Even at one unit we see a 0.2 uncertainty gain, and this rises rapidly as we add units. At 20 units we gain the most uncertainty, and past that we see some noise in the signal as a result of the increased probability of unit regions overlapping. By 100 units we see that uncertainty has decreased back to its initially low starting conditions; at that point, the viewable areas cover nearly the entire map and the transmitted information is again low.

For experiment *vary-rad* we vary the radius of the units by an order of magnitude around the Warcraft 3 radius, while keeping the number of units proportional with the map size. Figure 4.5 shows a substantial initial uncertainty gain initially even at a radius of one, with uncertainty leveling off slowly as the radius exceeds 100. We conclude that the specific radius per unit is less important than the number of units in the game in increasing uncertainty.

The uncertainty gain from unit radius and quantization is considerable. Our results indicate that the peak uncertainty of our scheme falls within the bounds of normal gameplay in terms of unit numbers and viewing radius.

Information Loss

We also present a second metric for evaluating the scheme: information loss. Whereas uncertainty quantifies the likelihood of guessing the color of a pixel, information loss quantifies the number of data points that are deleted. For example, when quantizing a large map into a two by two black and white grid, it is not possible to represent more than four points, no matter how many points existed initially. The lost information in our scheme comes from two sources: the dispersal

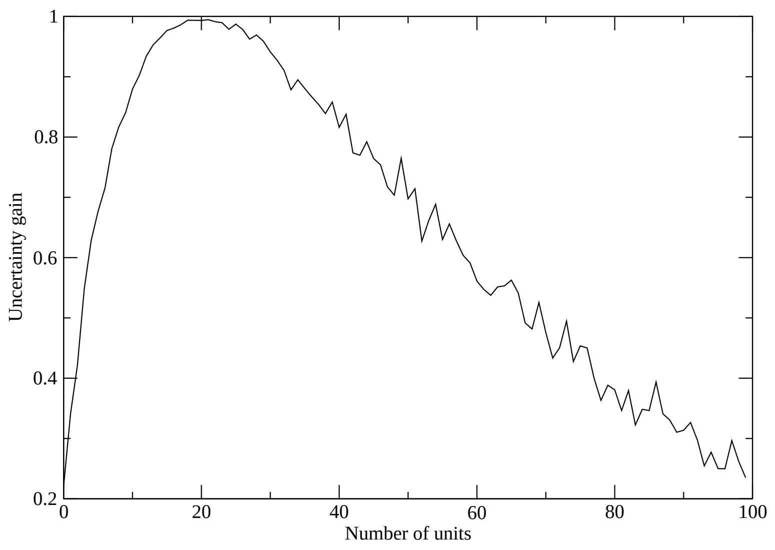


Figure 4.4: Uncertainty gain from varying the number of units (experiment *vary-num*)

of a unit's location over an area via its view radius, and the quantization of a large image into a small one.

We model each of these two sources. For quantization, we scatter points in a large map, downsample to the small map, and count the number of points. The ratio of downsampled points to original points is the measured information loss.

For the view overlap, we scatter points in a large map. When we calculate the viewable area for each point, we disperse its information value (say the constant 1) throughout its viewable area, but do not add anything to an area that is nonzero. By summing over the map and comparing to the original amount of information

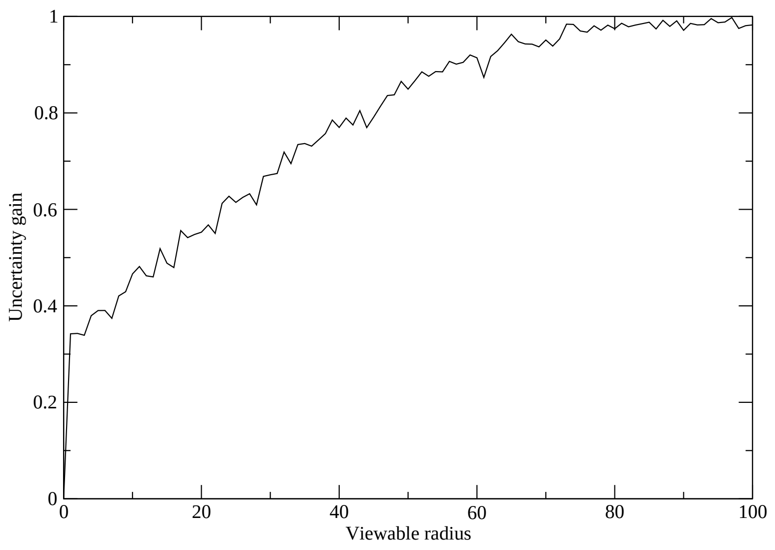


Figure 4.5: Uncertainty gain from varying the viewable radius of each unit (experiment *vary-rad*)

we have measured the information lost to overlap.

We calculate this loss with experiments *quant* and *overlap* from Table 4.3.4. Figure 4.3.4 shows that the information loss from overlap rises more rapidly than quantization for this map size, but both level off very slowly, and the combined positional information loss for our scheme is 11% for proportional numbers of units and map size.

We expect our modeling results show less information loss than trace-driven data would. This is because it is more common for units in RTS games to position in clusters instead of randomly, which increases information loss in both quantization

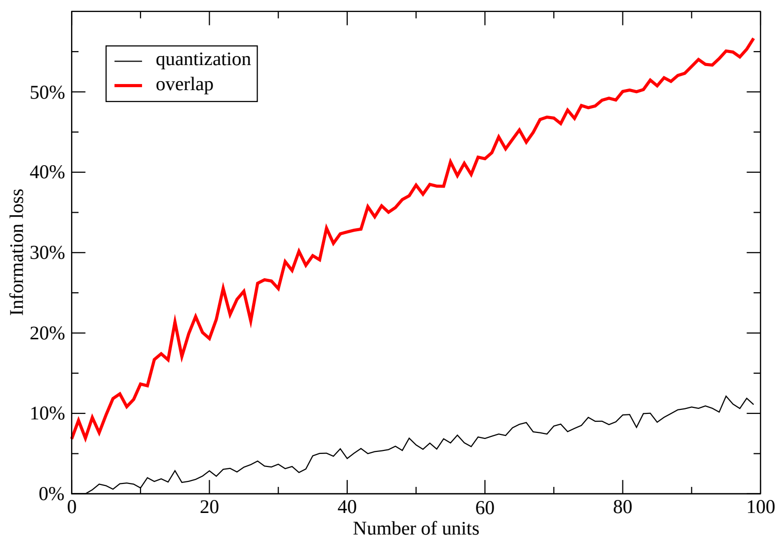


Figure 4.6: Information loss from quantization and overlap

and overlap.

Bandwidth

To calculate bandwidth requirements over time, we build towards an equation that determines how much data is sent by one player in a game played up to a particular instant.

RTS games supposedly happen in “real time”, but in fact they do have turns, albeit of the high granularity of a millisecond. In theory players could act every millisecond, but a typical move rate is an action every second, or four to five per second for especially intensive bursts. Our formal definition for bandwidth consumed should therefore scale down to milliseconds, but take into account the case of no user input for a given time slice.

Let vr_i be the enemy's viewable region at time i. We define m_i as the player's move at the given moment. This move can be considered a string containing the keyboard and mouse input.

Let

$$m_i = \left\{ \begin{array}{c} \text{player's move at time } i \\ \\ \epsilon \text{ if no move} \end{array} \right\} \tag{4.2}$$

We define sm_i, the secured version of the move as

$$sm_i = \left\{ \begin{array}{c} m_i \text{ if } m_i \in vr_i \\ \\ h(m_i, s) \text{ if } m_i \notin vr_i \\ \\ \epsilon \text{ if } m_i = \epsilon \end{array} \right\} \tag{4.3}$$

We define n_i, the new units at moment i as

$$n_i = \left\{ \begin{array}{c} \text{the string of units entering } vr_i \text{ at time } i \\ \\ \epsilon \text{ if no units enter } vr_i \text{ at time } i \end{array} \right\} \tag{4.4}$$

Let $sign(x)$ be a function that cryptographically signs string x with a player's secret key. We define s_i, the signature for the message at moment i as

$$s_i = \left\{ \begin{array}{c} sign(vr_i, sm_i, n_i) \\ \\ \epsilon \text{ if } (vr_i, sm_i, n_i) = \epsilon \end{array} \right\} \tag{4.5}$$

Using these definitions, we can construct the size of the data sent up to time t as

$$dataSent(t) = \sum_{i=1}^{t} |vr_i| + \sum_{i=1}^{t} |s_i| + \sum_{i=1}^{t} |sm_i| + \sum_{i=1}^{t} |n_i| \tag{4.6}$$

The last two summations of this equation are, for infrequent user input, considerably smaller in number of nonzero terms than the first two summations. Additionally, if $|vr_i|$ is stored as an image, it will likely exceed the data requirements for a string of user-input, or a signed hash. Users of today's Internet cannot expect to send and receive vr_i every millisecond. Therefore we relax the restriction that the viewable region be sent every time slice, and instead, send the region every r ms. This changes the definition to

$$dataSent(t) = \sum_{i=1}^{\lfloor \frac{t}{r} \rfloor} |vr_i| + \sum_{i=1}^{t} (|s_i| + |sm_i| + |n_i|) \tag{4.7}$$

Viewable areas can dominate equation 4.7 if they are large, as they may be sent frequently regardless of player interaction. On the other hand, if the cryptographic hashing or signing process is space-intensive, signatures will dominate the equation.

Replay ID	Player 1	Player 2	Date	Video Size
r1	mTw_ghostridah	SK_Insomnia	9/23/2004	2.82GB
r2	[4K]Grubby	WelcomeTo	10/10/2004	2.70GB
r3	HordeOfVad	aAa_RouF	11/07/2004	2.00GB
r4	Silvernoma	nT4everR[aDK]	11/07/2004	0.94GB
r5	64AMD_Cara	SK_Zad	11/29/2004	1.43GB
r6	SK_HeMaN	apm70	12/04/2004	2.65GB
r7	SK_Zacard	MYM]GoStop	12/05/2004	3.95GB
r8	30GamesADay	AzYWaSCrazY	12/13/2004	3.76GB

Table 4.3: User traces of Warcraft 3 games

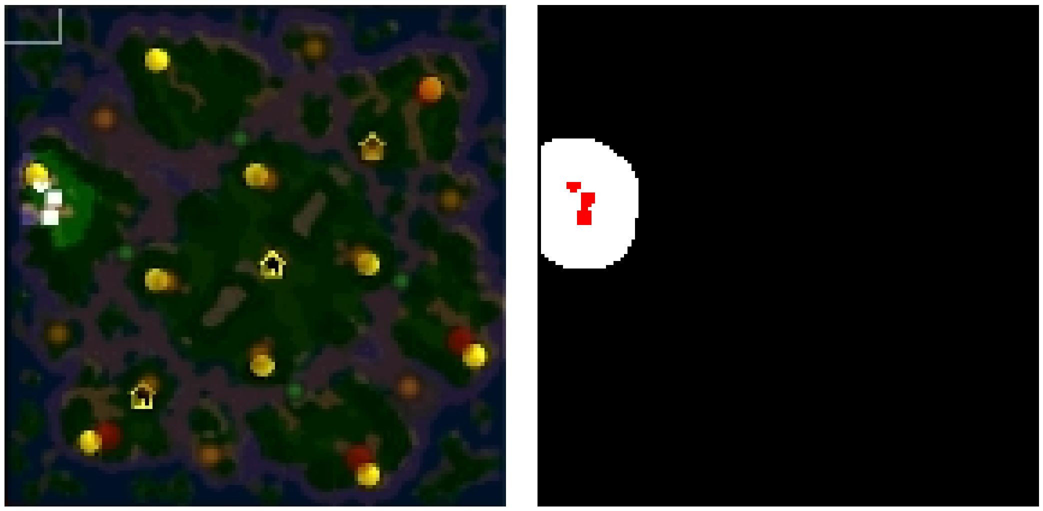


Figure 4.7: (a) mini-map for replay $r1$ (b) extracted region and unit information

4.3.5 Trace-driven Evaluation

In the previous section we evaluated protected RTS in a user-independent fashion, setting aside any gameplay dependent characteristics such as unit clustering or user event generation that may vary from game to game. We concluded that protected RTS would generate substantial uncertainty with the addition of each unit (peaking at 20 for our map size), and we provided an equation for the bandwidth of protected RTS. In this section we consider the effectiveness of protected RTS in as realistic a setting as we can generate without access to the source code for the game.

We more fully evaluate the network and uncertainty impact of our protocol by driving it from user traces of real-world Warcraft 3 games. These are freely available for download, and contain the information in them of which actions each user takes at each moment. They do not contain the viewable area information or unit positions. However, the replays are meant to be watched within the game engine, which derives the unit locations and viewable areas. Given access to user game traces and the appropriate information about unit locations and viewable areas, we can accurately evaluate the success of our scheme.

One technique to extract the needed data from a replay is to create a video capture of the replay, decode the video and focus on the "mini-map", which displays a small graphic indicating a player's units and their viewable area. The data is approximate; the mini-map is a downsampled two-dimensional representation of a three-dimensional collection of units and necessarily inaccurate. However, we can draw order-of-magnitude conclusions from this data.

To carry out our evaluation we select eight replays from well-known LAN tournament players whose games were unlikely to carry cheats, due to their in-person supervision and high profile. It should be noted however that the presence or absence of cheats in our traces has little bearing on our evaluation except to the extent that players with a maphack enabled may move their units more aggressively than non-cheaters. Our selection criteria for these replays included a variety of game lengths and race selections. The identification information for these replays is summarized in Table 4.3.5. We view each of these replays using the game engine and record video of the gameplay with a video capture tool designed for recording games [35]. We then extract each frame of the video, crop it to the small "mini-map" region of the game's heads-up display, and perform image analysis to extract unit and viewable region information. Figure 4.7 shows an example frame from the mini-map and the corresponding extracted unit and region information. We evaluate our traces on bandwidth and uncertainty gain.

Bandwidth

There are several unquantified areas of the protected RTS protocol: the size of the viewable areas, the hidden and clear events, and the unit transmission. We first examine the region information and the unit information in depth.

One of the unknowns in our analysis of the protected RTS protocol is the network impact of sending periodic region information. As an example we show in Figure 4.8(a) the size of a single game's viewable regions (player 1's) encoded in the PNG format as the game progresses. Figure 4.8(b) shows a histogram of the

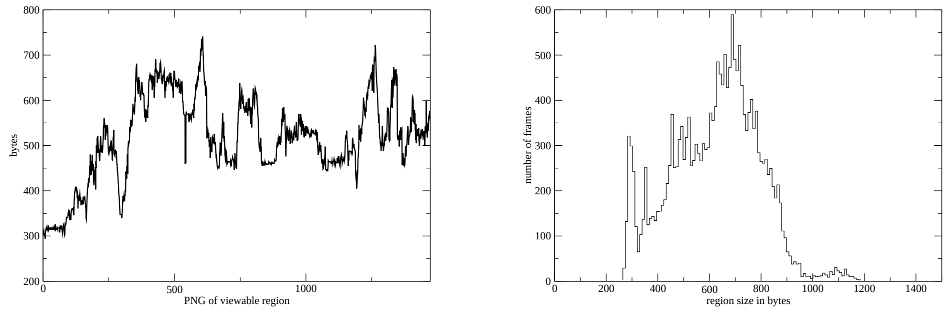


Figure 4.8: (a) Viewable regions for game $r1$ encoded as PNGs (b) Histogram of viewable region sizes over all traces

regions over all of our trace data, from which we conclude that most regions are under one kilobyte in size, and the median region in our traces is of size 647 bytes. The small peak in the distribution at 300 bytes is due to the equivalent starting positions for each player at the beginning of a game. To compare these region sizes with the space required for cryptographic signatures, the size of a signature using SHA-1 as the hash and RSA-1024 for the encryption is 128 bytes.

Another unquantified aspect of our analysis is the number of units controlled by each side of the game, and of those, the number who are visible to the other player over time. In Figure 4.9 we show an approximation of the number of units controlled by both players over the course of a typical game ($r1$) of Warcraft 3. This is only an approximation because it is difficult to deduce if a player-controlled block of pixels in the mini-map is a single large unit or several small ones, or not a unit at all but instead a building. In this work we treat buildings as stationary

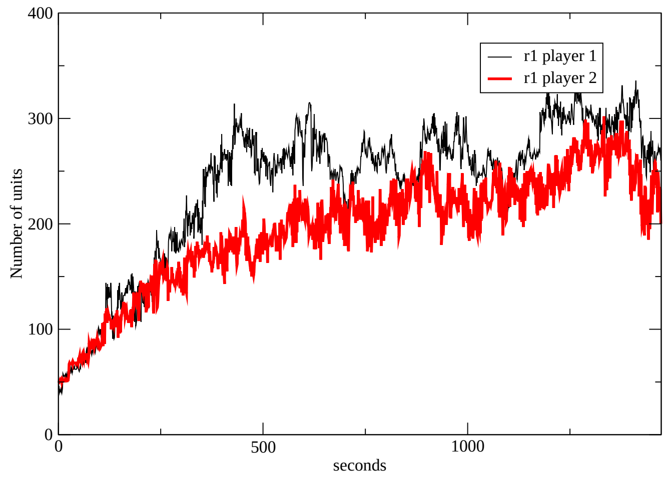


Figure 4.9: Number of units over time as extracted from Warcraft 3 replay $r1$

units, and we choose to over-estimate the number of units by assuming each block is composed of many units. Figure 4.9 shows the number of units increasing as the game progresses until a unit cap is reached, and peaking at around three hundred units for this game.

In our protected RTS scheme the number of units controlled by each side is largely unimportant except for when the units become visible to the other player, who must then be informed of their presence. As our trace data gives us access to each player's viewable area as well as the location of their units, we can calculate this quantity of visible units. One limitation of our analysis however is that we cannot identify units from one frame to the next. In the protected RTS protocol we would like to only send units that recently became visible to the other player (so as not to repeatedly send the same units). Since we cannot distinguish between units by frame however, we must overestimate the number of units to be sent at each frame. In Figure 4.10 we view player 1's perspective for replay $r1$ and show the number of units inside and outside of player 2's viewable region. The number of units sent is sporadic and typically more units are outside of the player's view than within. This is due to the nature of the game; many units are visible during a skirmish with the enemy, but this number quickly decreases as units retreat or eliminate each other. Looking over all games, we calculate the ratio of hidden units to sent units to be substantially in favor of unseen units: 6.97:1 on average. Figure 4.11 shows a CDF of the sent units over all traces. 40% of all frames are completely non-interactive between players, and 66% of all frames involve sending 20 units or less. We conclude that much of the RTS game takes place sight-unseen by the other player, and the instances of interplay between the two players are bursty as opposed to smooth.

The final unquantified bandwidth factor is the event data. Event data is sent with a mean number of events per second of 3.29. As with the unit data the ratio of hidden to clear events favors hidden (2.64:1). Given this last piece of data, we can complete our analysis of the bandwidth impact of our protected RTS scheme by calculating our scheme's total bandwidth using a one second window for transmitting events and viewable areas. Figure 4.12 shows the cumulative

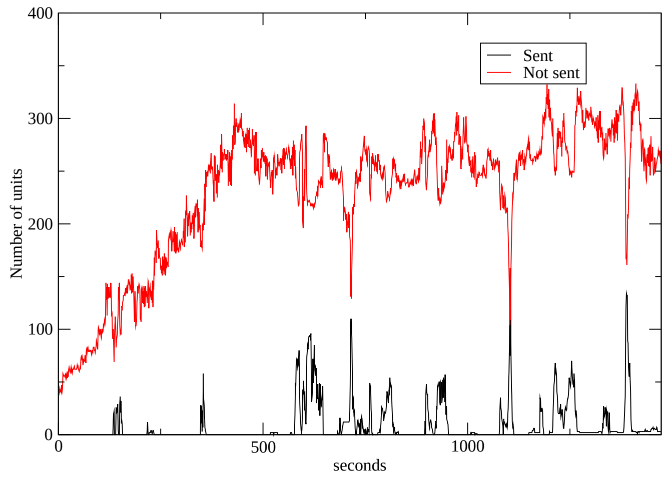


Figure 4.10: Number of units transmitted in cleartext to opponent over the course of game $r1$, player 1

bandwidth of the scheme over the course of game $r1$ from player 1's perspective, broken down into region data, event data, and unit data. The majority of the total bandwidth is consumed by the region data. The total bandwidth required for the 24.5 minute game is 784 kilobytes. We conclude that the bandwidth impact of our scheme fits within real-world network characteristics such as DSL.

Uncertainty Gain

Finally, we evaluate the protected RTS scheme using trace-driven data in terms of uncertainty gain; that is, we measure at each frame of the replay the extra quantity of entropy present in the viewable regions versus the known regions of

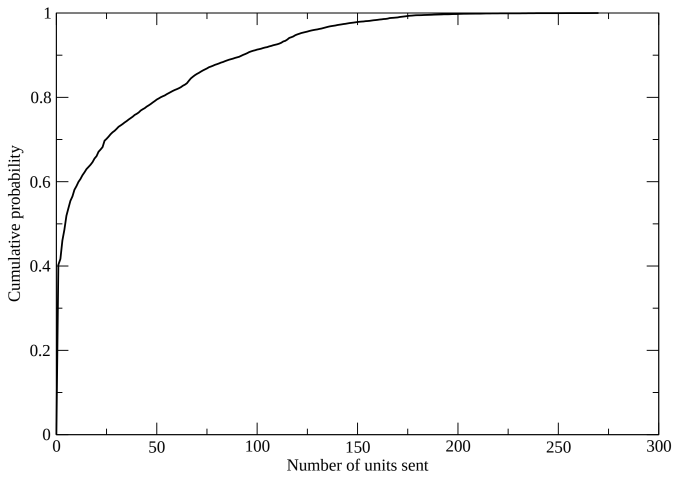


Figure 4.11: CDF of number of units sent cleartext per frame over all traces

the unprotected RTS protocol. To illustrate the gain over time we show the relative uncertainty for both the protected and unprotected protocol in Figure 4.13. The unprotected protocol has relatively constant uncertainty after an early build of units, while the protected protocol varies more widely but is consistently much higher. We summarize the gain over all traces in Figure 4.14, which shows a small peak around 0.1 for the beginnings of traces, and a larger peak around 0.6 for the rest of the traces. This indicates that the protected RTS protocol introduces substantial uncertainty into the protocol given real-world parameters.

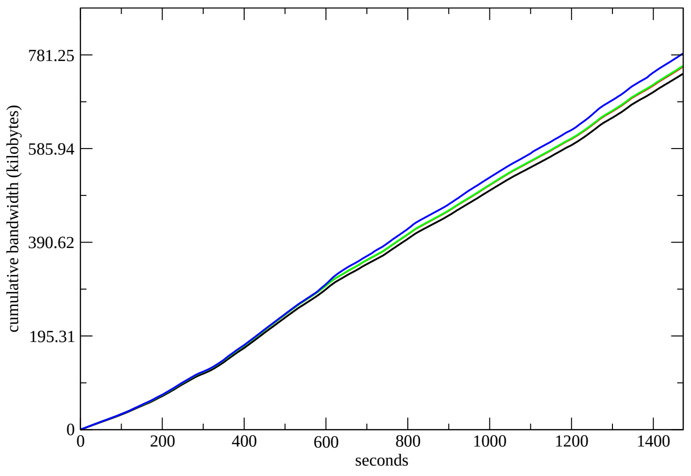


Figure 4.12: Cumulative bandwidth of protected RTS scheme

4.3.6 Conclusion

We have presented Protected RTS, a technique for securing on-line peer-to-peer applications from maphack cheats using bit commitment. In order to scale to a large number of entities and a rapid network protocol, Protected RTS makes the following information compromise: players no longer have access to each other's complete game state information, but instead have access to an opponent's viewable area. We recognize at least three limitations to this technique: it exposes some information, it detects but does not prevent cheaters, and the detection step requires a potentially complex verification procedure.

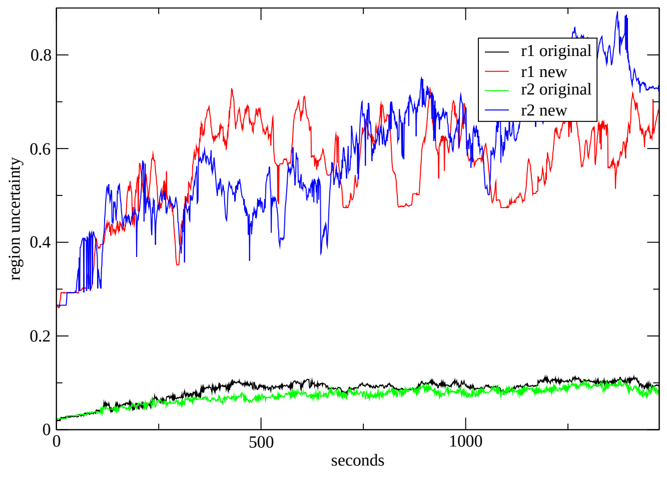


Figure 4.13: Uncertainty of standard RTS protocol vs. protected RTS protocol for game $r1$

The goal of the scheme is to hide information. We have evaluated the total information hidden using the metric of uncertainty in both user-independent and trace-driven experiments. Our user-independent evaluation show that varying the radius of view changes the amount of uncertainty slowly compared to varying the number of units, and that uncertainty peaks rapidly even for a small number of units (6 for our map size). The information concealed by our scheme is substantial, with a total uncertainty increase of .6 seen in our trace-driven evaluation.

While protected RTS substantially increases hidden information, its network performance characteristics must also be suitable for games. We have presented a model for the bandwidth consumed by this scheme, which depends heavily on the

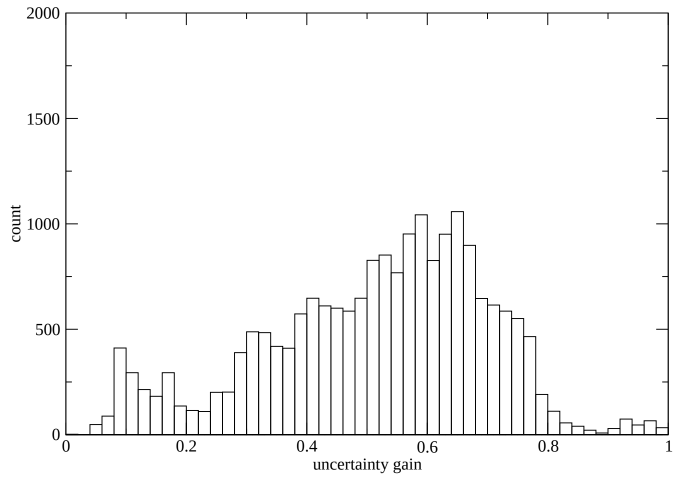


Figure 4.14: Histogram of per-frame uncertainty gain of protected RTS protocol over standard RTS protocol

amount of player interaction in RTS games. The conclusion from our trace-driven evaluation is that the protected RTS scheme introduces a substantial amount of additional bandwidth, dominated by the viewable regions and, to a lesser extent, the cryptographic signatures. However the increased bandwidth required by protected RTS still fits within modern networking capabilities.

Chapter 5

Conclusion

While on-line gaming is primarily a recreational activity, it shares a great deal in common with other interactive on-line applications, such as military or disaster simulations, distance learning, and interactive storytelling and art. Furthermore, the advances in gaming have historically driven the fields of graphics and networks forward. Unfortunately on-line games, while increasing in popularity, have become extremely costly to produce and maintain, and popular games are filled with cheaters. In this thesis we have taken the perspective of GameCorp, an imaginary company that would like to host a popular on-line game at a minimum of cost. This thesis has focused on addressing two important issues in networked games: (1) supporting a huge number of users for a popular game and (2) addressing the inevitable creation of cheats that come with a popular game. To this end we have undertaken a number of studies. Specifically, we have:

1. Completed a novel multi-year measurement study of gamers and game workloads based on data collected from a number of unique sources. The study shows that provisioning for hosting is challenging and users are demanding, based on the following specific results: (1) gamers are impatient and likely to leave an unsatisfying game in the first few minutes, (2) game popularity follows a power-law distribution, with a small number of games being orders of magnitude more popular than the others, (3) game workloads are very stable from week to week but difficult to predict over longer timescales and (4) game workloads are synchronized between games and between games and other interactive applications such as web traffic. Most importantly, we conclude in this study that games are unlikely to achieve substantial benefits from shared hosting infrastructure across games.

2. Addressed the server selection problem for the public server architecture via a centralized service that matches a player with a server fitting their criteria. The two challenging aspects of matching players with servers are (1) knowing a given server is currently available and (2) knowing that server fits the player's latency needs. The status quo addresses this problem via polling; each player polls all available servers and determines latency, availability and game rules directly. To centralize the service we must be able to determine the latency between a given client and server, neither of which are under

our control. Our approach to this problem is to group servers with clients by geographic region, and poll potential client server matches with *King*, a third-party latency tool. Experimental results show our service saves clients on the order of tens of milliseconds of latency, but also thousands of kilobit-miles. Analytic results show our service reduces the number of network probes per hour from $O(n^2)$ to $O(n)$ for a network with n clients and n servers.

3. Designed a new game hosting architecture for a public server MMO (PSMMO) that frees the publisher from the hosting burden of the client server model while allowing for a "massive" on-line gaming world and user-generated content. The PSMMO architecture focuses on persistent content shared between servers such as new abilities, levels, or equipment, generally called "loot". Loot and the public server model do not go together naturally, as a server cannot verify that a client should have a certain piece of loot, and a player can set up their own server to issue themselves loot. PSMMO addresses these concerns with a combination of cryptography, CAPTCHAs and incentives. The cryptography is used for player authentication and loot authentication, the CAPTCHAs are used to verify actual gamers are playing the game instead of bots, and incentives are used to motivate servers to provide interesting content and players to gravitate towards interesting servers. PSMMO also allows for user-generated content to be incorporated into the

world in a secure fashion, further harnessing user resources to decrease the burden of hosting an on-line game.

4. Developed a classification for client-side cheats based on the flow of control of the game from the user to the game to the network and back. All cheats on this control path can be classified into four distinct categories: information exposure, protocol manipulation, abstraction of input, and abstraction of output. The taxonomy enables game creators to consider the full spectrum of cheats that are likely to be created should their game become popular.

5. Addressed the *maphack*, the dominant cheat in peer-to-peer real-time strategy games. In this cheat a player can see information about another player's pieces that is intended to be hidden. In a peer-to-peer game secrets about unit types and locations need to be kept but fabricating hidden data needs to be prohibited. The traditional approach to solving this problem in cryptography is via bit-commitment: party a sends a commitment to a secret such as a cryptographic hash without revealing the secret. At a later date a can reveal the secret and others can verify that the secret and the commitment agree. The challenge in an RTS game is (1) scalability, as there may be hundreds of units per side and (2) visibility information. Each unit can see a certain distance, so the enemy units that are secret or visible changes

from moment to moment in an RTS game as units move. We address this challenge with a region-based bit commitment scheme in which each player knows the other player's visible area. This scheme compromises information exposure for scalability. We evaluate exactly how much information is exposed and how much additional bandwidth is consumed by the scheme and conclude that it substantially increases uncertainty while falling within acceptable bandwidth limits for on-line gameplay.

Future Work

The contributions of this thesis are steps towards addressing key challenges of hosting and cheating in on-line games, but they merely touch on specific problems in broad areas.

We believe further research into cheating in on-line games is required for on-line games to thrive. Two ideal results in this area would be (1) a scheme that prevents client modification of software and (2) a generalized statistical cheat detector for game servers. The first result would be very powerful in preventing cheating, as the dominant cheating concern for games is client-side cheats. This result may be theoretically unachievable, or require enabling compromises such as trusted hardware. The second result would be less powerful, as any statistical detector will ride the line between detecting normal players as cheaters, and letting cheaters go.

However a generalized statistical detector would have the opportunity to detect any anomalous behavior, even behavior achieved without client software modification. Examples of such behavior could include game traffic modification in the network or flaws in the game design being exploited by a few players. As future work, we plan to work towards these two results.

Continued research into scalable solutions for hosting and maintaining interactive on-line applications is also needed if these applications are to succeed in the future. We believe the public server model is especially suited for the future growth of games, and so our future work in this area involves further developing the public server model. Our PSMMO architecture has been designed but not evaluated. We plan to modify an existing public server game such as Quake to conform to the PSMMO design, as well as create a centralized authority implementation for tracking player minutes and issuing loot. This will enable us to directly quantify load on the publisher, as well as provide a reusable service for other public server games. We also plan to develop a networked reputation system for gamers and game servers that can disseminate knowledge more efficiently than player social interactions. The public server model can benefit greatly from such a system, as the number of gamers and game servers is typically too large for people to track. As reputation systems are an open research field, it remains to be determined what techniques will work best for public server games.

With advances allowing for scalable hosting and content development for games, and techniques to prevent or detect cheating in on-line games, we believe on-line gaming will be prepared to grow into a dominant form of entertainment worldwide.

References

[1] A. Akkawi, S. Schaller, O. Wellnitz, and L. Wolf. A Mobile Gaming Platform for the IMS. In *Proceedings of Netgames*, pages 77–84, 2004.

[2] "Trusted Computing Platform Alliance". "trusted computing group specifications".

[3] AMDZone. Valve Releases Hammer Port of Counter-Strike Server. `http://www.amdzone.com/releaseview.cfm?ReleaseID=1050`, 2003.

[4] AMX Mod Developers. AMX Mod Server Plugin. `http://amxmod.net/`.

[5] G. Armitage. Sensitivity of Quake3 Players To Network Latency. In *Internet Measurement Workshop (Poster Session)*, November 2001.

[6] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof Playout for Centralized and Distributed Online Games. In *INFOCOM*, pages 104–113, 2001.

[7] T. Beier and S. Neely. Feature-based Image Metamorphasis. *Computer Graphics(SIGGRAPH '92)*, pages 35–42, July 1992.

[8] BioWare. BioWare's Neverwinter Nights. `http://nwn.bioware.com/`, 2006.

[9] M. Buro. ORTS: A Hack-free RTS Game Environment. In *Proceedings of the International Computers and Games Conference*, 2002.

[10] Butterfly.net, Inc. Butterfly Grid Solution for Online Games. `http://www.butterfly.net/`, 2003.

[11] J. Castellà-Roca, J. Domingo-Ferrer, A. Riera, and J. Borrell. Practical mental poker without a ttp based on homomorphic encryption. In Thomas Johansson and Subhamoy Maitra, editors, *Progress in Cryptology-Indocrypt'2003*, number 2904 in LNCS, pages pp 280–294. Berlin: Springer-Verlag, 2003.

[12] L. Catledge and J. Pitkow. Characterizing Browsing Strategies in the World-Wide Web. *Computer Networks and ISDN Systems*, 27(6):1065–1073, 1995.

[13] B. Chen and M. Maheswaran. A Cheat Controlled Protocol for Centralized Online Multiplayer Games. In *NetGames*, April 2004.

[14] A. D. Cheok, S. W. Fong, K. H. Goh, X. Yang, W. Liu, and F. Farzbiz. Human Pacman: a sensing-based mobile entertainment system with ubiquitous computing and tangible interaction. In *Proceedings of Netgames*, 2003.

[15] M. Claypool, D. LaPoint, and J. Winslow. Network Analysis of Counter-Strike and Starcraft. In *IEEE Internation Performance, Computing, and Communications Conference*, April 2003.

[16] CNN. SoBig.F Breaks Virus Speed Records. `http://www.cnn.com/2003/TECH/internet/08/21/sobig.virus`, 2003.

[17] C. Crepeau. A Secure Poker Protocol That Minimizes the Effect of Player Coalitions, 1986.

[18] C. Crepeau. A Zero-Knowledge Poker Protocol That Achieves Confidentiality of the Players' Strategy or How to Achieve an Electronic Poker Face. In *CRYPTO*, pages 239–247, 1986.

[19] E. Cronin, B. Filstrup, A. Kurc, and S. Jamin. An Efficient Synchronization Mechanism for Mirrored Game Architectures. In *Proceedings of Netgames*, 2002.

[20] Eric Cronin, Burton Filstrup, and Sugih Jamin. Cheat Proofing Dead Reckoned Multiplayer Games (Extended Abstract). In *Proceedings of the International Application and Development of Computer Games Conference*, 2003.

[21] M. Crovella and A. Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of ACM SIGMETRICS*, May 1996.

[22] C.W. Reynolds. Steering Behaviors for Autonomous Characters. In *Proceedings of Game Developers Conference*, 1999.

[23] S. Davis. Why cheating matters. In *Proceedings of the Game Developer's Conference*, 2001.

[24] M. Delap, Bjorn Knutsson, Honghui Lu, Oleg Sokolsky, Usa Sammapun, Insup Lee, and Christos Tsarouchis. Is Runtime Verification Applicable to Cheat Detection? In *NetGames*, April 2004.

[25] DFC Intelligence. Online Game Market is Growing but Making Money is Difficult. `http://www.dfcint.com/news/prjune252003.html`.

[26] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[27] E. Castronova. Virtual worlds: A first-hand account of market and society on the cyberian frontier. In *CESifo Working Paper Series No. 618*, 2001.

[28] eBay. eBay - The World's Online Marketplace. `http://www.ebay.com`.

[29] Electronic Arts, Inc. EA.com. `http://www.ea.com/`, 2003.

[30] Everquest. EverquestLive.com. `http://eqlive.station.sony.com`.

[31] F. Yu. Collapse, Death And Re-Birth In The Chinese Online Game Market. `http://www.chinatechnews.com/index.php?action=show&type=news&id=2480`, March 2005.

[32] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server. In *Internet Measurement Workshop*, November 2002.

[33] W. Feng, F. Chang, W. Feng, and J. Walpole. Provisioning On-line Games: A Traffic Analysis of a Busy Counter-Strike Server. In *Proc. of the Internet Measurement Workshop*, November 2002.

[34] W. Feng and W. Feng. On the Geographic Distribution of On-line Game Servers and Players. In *NetGames*, May 2003.

[35] "Fraps". "fraps real-time capture and benchmarking".

[36] GameSpy Industries. GameSpy: Gaming's Home Page. `http://www.gamespy.com/`, 2002.

[37] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games. In *Proceedings of NOSSDAV*, June 2004.

[38] Geobytes, Inc. Geobytes Home Page. `http://www.geobytes.com/`, 2003.

[39] Global Grid Forum. `http://www.ggf.org`.

[40] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer Workload. In *Proceedings of ACM SOSP*, October 2003.

[41] Half-Life Admin Mod Developers. Half-Life Admin Mod Home. `http://www.adminmod.org/`.

[42] T. Henderson. Latency and User Behaviour on a Multiplayer Game Server. In *Networked Group Communication*, pages 1–13, 2001.

[43] T. Henderson. Observations on Game Server Discovery Mechanisms. In *NetGames*, April 2002.

[44] T. Henderson and S. Bhatti. Modelling User Behavior in Networked Games. In *ACM Multimedia*, pages 212–220, 2001.

[45] S. Hu and G. Liao. Scalable Peer-to-Peer Networked Virtual Environment. In *NetGames*, April 2004.

[46] IBM Corp. On demand business. `http://www.ibm.com/ondemand`.

[47] IBM Corp. Tivoli intelligent thinkdynamic orchestrator. `http://www.ibm.com/software/tivoli/products/intell-orch`, 2004.

[48] IDC. HP utility data center: Enabling enhanced data center agility. `http://www.hp.com/large/globalsolutions/ae/pdfs/udc_enabling.pdf`, May 2003.

[49] IGE. IGE: The Leading MMORPG Services Company. `http://www.ige.com/`, 2005.

[50] IGN. Neverwinter Nights Vault. `http://nwvault.ign.com/`, 2006.

[51] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned Federation of Game Servers: a Peer-to-peer Approach to Scalable Multiplayer Online Games. In *NetGames*, April 2004.

[52] Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. On the placement of internet instrumentation. In *INFOCOM (1)*, pages 295–304, 2000.

[53] K. Gummadi and S. Saroiu and S. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Internet Measurement Workshop*, Marseille, France, November 2002.

[54] K. Mørch. Cheating in Online Games- Threats and Solutions. In *Publication No: DART/01/03*. Norwegian Computing Center/Applied Research and Development, January 2003.

[55] Y. Kaneda, H. Takahashi, M. Saito, H. Aida, and H. Tokuda. A Challenge for Reusing Multiplayer Online Games without Modifying Binaries. In *NetGames*, October 2005.

[56] A. Kirmse and C. Kirmse. Security in On-line Games. In *Game Developer*, July 1997.

[57] B. Krishnamurthy and J. Wang. On Network-aware Clustering of Webclients. *Proceedings of SIGCOMM 2000*, 2000.

[58] B. Krishnamurthy, C. Wills, and Y. Zhang. the use and performance of content distribution networks, 2001.

[59] M. Lewis and J. Jacobson. Game engines in scientific research.

[60] Kang Li, Shanshan Ding, Doug McCreary, and Steve Webb. Analysis of State Exposure Control to Prevent Cheating in Online Games. In *Proceedings of NOSSDAV*, 2004.

[61] M. Pritchard. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. `http://www.gamasutra.com/features/200000724/pritchard_01.htm`.

[62] Edson Manoel et al. *Provisioning On Demand: Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator*. IBM International Technical Support Organization, December 2003. `http://www.redbooks.ibm.com`.

[63] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Computer Graphics*, 31(Annual Conference Series):415–420, 1997.

[64] Microsoft. MSN Games. `http://zone.msn.com/`, 2006.

[65] Microsoft Corporation. Xbox Live. `http://www.xbox.com/live`, 2003.

[66] mshmro.com. mshmro.com. `http://www.mshmro.com/`.

[67] C. Ondrejka. People Powered Places: Second Life, Saving Games, and 6 Missing Pieces. In *Microsoft Research Tech Talk*, August 2005.

[68] Online Game Publishers. Private Communication. 2004.

[69] Open Source. Crypto++: A Free C++ Class Library of Cryptographic Schemes. `http://cryptopp.com/`.

[70] V. N. Padmanabhan and L. Subramanian. Determining the Geographic Location of Internet Hosts. In *SIGMETRICS*, June 2001.

[71] C. Palazzi, S. Ferretti, S. Cacciaguerra, and M. Roccetti. On Maintaining Interactivity in Event Delivery Synchronization for Mirrored Game Architectures. In *Global Telecommunications Conference Workshops*, December 2004.

[72] D. Papagiannaki, N. Taft, Z. Zhang, and C. Diot. Long-term forecasting of internet backbone traffic: Observations and initial models, 2003.

[73] QStat Developers. QStat - Real-time Game Server Status. `http://www.qstat.org/`.

[74] ReliaSoft Corporation. Life Data Analysis and Reliability Engineering Theory and Principles Reference from ReliaSoft. `http://www.weibull.com/lifedatawebcontents.htm`, 2003.

[75] T. Rhyne. Entertainment Computing: Computer Games' Influence on Scientific and Information Visualization. *Computer*, 33(12):154–156, December 2000.

[76] Y. Rui and Z. Liu. Artifacial: Automated Reverse Turing Test Using Facial Features. In *ACM Multimedia*, 2003.

[77] S. Kline and A. Arlidge. Online Gaming as Emergent Social Media: A Survey. `http://www.sfu.ca/media-lab/onlinegaming/report.htm`, January 2003.

[78] Debanjan Saha, Sambit Sahu, and Anees Shaikh. A service platform for online games. In *NetGames*, Redwood City, CA 2003.

[79] "ServerSpy". "server spy: Find gamers online".

[80] S.Gadde, J.Chase, and M. Rabinovich. Web Caching and Content Distribution: A View from the Interior. In *Web Caching and Content Delivery Workshop*, May 2000.

[81] A. Shaikh, S. Sahu, M. Rosu, M. Shea, and D. Saha. Implementation of a Service Platform for Online Games. In *NetGames*, August 2004.

[82] A. Shamir, R. Rivest, and L. Adleman. Mental poker, 1982.

[83] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423, July 1948.

[84] SirBruce. Total MMOG Active Subscriptions. `http://www.mmogchart.com/Chart4.html`.

[85] T. Standage. Chasing the Dream. *The Economist*, August 2005.

[86] Sun Microsystems. N1 Grid – introducing just in time computing. `http://wwws.sun.com/software/solutions/n1/wp-n1.pdf`, 2003.

[87] Valve, Inc. Steam. `http://www.steampowered.com/`, 2005.

[88] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt*, 2003.

[89] Wikipedia. List of Half-life Mods. `http://en.wikipedia.org/wiki/List_of_Half-Life_mods/`, 2006.

[90] World of Warcraft. World of Warcraft Community Site. `http://www.worldofwarcraft.com`.

[91] J. Yan and H-J Choi. Security Issues in Online Games. *The Electronic Library: International Journal for the Application of Technology in Information Environments*, 20(2), 2002.

[92] Jeff Yan and Brian Randell. A Systematic Classification of Cheating in Online Games. In *Proceedings of Netgames*, 2005.

[93] W. Zhao, V. Varadharajan, and Y. Mu. A Secure Mental Poker Protocol over the Internet. In *CRPITS*, pages 105–109, 2003.