Effects of Dynamic Player Behavior in Massively Multiplayer Online Games

Xinyu Zhuang

May 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Srinivasan Seshan (Advisor)
Dave Andersen

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Keywords: TBD

Abstract

This paper describes a measurement study done on churn rates in massively-multiplayer online games (MMOGs). A leading commercial MMOG was studied, and the rates at which a sample of players log in and out of the game were determined. Analysis of this data yields several patterns of interest, and give rise to a set of recommendations in the design of future large-scale systems. In particular, churn rate is of utmost importance in the design of a distributed MMOG, and we study how the results of our measurements indicate deficiencies in current distributed infrastructure and what changes must be made to accomodate the churn rates we have found.

Acknowledgments

TBD

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Massively-multiplayer online computer games (MMOGs) are becoming increasingly common. With the advent of fast broadband Internet connectivity, more and more gamers are spending their time playing games online in virtual worlds. A good example is Blizzard Inc.'s World of Warcraft [16].

As people make the foray into virtual worlds, the demands on the systems that support these worlds increases. Not only do gamers want to play in an MMOG with thousands of other players, they also want to be active participants. Linden Labs' Second Life [20] is one MMOG where players create custom content in a dynamic environment that can change based on their actions. All this points to the need for infrastructure capable of supporting a large number of players, and also one where dynamic content is constantly pushed to players over the network.

Distributed systems are a proposed solution to the scalability problem, and there has been much work in this area recently. While distributed systems have been previously studied in great detail, the environments in which they have been used are somewhat different from that of a typical MMOG. It is unknown how the dynamic nature of gameplay affects the performance of distributed systems, and in particular, whether these dynamic behaviors should influence the design of systems.

This thesis seeks to address these questions:

- What are the dynamics of MMOG clients, and how do these dynamics differ from more traditional distributed system clients?
- How do we design systems to take into account such dynamic behaviors?

1.1 Massively Multiplayer Online Games

Massively Multiplayer Online Games are computer games where a large number of players interact with each other online in a persistant virtual world. Any game meeting these requirements is considered an MMOG, and often sub-genres are used to designate particular variations, such as the Massively Multiplayer Online Roleplaying Game (MMORPG). They key features of an MMOG are the large player base, and the game world where players are able to permanently modify their character and perhaps other aspects of the world as well. Thus a game with many players but no sense of permanence (*e.g.* once a player logs off she loses all the items she collected during the game) is generally not considered an MMOG.

Another important aspect of MMOGs is the lack of a clearly defined end-goal. Unlike other games where players can win or lose (for example by defeating all the other players in the game), an MMOG "lasts forever" in that players can never reach a state where they are considered to have won the game. Thus, the time a player spends in the game (whether in total or for a single game session) does not depend on how long it takes to win the game, but rather is governed by how long the player wishes to play.

MMOGs usually contain many computer-controlled elements, such as monsters or non-player characters that are AI-driven. Maintaining these elements requires computational resources.

1.1.1 World of Warcraft

World of Warcraft [16] is a popular MMOG made by Blizzard, Inc. As of March 2007, there were an estimated 8.5 million subscribers to the game[1], with more players joining daily.

To accomodate the large number of players, World of Warcraft uses *realms*, which are independent copies of the game world. Players choose a realm in which they wish to play in, and their character is then bound to that realm. This allows Blizzard to keep a limit on the number of players per realm, at the cost of preventing players from experiencing a true seamless virtual world.

1.1.2 Second Life

Second Life [20] by Linden Labs is a unique MMOG that gives users the ability to create and modify in-game content at a level that is unseen in any other game. The majority of in-game content is player-created, and all content down to the background music is delivered to the client on demand. This results in large amounts of data that need to be constantly sent to multiple clients.Furthermore, the SecondLife virtual world is not divided into separate realms as is done in World of Warcraft. Any two SecondLife players logged into the game are in the same huge world and can interact with one another. SecondLife uses large server farms to deal with the high computational demand on maintaining such a world.

The population count of SecondLife as of February 2007 was close to 4 million[2].

1.1.3 Scalability Issues

While most games do not have as large bandwidth requirements as Second Life, the popularity of the game is a sign that future MMOGs might move towards more dynamic content. Even as of now, most MMOGs are heavily taxed in terms of bandwidth and computational resources. The

[1]See `http://www.computerandvideogames.com/article.php?id=159422`

[2]See http://www.secondlifeinsider.com/2007/02/25/today-in-second-life-saturday-24-february-2007/

major scalability issues facing MMOGs are:

- Handling large number of client connections.
- Maintaining a large virtual world.
- Distributing dynamic content.

The most common approach to problem of scalability is to divide the virtual world into realms. However, this is generally considered to be a solution to a technical limitation, rather than having any game-related significance. In many cases (such as with Second Life), there is an inherent attraction to having a single virtual world, and thus alternate solutions to the scalability problem are appealing.

1.2 Solving Scalability Issues with Distributed Systems

Several authors have proposed solutions to the scalability problem facing MMOGs. All of these solutions hinge on the use of distributed hash tables (DHTs) for routing and delivery: The system described by Knutsson *et. al.*[18] uses Pastry [22] and Scribe [8], while Colyseus [6] makes use of Mercury [5]. GauthierDickey [11] describes a generic architecture that suggests the use of a DHT.

[18] and [11] both discuss the requirement of distributed MMOGs to implement some form of data storage. Given the importance of permanence in MMOGs, a reliable storage system is necessary in any framework designed to support MMOGs. While there has not been a lot of work with regards to permanent distributed data stores specifically for online games, the long line of work in distributed filesystems is likely to apply to this area. For example, OceanStore [19] is a distributed object store designed for large deployment scenarios.

1.3 Problems Facing Distributed Games

While distributed systems can help solve the scalability issue, they do have problems of their own that must be addressed. The problems this thesis focuses on are those to do with player dynamics: the effects of human players on session times, load distribution, and other metrics.

Specifically, moving a game to a fully distributed system entails treating game players as not mere clients but peers that bear some of the responsibility of keeping the game functioning. These responsibilities are as outlined above: maintaining the game state and distributing content. The game player thus has direct influence on the reliability and availability of the system, and the player's decisions on when to play the game and how long to play the game affect the overall performance of the distributed system.

There are of course other challenges that must be overcome when building a distributed game, with security (*i.e.* players cheating) being one big issue. Some work has been done in this area ([4], [13]), but we choose not to focus on this problem.

1.3.1 Player Dynamics

An important player dynamic is churn rate. Churn rate is a general term that refers to the rate at which entities enter or leave some state. The most common measurement of churn as related to distributed systems is *session churn*, the rate at which nodes join and part from the system. Session churn poses a problem for DHTs because there is usually some overhead involved in adding or removing a node from the system. Additionally, if some game state is maintained at nodes, each time a node leaves the system it must pass on this state to someone else. This can lead to poor performance, and even temporary inability to access data.

Closely related to session churn is *peer availability*, the fraction of time a node is connected to the network. [7] argues that building a large-scale, highly-available storage system utilizing inherently unreliable distributed peers is not possible. If peers have low availability, then the only way to ensure data is always available is to heavily replicate it, and beyond a certain level peers will not have enough bandwidth to support the level of replication necessary. While the results mainly apply to distributed storage systems that aim to hold huge amounts of data, it is still true that a distributed game will need high levels of replication if node availability is low. *Session length* and *downtime* directly affect a node's availability.

Another form of churn is *location churn*, which is the rate at which players enter and leave a logical area of the game world. Location churn and session churn are identical concepts, if we think of a location as being served by a system of its own. In fact, many existing MMOGs partition the game world into locations and have each location be handled by a specific server. Any system which performs such partitioning will be affected by location churn.

Fundamentally, we can explain the different forms of churn a game might face with the concept of *interest churn*, which refers to the rate at which a player's interest in something changes. For example, session churn is the rate at which a player's interest in playing the game changes, and location churn is the rate at which a player's interest in staying in a particular location changes. Thus there is a form of churn associated with every object (or collection of objects) a player could be interested in, and if the system needs to keep track of a player's interest, then it will be affected by churn. Obviously, the length of time over which the player is interested in the object will also affect the system.

Currently, there is little literature on the nature of these player dynamics, whether for distributed games or for traditional client-server model games. Since there are very few truly distributed games to begin with, the most appropriate targets for observation are popular existing MMOGs. Understanding player behavior in such games is crucial if we are to build systems to eventually replace them.

1.4 Thesis Overview and Contributions

The main contribution of this thesis is an advancement in the understanding of player dynamics in multiplayer games. Specifically, we conduct an extensive measurement study of a popular

MMOG, and present the results obtained in the context of the design of distributed systems for such games. Our primary focus is on producing a set of recommendations for the design of distributed game architectures.

1.5 Related Work

To the best of our knowledge, there has been no prior work focusing on player dynamics in MMOGs. Chambers *et. al.* [9] conducted a measurement study of many online games, focusing in particular on the first-person-shooter game Counter-Strike. While some of the games that were measured could loosely be considered MMOGs, the majority were online games that consisted of separate game sessions lasting at most a few hours. For example, in Counter-Strike, a team of players can win a game session by defeating the opposing team. Thus, the session lengths (the time a player spends connected to a *server*, not the time spent in a specific game session) reported in [9] for Counter-Strike are likely to be influenced by the average length of a game session, since a gamer will probably complete a session before leaving the game. [9] also describes higher-level characteristics of games such as game popularity, which are not the focus of this thesis.

Another analysis of the game Counter-Strike is presented by Feng et. al [10]. However, the paper focused mainly on bandwidth measurements and traffic behavior.

There has been more work in characterizing churn in peer-to-peer systems. Stutzbach and Rejaie [25] conducted an in-depth analysis of churn rates in P2P systems, and also provided a set of guidelines to follow when attempting to measure churn. They claim to be the first study to focus primarily on churn in P2P networks. The metrics used in this paper heavily influenced our choice of metrics when measuring player patterns in MMOGs. Sen and Wang [24] provide an analysis of peer-to-peer traffic, with some information on session lengths. Gummadi *et. al.* [14] also provides a characterization of session lengths in the P2P application Kazaa.

Chapter 2

World of Warcraft Measurement Study

It is important to appreciate the nature of dynamic player behavior in massively-multiplayer online games before attempting to design systems for such games. To this end, an extensive study of a popular online MMOG was conducted.

2.1 World of Warcraft

World of Warcraft (WoW) is a well-known MMOG that has a very large population of players. The game is designed on the standard client-server model, with players connecting to various servers operated by Blizzard. As mentioned previously, players choose a realm that they wish to create their character on, and that character is then limited to interacting only within that realm.

The realm chosen for the measurement study was Frostmane[1].

2.1.1 Characters

More specifically, each player (WoW account holder) is able to create up to 10 characters per realm, and up to 50 characters across all realms [17]. A player may only play as one character at

[1]Frostmane is a Player-versus-Player realm. There are several different realm types, where the main distinction is that game mechanics are somewhat different. For our purposes there is no real difference between the realm types.

any time.

A character corresponds to an in-game avatar, which possesses a set of attributes as well as an inventory of equipment. The key attributes that concern us are:

Name. A unique-per-realm identifier of the character.

Race and Class. Specific character types that imbue different abilities and properties to the character.

Level. A measure of the power a character has; a higher level character is more powerful than a lower level character. As of the time of writing, levels range from 1 to 70. Characters gain levels by earning *experience points* through fighting or completing quests. When enough experience points are obtained, the character will gain a level. Levels are expected to be proportional to amount of time spent playing the game.

Due to limitations in the amount of information exposed to clients by the game server, there is no straightforward way to correlate a character with a particular player (through any unique identifier such as an IP address or an account number) or vice versa. Therefore it is not possible to tell whether two characters actually belong to the same player or not.

2.1.2 Factions

There exist two opposing factions in the game, the Alliance and the Horde. Characters are either in one faction or the other, and the game is structured such that members of opposing factions will be hostile to one another. A character is also usually unable to understand anything said by a character of the other faction, and additionally, characters from one faction are only able to obtain information about chracters of the same faction. Also, a player cannot have characters that are in the same realm but of different factions.

There are no major differences between the factions that would affect our measurements. The faction chosen for measurement was the Alliance.

2.1.3 Game World

The game world is partitioned into *zones*. In general, movement from one zone to another is seamless, with a few exceptions. Players have large freedom of movement, and there exist several forms of transportation that help players move from one zone to another.

2.2 Measurement Infrastructure

World of Warcraft is a commercial, closed source game. As such, it was never designed to be instrumented by end-users for purposes such as ours. This made it difficult to build a measurement infrastructure for the game. However, some key features provided by the WoW client turned out to make this possible.

2.2.1 Lua

In the interests of allowing end-users to develop their own user interface extensions, WoW implements a user interface scripting environment for which users can write scripts to manipulate various aspects of the client. The scripting language used is Lua [15], a programming language that is popularly used as an embedded scripting language, particularly in game engines. WoW also includes a user interface markup language based on XML that allows users to create their own user interface elements such as windows and dialog boxes. A collection of scripts and user interface markups that work together to provide some feature are usually referred to as addons, or mods (modifications). Addons are loaded when the player logs into a WoW server. Note that addons are purely client-side scripts; they do not run on the server.

The primary purpose of WoW's Lua scripting engine is user interface manipulation. The API exposed to scripts thus consists mostly of interface-related functions. For example, there are functions that allow scripts to automate player actions like picking up objects or moving things about in the player's inventory. There are also a set of functions that are marked as *protected*;

these functions can only be used by addons which are signed by Blizzard. The purpose of this protection is to prevent third-party addons from accessing certain features such as controlling a player's movement in the game, which could be exploited to write scripts that entirely control a player. Only addons provided by Blizzard are able to do so.

There are also API functions that allow scripts to inspect the state of the game world. Of particular relevance to our goals are the functions that manipulate and query the *friends list*, and the functions that can send and process *who queries*. These are described shortly.

The API also consists of a set of events that correspond to in-game events. Addons can register a callback function to be called whenever an event is triggered.

Blizzard does not provide very comprehensive documentation of the WoW API. However, the online community site WoWWiki [3] contains a fairly complete list of functions and events in the game. This proved invaluable in implementing the experiments.

2.2.2 Friends List

Every player in the game can keep a *friends list*, which is a list of other players which the player considers *friends*. When player Alice adds player Bob to her friends list, Alice is able to know whether Bob is online, and what his current level and location are. Specifically, Alice is sent a notification whenever Bob's status changes, *i.e.* whenever he logs on, logs off, changes his level or changes his location.

The WoW API provides functions to add and remove friends from the friends list (`AddFriend`, `RemoveFriend`), determine the number of friends in the friends list (`GetNumFriends`), and obtain the current status information of a friend (`GetFriendInfo`). Additionally, every time a friend logs on or off, the `FRIENDLIST_UPDATE` event is triggered.

One important limitation of the friends list is that it can only contain up to 50 players.

2.2.3 Who Queries

WoW also provides a more generic player querying system, usually referred to as the *who* command (or dialog box). Players can construct queries of varying complexity, which are sent to the WoW server. The server will then reply with a list of player names that match the query. The query only matches players who are online.

A query consists of the following elements:

Name A string that matches if it occurs anywhere in a player's name.

Zone Matches all players in the zone with that name.

Race Matches all players with the race of that name.

Class Matches all players with the class of that name.

Guild Matches all players with the guild of that name.

Level Either a single number or a range. If a single number, it matches all players whose level is that number. If a range, it matches all players whose level is within that range (inclusive).

The query can contain any or none of these elements. Elements are combined via conjunction; a query for players in zone "Stormwind City" with level range "10-20" will return the players who are in Stormwind City *and* have a level between 10 and 20. An empty query matches all possible players.

The WoW server returns at most 50 player names that match the query, whether or not there are more matches. When there are more than 50 matches, the exact way the server decides which subset of player names to return is unknown; it does not seem to be random, nor have any discernable pattern. In particular, when multiple empty queries are sent over a short period of time, the results returned by each query are nearly identical. As such, empty queries cannot be used to effectively obtain a random sample of player names.

A client may not send too many queries at one time. The server throttles the number of queries it processes per client; if a client sends another query too soon after a previous one, it is ignored. Again, the nature of the throttling is unknown. Generally, it is possible to send one query every 3 seconds, any rate faster than this results in some queries being lost.

API functions are provided to send who queries (`SendWho`) and inspect the results (`GetNumWhoResults`, `GetWhoInfo`). An event (`WHO_LIST_UPDATE`) is triggered whenever the results of a query are returned by the server.

2.2.4 SavedVariables File

The WoW API also provides a feature that allows addon to save some state in between game sessions. Normally, variables created by addons exist only for as long as the player's game session is active; once the player logs out of the server, the Lua engine terminates the addon and all variables are lost. However, it is possible to register a set of variables whose contents will be saved just before the game terminates. The values of these variables will then be restored the next time the same addon is loaded.

WoW implements this feature by writing to disk a file consisting of the serialized contents of all saved variables. This file is typically referred to as the SavedVariables file, and is actually a Lua script that when run reinitializes the saved variables to have their previous value.

The SavedVariables feature is the *only* way a WoW addon is able to effectively write data to disk. This feature was exploited in our experiments as a crude communication medium between the measurement addons and management scripts that run external of WoW. In particular, it was used to transfer information gathered by the measurement addons into an external database for permanant archival.

2.2.5 Manager

A measurement manager was developed to facilitate the conducting of experiments. The manager had three main taks:

1. Launch the WoW client.

2. Ensure it successfully connects to the WoW server.

3. Collect the data gathered by the measurement addons and store them in a database.

At the core of the manager is a single Perl script, referred to as the *monkey* script, which performed the tasks outlined above. One of the most important tasks of the script was to *automate* the process of running WoW and entering the game.

2.2.6 Wine

WoW is a game built for the Microsoft Windows environment. However, we chose to run WoW in GNU/Linux using the Wine Windows API emulator [2] in order to make use of several features easily available in our chosen environment. Wine runs WoW very well, and is in fact used by many WoW players who prefer to play the game in a non-Windows operating system.

2.2.7 XTEST

The use of Wine to run WoW exposed the WoW client as an X Window System (X11) application. This allowed us to make use of the X11 XTEST extension library [12], which is a set of client and server extensions designed to allow the X11 server to be tested with no user intervention.

The XTEST extension provides a set of functions that can be called to insert X11 events into an X server's event queue. This creates the illusion (to an X11 client) that a user had performed the actions that caused the event, for example depressing a key on the keyboard or moving the mouse. It is thus possible to use the XTEST extension to automate an X11 application, such as the WoW client.

2.2.8 ImageMagick

In order to determine whether or not the WoW client was behaving as expected, a feedback mechanism was required. Since the client is designed to be an interactive application, the main form of feedback is via the graphical display output to the X11 server.

ImageMagick [21] is a suite of commandline utilities that allow for the manipulation of image data. The `import` program was used to obtain a screenshot of the WoW client, and the `compare` program was used to compare the obtained screenshot with a pre-generated reference screenshot. The extent at which the screenshots differed could then be used to infer whether or not the WoW client was in the state expected by the script.

While one might expect that two screenshots of the game in the same state (for example, waiting for the user to login) would end up to be identical, this did not turn out to be true. All of WoW's user interface is implemented on top of the OpenGL 3D graphics API. As a result, some of the colors of interface elements vary due to the presence of variable lighting. Other parts of the interface might be animated and thus result in a different static image dependent on the time at which the screenshot was taken.

The ImageMagick `compare` utility is able to produce a metric indicating the amount of difference between two images. By varying the treshold of the value that would be used to consider two screenshots identical, most of the subtle differences in screenshots could be taken into account. Additionally, the screenshots are restricted to only capturing a small portion of the screen, usually corresponding to a button or some other user interface element that is known to be unique to that game state. This was particularly necessary when checking to see if the client was logged into the game world, as during then the majority of the screen is displaying the dynamic game world.

2.2.9 Monkey Script

The monkey script is a program written in Perl, which makes use of a Perl module implementing the XTEST extension. An outline of the script's behavior follows.

Algorithm 1 Monkey script.

```
loop
    if Frostmane WoW realm is online then
        Launch WoW client.
        Wait for login screen.
        Input username and password, log in.
        Wait till client has entered game world.
        for 2 hours do
            Periodically send a "jump" command to prevent client from idling out.
            Check to see if the client is still in the game world.
        end for
        Cause WoW client to quit.
        Wait for client to quit cleanly.
        Retrieve contents of SavedVariables file, write to database.
    end if
end loop
```

Determining server status

Blizzard publishes an XML file containing the status of every WoW server. This is used to determine whether or not a server is online. If the server is not online, the monkey script waits till it is online before attempting to connect.

Waiting for client state

The current client state is determined by making use of ImageMagick as described above.

Input to client

Input is sent to the WoW client using the XTEST extension.

Measurement

Once the monkey successfully connects the client to the game world, the actual measurement addons will run and begin to collect data. The monkey script expects the addons to write *events* (different from the WoW API events) to an *event log*, which is a saved variable.

Anti-idling

If a user does not interact with the WoW client for more than several minutes, the WoW client disconnects from the game server. It is unclear whether this is enforced by the WoW server, or by the client. Nevertheless, the feature cannot be disabled. As a result, the monkey script is required to send some form of input to the client every so often in order to prevent “idling out”. This is done by sending a jump command (corresponding to a single space character) to the client via XTEST every 2 minutes.

Quitting

The contents of the event log as written to by the measurement addons is stored in memory. In order for it to be written to disk, the WoW client must disconnect from the game server. The monkey script does this every 2 hours to prevent too much data from being stored in memory before being flushed to disk. It also causes the client to fully quit, instead of merely logging off; this is purely for convenience and has no effect on the actual automation process.

Retrieving Data

The monkey script extracts the contents of the saved event log from the SavedVariables file, and then inserts each event it finds into an external database.

Event Log

The exact schema of the event log differs from experiment to experiment. These will be discussed in Sections 2.3, 2.4 and 2.5.

Exception Handling

The WoW client is a complicated piece of software, and it would be difficult to write a program that was able to react to every state change that might take place. Instead, the monkey script continually monitors the state of the client during measurement, and once it detects that the client is in an unknown state (it might have disconnected from the game world for example), it will immediately terminate the client process, cleanly if possible, and restart the client so as to resume measurement. In practice, the monkey script is capable of handling most client state changes.

2.2.10 Measurement Interval

A *measurement interval* is defined to be the period from the monkey script starting the WoW client and causing it to enter the game, to the monkey script causing the WoW client to quit. It is thus a period over which the measurement addons were running continuously without disruption.

2.2.11 Measurement Outages

There were several sources of outages throughout the measurement period.

As previously mentioned, the WoW client had to be shut down every 2 hours in order to flush the collected measurements to disk. This was a planned outage, and on average each outage lasted for slightly more than a minute.

Blizzard will occasionally bring some of the WoW servers offline, typically for maintenance purposes. Such outages typically take place on Tuesday mornings, and last for a few hours. Originally, this happened every Tuesday, but towards the end of December 2006, changes to the server infrastructure allowed Blizzard to perform most maintenance online and thus reduced the occurences of the outages. The monkey script handled such outages by making sure the server's status is online before attempting to connect.

There were also occasional outages due to server-side errors. From the perspective of the client, these were no different from the maintenance outages described above.

The other major outages were due to patches to the game client. At two points during our measurement study, patches were released that were incompatible with the Lua measurement addons. In one case, the patch updated the in-game Lua interpreter to version 5.1, from version 5.0. This caused problems with the addons because they were not written for the newer version of Lua. In another case, an expansion set (The Burning Crusade) was released, which changed the set of possible races and increased the highest possible level. This caused problems with parts of the addons that had to be fixed.

As a result of the outages, the long-term experiment's dataset is partitioned into 3 periods. This will be elaborated upon in Section 2.3.

2.3 Long-Term Measurement Experiment

The first experiment that was conducted was the long-term measurement experiment. This experiment monitored the behavior of a set of 1,100 players over 71 days (nearly 3 months). The main goal of the experiment was to obtain an understanding of the behavior of a large number of players over a long period of time.

2.3.1 Obtaining Player Set

In order to obtain the sample set of players that were to be monitored, a WoW addon (hereafter referred to as NameGrabbber) was written to collect as many player names as possible from the game server.

NameGrabber makes use of WoW's who query system, and sends out a series of who requests every few minutes during a poll. For each query that is sent out, NameGrabber determines whether the query was complete (all matching players were returned) by observing the size of the set of player names returned. If the set size was equal to 50, then there would be a chance that the acutal set of matching players was larger than 50, but the WoW server had truncated the returned set.

Incomplete queries are handled by performing a *partition* operation on the query. As described in Section 2.2.3, who queries are made up of several elements, each element adding an additional constraint to the set of players that match the query. A query that is incomplete can thus be broken down into several new queries, each of which is more constrained than the original, and will thus match less players. For example if a query for all players with levels between 10 and 20 is incomplete, it can be partitioned into two queries for players of level between 10 and 15, and players of level between 16 and 20. If all the elements in a query are maximally constrained (*e.g.* the level element matches a single level), then new elements can be added to the query to increase the constraint. An enumeration of all possible races and classes in the games allows for constraining on those elements.

NameGrabber partitions queries first by level, then race, and finally class. If a query is maximally constrained on the level, race and class elements, yet it still incomplete, then as a final attempt NameGrabber partitions such queries by adding a name element that matches any name containing a particular letter. The set of letters used is: a, e, i, o, u, t, s, d, r. Note that this method of partitioning the query yields sub-queries that overlap; many names may contain both the letter 'i' and the letter 'e', and as a result will appear in both sub-queries. Also, there might be some names which do not contain the letters in the set at all, and would thus be missed; this however is unlikely since few names contain no vowels. If such queries still fail to be complete, then NameGrabber just accepts the set of names returned. The zone element was not used in the partitioning because the set of possible zone names was too large.

NameGrabber initializes each poll by sending 12 queries, one for every 5 levels in the game (NameGrabber was used before the Burning Crusade expansion, which increased the level range from 0-60 to 0-70). The queries are sent using the SENDQUERY function, as detailed in Algorithm 2, where PARTITION is a function that partitions queries using the method described above.

This method for collecting names from WoW server was based on a very similar technique used in the CensusPlus [1] addon, which gathers player information in order to publish a census.

Algorithm 2 NameGrabber.

```
function SENDQUERY($q$)
    Send out query $q$.
    $S \leftarrow$ set of players returned by server.
    if $|S| \geq 50$ and $q$ is partitionable then
        $Q' \leftarrow$ PARTITION($q$)
        for all $q' \in Q'$ do
            return $S \cup$ SENDQUERY($q'$)
        end for
    else
        return $S$
    end if
end function
```

NameGrabber was used over more than a week to obtain a set of 9,530 names. Out of these, 1,100 were randomly selected to form the final sample set. According to the CensusPlus [1] census, the server our measurements were conducted on (Frostmane) had slightly over 10,000 Alliance faction players, which was slightly more than the amount NameGrabber found. Thus the set of names our random selection was obtained from is close to the entire population of the game world.

2.3.2 Polling

Players were monitored by polling the WoW server every 5 minutes to determine whether a player was online, and if so, the player's level and current location in the game. Polling was implemented using the friends list.

Algorithm 3 Long-term measurement experiment poll.

```
Let $P$ be the set of players to be monitored.
$B \leftarrow$ set of batches obtained from $P$.
**for all** $b \in B$ **do**
    Clear the friends list.
    Add all the players in $b$ to the friends list.
    Wait for 5 seconds.
    Obtain player information from the friends list.
**end for**
```

Algorithm 3 describes the polling process that takes place every 5 minutes. Since the friends list can contain at most 50 players, the set of players to be monitored (denoted by P in the algorithm) was partitioned into sets of 50 players, which are referred to as *batches*. Each batch was then added to the friends list. Due to limitations in the way the Lua interpreter works, and also to allow the WoW client time to retrieve friend information from the server, an interval of 5 seconds was allowed to elapse. The status information of each player in the friends list was then obtained, and written to the event log.

A poll took about 110 seconds, mainly due to the 5 second delay in the polling process. Polls were spaced 5 minutes apart, as opposed to having them occur immediately after each other, in order to keep the amount of traffic being sent to the game server from being too large.

2.3.3 Captured Information

Each poll recorded the online status of every player, and for players that were online, they also recorded down the location and level of the player. The exact time at which the player information was obtained was also recorded, and all the information was logged as either an online or offline event.

2.3.4 Missing Players

Occasionally, the measurement addon was unable to add a player to the friends list. Attempting to do so resulted in the WoW server returning an error message stating that the player was not found. The most likely reason for this was that the player's account had been deleted, and thus the player no longer existed in the game world. Another cause could simply be a case of a server error, or some server-side reason that is unknown to us. Whenever the measurement addon failed to add a player to the friends list, it records down an event stating that the player was *missing*.

2.3.5 Event Log Schema

Field	**Description**
id	Unique identifier
type	Type of event
player	Player associated with the event
ts	Unix timestamp of event
zone	Location of player in the game
level	Player's current level

Table 2.1: Long-term experiment event log schema.

The event log schema used by the long-term measurement addon is shown in Table 2.1. The different types of events are listed in Table 2.2. The player field of an event was only used by `ONLINE`, `OFFLINE` and `MISSING` events, and the zone and level fields were only used by `ONLINE` events.

The `START`, `STOP`, `POLLSTART` and `POLLSTOP` events were inserted by the measurement addon when appropriate.

Name	Description
`START`	Start of a measurement interval.
`STOP`	End of a measurement interval.
`POLLSTART`	Start of a poll.
`POLLSTOP`	End of a poll.
`ONLINE`	Player is online.
`OFFLINE`	Player is offline.
`MISSING`	Player is found missing.

Table 2.2: List of types of the long-term experiment event log.

2.3.6 Periods of Measurement

Start Date	End Date	Length (days)
Nov 15 2006	Dec 4 2006	19
Dec 10 2006	Jan 3 2007	24
Feb 5 2007	Mar 5 2007	28

Table 2.3: Dataset periods for the long-term experiment.

The long-term measurement experiment ran from November 15th 2006 to March 6th 2007. As mentioned previously, unplanned outages resulted in a partitioning of the measurement data. Table 2.3 lists the start and end dates of the three resulting datasets. As can be seen, each period was at least 2 weeks, and the longest period was nearly a month. In the results section, we often refer to the last dataset (Feb 5 2007 to Mar 5 2007) as results for the month of February.

The other outages that occured during the long-term experiment were at most a few hours long, and so we chose not to use them to further partition the dataset. Instead, post-processing scripts that made use of the raw event data took note each time the interval between a `STOP` and the next `START` event was longer than 15 minutes. Whenver this happened, it would be assumed that the previous poll was not a good indication of the state “just before” the following poll. For example, if a poll indicates player P is online follows a poll that also indicates player P is online, but more than 15 minutes elapsed between the polls, then we do not assume P was online throughout the 15 minutes because P could have left the game and returned within the time interval.

2.3.7 Accuracy and Errors in Measurement

One of the goals of the long-term experiment was to monitor a large group of players. Due to the limited ability of the measurement tools available to us (who queries and the friends list), broadness in measurement had to be achieved at the expense of accuracy, particularly in terms of the timing of events. Since the polling interval was 5 minutes, the time between two measurements of a single player was also 5 minutes[2]. Thus the recorded times at which a player entered or left the game (as determined from the raw online/offline events) had errors of 5 minutes, and as a result the recorded sesion times had errors of 10 minutes.

However, the advantage of the long-term experiment is that in monitoring a large number of players, the results obtained are more likely to reflect the entire population of the game. The sample size of 1,100 players was nearly 11% of the entire population (as determined by the CensusPlus census).

An important thing to note is that ultimately the accuracy of this experiment (and of the others as well) depends solely on the accuracy of the methods provided by the WoW client to determine player status. If for example the friends list was prone to errors, reporting a player as being offline when he was really online, then the data presented in this thesis is likely to be invalid. Since the inner workings of the friends list and the who query system are proprietary and not known to us, only careful experimental validation will be able to determine their accuracy,

[2]The time interval was at times less than 5 minutes, particularly in between the last poll conducted before the client was restarted, and the first poll after the restart. Restarting the client took only a bit more than a minute.

and this remains important future work.

2.4 Location Measurement Experiment

The location measurement experiment was designed to characterize the nature of player movement within game areas. Several locations (referred to as *zones* in the game) were chosen and continually monitored for players entering and leaving each location.

2.4.1 Chosen Locations

Location	**Description**
Stormwind City	Busy central city.
Wetlands	Transportation hub.
Stranglethorn Vale	Transportation hub, quest area.
Silithus	Remote.
The Barrens	Battleground.
Burning Steppes	Remote, along popular travel route.

Table 2.4: List of game locations that were chosen for measurement.

6 different locations in the game were chosen for measurement. They are listed along with a brief description in Table 2.4. While an attempt was made at choosing locations that posessed certain specific qualities, zones in WoW are typically large areas that contain several different cities/towns. As such, attempting to classify an entire zone as say a transportation hub is not entirely accurate.

The rationale behind choosing different locations was to observe the similarities and differences between the following generic kinds of areas:

- *Busy City*: A location where many players will gather to socialize, purchase or sell items, and perform other “non-fighting” activities.

- *Transportation Hub*: A location where players will go to in order to travel to other parts of the world, or arrive at from other parts of the world.
- *Battleground*: A location where players are mainly taking part in large-scale fights.
- *Remote*: A location that players generally do not travel to.

Busy cities are expected to have a large number of players. Transportation hubs will likely experience high levels of churn, since players arrive at such locations only to leave for other locations. The same is expected for the Burning Steppes location, because it lies along a frequently-used travel route.

There was particular difficulty in choosing an appropriate location as a representation of a battleground. In WoW, the areas where players commonly fight with each other (*i.e.* members of opposing factions) are usually *instanced*, which means that the server creates a private copy of the location for each group of players that wishes to enter the location. Thus, if 1000 players are reported to be in an instanced location, in reality there may be 10 copies of the location, with 100 players in each location. Measuring an instanced location thus yields inaccurate information.

As a result, the location chosen as a battleground was one that contained a particular town (The Crossroads) that was reported to often have large fights occuring. The town (and the whole of the location) is controlled by the opposing faction to the one that was being measured; thus any characters found in the area were likely to be there to engage in fights.

2.4.2 Polling

A poll-based monitoring system was used for the location experiment. Polls were conducted every 2 minutes, and attempted to obtain the details of every player found in the location.

The polling process used is described by Algorithm 4. The SENDQUERY function as implemented by NameGrabber described in Section 2.3.1 was used to find as many players as possible within each location. The exact time taken by each poll varied, depending on how many players there were in each location.

Algorithm 4 Location measurement experiment poll.

```
Let $L$ be the set of locations to be monitored.
for all $l \in L$ do
    $q \leftarrow$ a query with the zone element set to $l$.
    $P \leftarrow$ SendQuery$(q)$
    Record player information found in $P$.
end for
```

2.4.3 Captured Information

For each player found in a location, the player's name and level was recorded, along with the time at which the player was found.

Note that for a player to be found in a location, that player must necessarily be online.

2.4.4 Event Log Schema

Name	Description
`START`	Start of a measurement interval.
`STOP`	End of a measurement interval.
`POLLSTART`	Start of a poll.
`POLLSTOP`	End of a poll.
`ONLINE`	Player is in a location being monitored.

Table 2.5: List of types of the location experiment event log.

The location experiment uses an event log schema identical to the long-term experiment, listed in Table 2.1. Table 2.5 lists the event types used by the location experiment, which is basically a subset of those used by the long-term experiment. The `START`, `STOP`, `POLLSTART` and `POLLSTOP` are all used in the same was as they were in the long-term experiment, with the measurement addon inserting them as appropriate. An `ONLINE` event is inserted for every player that is found in one of the locations being monitored.

2.4.5 Period of Measurement

Start Date	**End Date**	**Length (days)**
Mar 6 2007	Mar 27 2007	21

Table 2.6: Dataset period for the location experiment.

The period of measurement for the location experiment is listed in Table 2.6. There were no noteworthy outages during the entire period, thus we take the data to represent a single contiguous measurement period.

2.4.6 Accuracy and Errors in Measurement

Since the location experiment uses a polling mechanism, it can only be as precise as the polling interval of 2 minutes. The same caveats as in the long-term experiment apply.

Players take time to move through the game world. For many players, it takes substantially more time than 2 minutes to traverse an entire location, *i.e.* to enter from one end and exit the other end. Thus the 2 minute polling interval might be sufficiently fine-grained to detect all players who entered each location. However, if a player enters and leaves a location along the same border, such behavior will likely be missed by the location experiment. For example, a player who enters a particular location, changes his mind about being in that location, and immediately leaves the way he came, will not be detected.

2.5 Detailed Measurement Experiment

The large polling interval in the long-term experiment meant that metrics with small values (in the order of seconds or minutes) could not be collected from the resultant data. Thus, an additional experiment similar to the long-term experiment was conducted, but with a much finer-grained level of measurement. This was achieved by compromising the set of players being monitored.

2.5.1 Sample Player Set

A set of 50 players was obtained via random selection from the set of all players found throughout the location measurement experiment. This constrained the set of players to only those who passed through any 1 of the 6 locations at any point during the experiment, but based on the popularity of the locations, particularly Stormwind City, most game players were likely to have been detected. In addition, the number of names detected was close to that of the reported number of players in the game.

2.5.2 Monitoring

Unlike the previous two experiments, the detailed experiment did not use a polling mechanism to determine player status. Instead, it made use of the in-game friends list, which should report if a player's status changes nearly instantaneously.

The measurement addon added the 50 players into the friends list, and recorded an event each time any player in the list changed status, consisting of the player's online status, level, and location in the game.

2.5.3 Event Log Schema

Table 2.7 lists the event log schema for the detailed experiment, and Table 2.8 lists the event types used. `START`, `STOP` and `MISSING` events are used in the same way as in the long-term experiment, and the `STATUSCHANGE` event corresponds to a player's change of status. Only `STATUSCHANGE` events made use of the online, zone and level fields.

2.5.4 Period of Measurement

Table 2.9 lists the period over which the detailed experiment took place. Again, there were no significant outages and thus the entire period is assumed to be contiguous.

Field	Description
id	Unique identifier
type	Type of event
player	Player associated with the event
ts	Unix timestamp of event
online	Player’s online status
zone	Location of player in the game
level	Player’s current level

Table 2.7: Detailed experiment event log schema.

Name	Description
`START`	Start of a measurement interval.
`STOP`	End of a measurement interval.
`STATUSCHANGE`	Player’s status has changed.
`MISSING`	Player is found missing.

Table 2.8: List of types of the deailed experiment event log.

2.5.5 Accuracy and Errors in Measurement

The detailed experiment did not use a polling mechanism and should thus have been able to obtain very accurate measurements of the time a player’s status changed. The exact amount of error in the measurement is dependent on the friends list, and is thus unknown.

Due to the small sample size of 50, it is difficult to claim that the results from the detailed experiment are a good representation of the general population.

Start Date	**End Date**	**Length (days)**
Mar 27 2007	Apr 24 2007	28

Table 2.9: Dataset period for the detailed experiment.

2.6 Motivation

In order to better understand the significance of the metrics being measured, and the consequences of the results, this section presents a set of hypothetical game frameworks which will be used to demonstrate the effects of dynamic player behvaior.

2.6.1 Prototype Game

Fundamentally, the purpose of a framework is to provide a set of services upon which games can be developed. For purposes of simplification, we consider the games which will make use of the hypothetical framework to have the following properties:

- The game world consists of objects which have properties.
- Players are objects.
- Objects can only interact with other objects that are nearby, given some notion of distance.
- Every object has a *think function* which is executed continuously. The think function may change the properties of the object or of other objects (*i.e.* interacting with them).
- The game is divided into areas called *locations*.

In addition, objects can be classified into four different categories:

Ephermeral An object that exists in the game world for a short period of time.

Persistent Static An object that exists in the game world for a long period of time, and its properties never change.

Persistent Dynamic An object that exists in the game world for a long period of time, and its properties may change.

Player-bound An object that exists in the game world *only* when the player it is associated with is logged in.

For example, summoned creatures (that disappear after a while) are ephermeral, a mountain is persistent static, an AI-controlled monster is persistent dynamic, and the player itself is player-bound. This categorization turns out to be important when considering how objects are managed by the framework.

A node is a member of the network that makes up the game framework. Nodes can host multiple objects, and can execute the think functions of those objects.

2.6.2 Framework Requirements

We can describe the services that a framework must provide in terms of three components: the *routing* component, the *discovery* component, and the *object management* component.

Routing

Frameworks must provide a means for nodes to contact other nodes. These nodes could correspond to players, or to servers. In a client-server framework, the routing component is simply the communication channel between the client and the server. In a peer-to-peer framework, something like a distributed hash table might be used instead.

Discovery

Objects need to know what are the objects that are nearby, in order to interact with them. The discovery component allows nodes to determine this information, as well as information on how to contact the node that the object required is on. Quite often, DHTs can also serve as the discovery component, as the node just has to lookup the key corresponding to its current location.

Object Management

The most critical task of a framework is to allocate objects to nodes, and make sure objects are kept within the system. The object management component decides which objects are hosted on which nodes (an *object placement strategy*), establishes the rules under which objects may interact (read/write) other objects, and ensures that all objects in the game are always available when required.

Effectively, the object management component manages an *object store*. The availability requirement is especially important with regards to persistent objects, because the period over which these objects must be available is long. A distributed object store might require such objects be replicated in order to increase availability. This leads to additional concerns with regards to whether the object is static or dynamic; if a dynamic object changes a property, this must be reflected in all replicas of the object.

It is often beneficial to have the object placement strategy take into account the positions of the objects in the game world. It is advantageous to place objects that are nearby each other in the same node, as this reduces communication overhead.

2.6.3 P2P Framework

The P2P framework is a fully distributed system where every node corresponds to a player. It uses a DHT to provide routing and discovery, and objects are evenly distributed amongst nodes using some selection mechanism. Persistent objects are replicated to increase availability, and when a change is made to a dynamic object, it is reflected amongst all replicas as soon as possible. Player objects are always hosted on the associated node.

2.6.4 Federated Framework

The federated framework consists of a large collection of server nodes which player clients connect to. Routing between servers is statically assigned, and clients statically determine the server

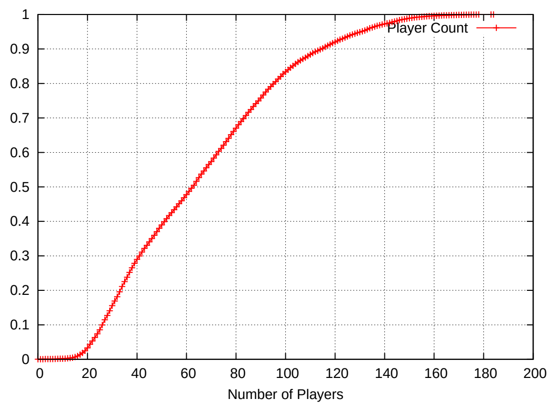

Figure 2.1: Player count.

to connect to based on their current location in the game. Each server corresponds to a *location*, and all objects within the location are hosted by the server, including player objects.

2.7 Results

The results of the measurement study are shown here. The main focus is on how the metrics we measured influence the design of distributed games; as such less emphasis is placed on modeling.

2.7.1 Player Count

One of the most basic metrics of any system is the number of participants at any given time. In our measurement study, this is referred to as the *player count*; how many players are online at some time.

To determine the player count in the long-term experiment, the total number of players found online at each polling period was determined. The resulting distribution is shown in Figure 2.1. which has a median of 63 players (5.7%). There were never more than 200 players online at the same time, which given our sample size of 1,100 players is less than 18%.

This is not unexpected, and is a fairly common property of subscriber sets. Many systems take advantage of the fact that the number of active subscribers at any time is usually low, and

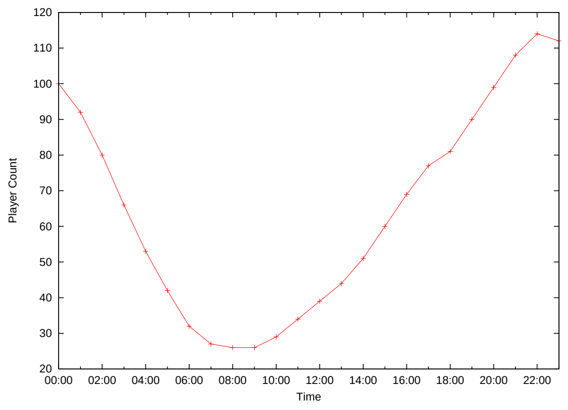

Figure 2.2: Relationship between player count and time of day.

World of Warcraft is likely to be no exception.

Interestingly, only 993 players in total were recorded to have *ever* appeared in the game, about 90% of the total population being monitored.

Player count is expected to vary in a diurnal pattern, with more players logging in at night after they return from work (or classes). [9] reports such a pattern in player population of three popular games, as well as a weekly pattern. A plot of player count versus the time of day the count was obtained yielded the graph in Figure 2.2, which clearly demonstrates time-of-day effects. Possible day-of-week effects were also investigated, but no substantial patterns were found, *i.e.* people log on no more during the weekends than they do during the weekdays, in constrast to what was reported in [9].

Summary

The number of active players at any time is low compared to the total number of players in the system. Player count varies according to time of day in a diurnal pattern, but no weekly patterns were found.

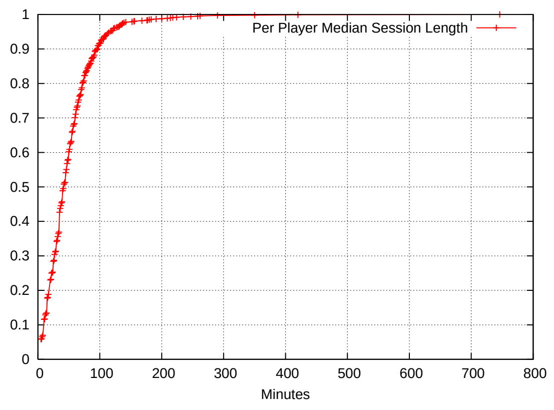

Figure 2.3: Median session length for each player.

2.7.2 Session Length

A *session* is defined as the period between a player logging in and logging out as a particular character in a particular realm. The session length is then a measure of how long a character is logged into a realm.

Since a player can spend some time playing as one character and then change to another, sessions are not equivalent to the period a player spends playing the game. However, since we are unable to determine the relationship between players and characters, we work with the assumption that session length is roughly equal to the player's *uptime*.

Session length is important in a distributed system because it indicates how long a particular node is online and able to participate in the system. For example, in the P2P framework, a node with a very short session length might be unsuitable for hosting a persistent object. Having too many short-lived nodes would severly impact performance.

Additionally, session length has some ramifications on DHT efficiency. [23] shows that several popular DHTs begin to degrade in performance when the median session length gets shorter.

Figure 2.3 shows the distribution function of median session lengths for each of the 1,100 players monitored in the long-term experiment. The median of the median is 42 minutes.

The distribution function of all session lengths recorded shown in Figure 2.4 is similar in

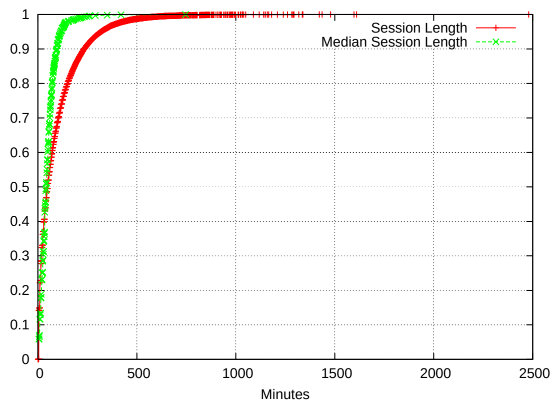

Figure 2.4: Comparison of session length and per-player median session length.

shape. However, it is far more long-tailed, and beyond the 50th percentile it begins to shift towards the right. This might suggest that the longer sessions tend to be distributed over all players; most players will occasionally play the game for a longer period of time, as opposed to there being a group of players who exclusively play for a long period of time.

The median session length is 50 minutes. There were 4 sessions that lasted longer than a day [3], and 876 that lasted more than 8 hours. In contrast, the player with the longest median session length had a median session length of around 12 hours, and the next longest median session length was 7 hours long.

The median of the mean session lengths of each player is 70.7 minutes, and the median standard deviation is 74.7 minutes. This validates the earlier hypothesis that longer sessions are distributed over all players; the high standard deviation indicates that players have very short as well as very long sessions, and a mean larger than the median suggests a skewed distribution.

Different measurement studies of P2P filesharing systems have produced differing results for median session lengths. These range from a few minutes [14] to close to an hour [23]. An average session length of 42 minutes falls within this range and is thus probably not that different from a P2P system. This in an interesting result, because the nature of filesharing applications is

[3]Out of the 4, 2 were from the same player, and occured slightly more than 2 weeks apart. In fact the top 10 longest sessions originated from a set of 5 players.

very different from that of an online game; a game player is almost always an active participant throughout the session, whereas a person downloading a file is likely to leave the filesharing application running while she does something else.

Results from [25] suggest that session lengths in P2P datasets (specifically BitTorrent) are not heavy-tailed, but best described by Weibull distributions. Analysis of the session length data collected by the long-term experiment gives similar results. A Weibull distribution with shape 0.84 and scale 4914.4 yields the best fit to our data.

In [23], experiments were conducted on several DHTs to observe their performance when sessions lengths were varied. Their results show that most DHT systems can cope relatively well with median session lengths of around 40 minutes, and thus making use of a DHT to implement the routing layer of a game engine is likely to be acceptable.

An important implication of the fact that long sessions are more evenly distributed amongst players is that designs which rely on super-nodes may not perform as well under a game workload as opposed to say a filesharing workload. For example, consider a system which attempts to place frequently accessed data on nodes with long session lengths. In a filesharing environment, there might be users who leave their filesharing application constantly running in the background, and are thus good candidates for hosting such data, since their session lengths are always long. In a game, however, players might occasionally have a very long session, and at other time stay in the game for only a few minutes. Thus, there might not be any node that regularly chalks up long sessions.

That being said, the players in the tail of Figure 2.3 could be considered as suitable super-nodes. Additionally, there is some level of predictability in the length of future sessions based on previous sessions. Figure 2.5 show a plot of the median session lengths of players in the long-term experiment over the first week of measurement, versus the final median session length as shown in Figure 2.3. The Kendall's τ correlation coefficient is 0.76, which indicates positive correlation. Thus, a player whose sessions are long on average is likely to continue having long

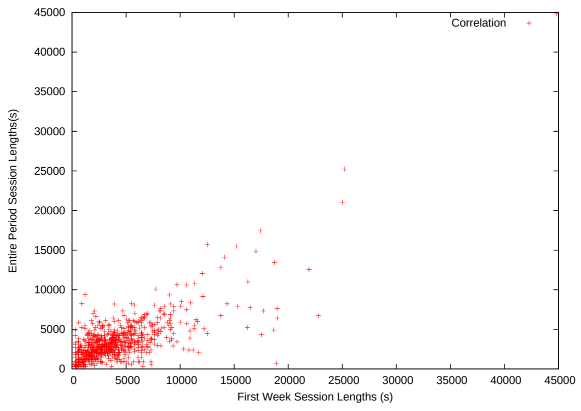

Figure 2.5: Predictability of future session lengths.

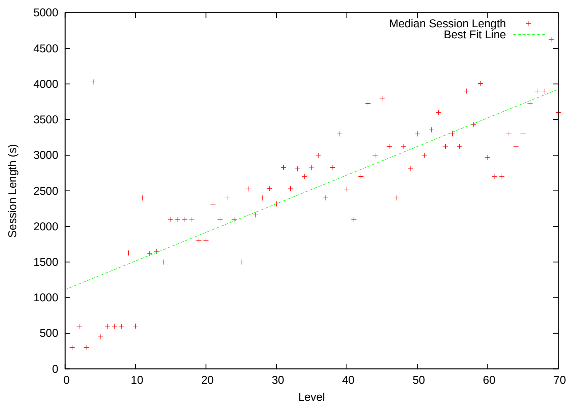

Figure 2.6: Relationship between player level and session lengths.

sessions.

The slope of the linear regression line of Figure 2.5 is 0.575, which suggests that in general players had shorter sessions overall than they might have had over the first week.

Another parameter that seems to influence session length is the level of the player at the beginning of the session. Figure 2.6 shows a plot of player level versus session length, and a linear regression line of the data. The Kendall's τ correlation coefficient for this dataset is 0.81, corresponding to a high degree of correlation. Thus, a higher level player is likely to have longer sessions. If we also consider player level to be an indicator of the *age*[4] of the player, then this

[4]How long since the player's character was created, *i.e.* how long the player has been a player of the game.

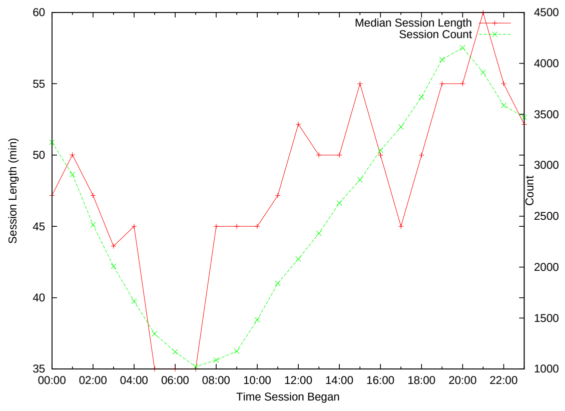

Figure 2.7: Relationship between session length and time session began.

suggests a relationship between age and session length as well.

Finally, Figure 2.7 shows a pattern in the variability of the session length depending on the time of day the session started. In general, the longer sessions all begin in the evening and at night. Sessions that begin in the early morning tend to be short, particularly from the period from 5-7am. Thus, the length of a session could be predicted based on the time the session is started. In addition, similar patterns can be observed in the sessions of individual players; players tend to start long sessions at around the same time each day. This is logical since most people have regular daily schedules and as such are likely to begin playing the game at roughly the same time (for example after dinner).

Also in this graph is a plot of the number of sessions observed. A very clear diurnal pattern is evident, indicating that most sessions begin at night. This agrees with the pattern visible in Figure 2.2. The general trend emerging from the two graphs is that most players log on at night, and tend to have longer sessions at night. Conversely less players log on in the period around noon, and those who do have shorter sessions.

In light of this trend, the P2P framework will benefit from the large number of long sessions available during the night. Conversely, it might face difficulties during the day; each node available during the day would have to host a very large number of objects. Fortunately the low player count in the day also means that there will be less requests for objects, but nevertheless there is likely to be a negative impact on gameplay during the day.

Since many people enjoy playing games with a group of friends, one would also expect that there will be groups of players who have similar session lengths, because they had made arrangements to play at the same time.

Stutzbach and Rejaie [25] describe a pitfall in characterizing churn, where long sessions are not properly accounted for because of the limited size of the measurement period. For example, all sessions longer than our measurement period would never be recorded, and all those that were longer than the time between the start of the session and the end of the measurement period would similarly be lost. The former is not a problem in our experiments, because sessions are rarely longer than a day, as compared to our measurement period of several weeks/months. We investigated the effect of the latter by filtering out all sessions that began after 2 days before the end of our measurements. Comparing the filtered dataset to the unfiltered one yielded negligible differences.

Summary

Sessions are generally around 42 minutes long. The range of session lengths for a single player is large, and the range of session lengths in the system is even larger. The median session length of 42 minutes is still in the acceptable range of most DHTs, and so using a DHT in a game framework is viable. There are also several parameters that affect session length, which could be leveraged to predict the length of a session. The length of sessions per player over the first week of measurement was positively correlated with the overall session length. Higher level players will also tend to have longer sessions. Finally, sessions vary according to the time of day, with longer and more sessions at night than in the day.

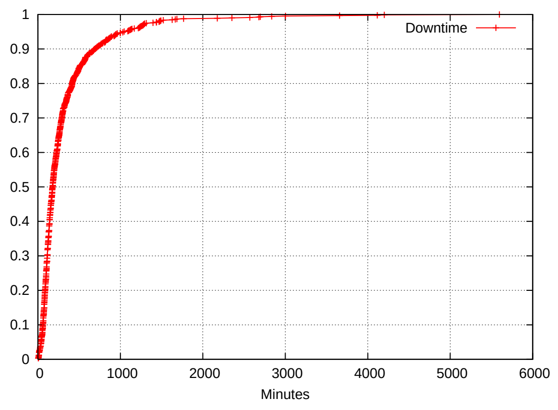

Figure 2.8: Median downtime for each player.

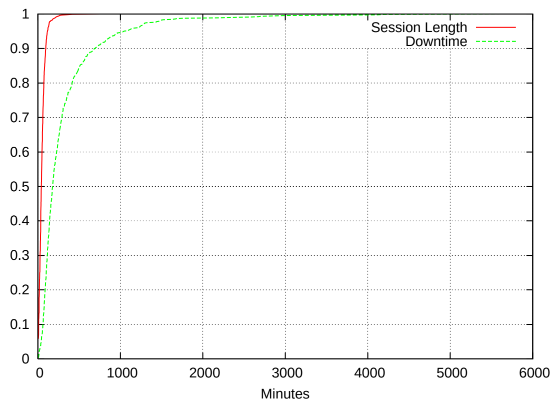

Figure 2.9: Median downtime and session for each player.

2.7.3 Downtime

Closely related to the session length is downtime. This is the time between two consecutive sessions by the same player (i.e. the time spent offline), and it provides an indication of the time it will take an offline node to return online. If the average downtime for a node in the P2P framework is long, then the time during which the data hosted on that node is unavailable is similarly long. Systems where nodes have long downtimes must thus perform more data replication to ensure a minimum level of availability.

The distribution of median downtimes of each player is shown in Figure 2.8. Figure 2.9 compares median downtimes to median session lengths, and shows that downtimes are longer

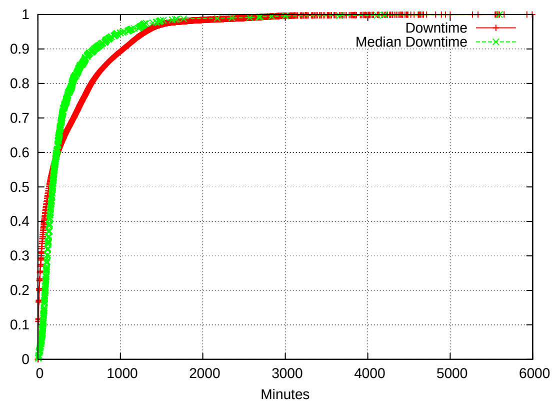

Figure 2.10: Comparison of downtime and median downtime.

than session lengths. This is expected since the average player is likely to spend less time in the game than out of it.

Surprisingly, the median of the median downtimes is 179.38 minutes, which is nearly 3 hours. One would imagine this value be much higher, at the very least to accomodate for 7-8 hours of sleep, not to mention around 8 hours of work. Upon further analysis, the median of the means is 6.86 hours, and the standard deviations of downtime for each player turn out to be very large, with a median over all players of around 8.34 hours. This is similar to the values obtained for session lengths. Thus, a possible explanation for the low median downtime is that players tend to have many sessions held close to one another, resulting in many short downtimes, and also a few sessions held very far apart, resulting in the much larger mean value. The period of downtime when the player is asleep or at work would then correspond to the fewer longer downtimes, which also suggests that a player has multiple sessions every night, held close together.

A comparison of downtime versus the per-player median downtime is shown in Figure 2.10. The distribution of the non-median downtime is much more heavily-tailed, which again suggests the longer downtimes are distributed amongst players.

The longest observed downtime was about 4 days. It is important to note here that downtime is taken to be the time *in between* two consecutive sessions; thus if a player logs out of the game at some point and does not log back in till after the long-term experiment concluded, the

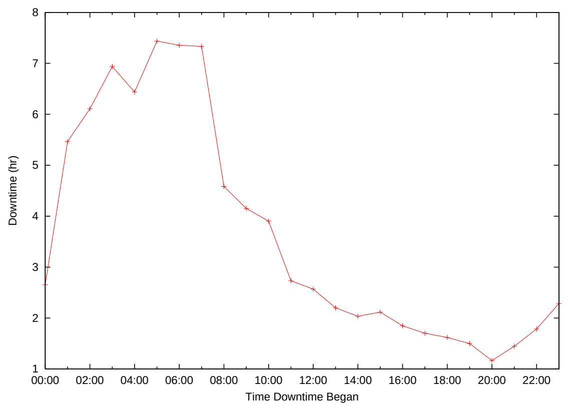

Figure 2.11: Relationship between downtime and time of day.

period from the final logout event to the end of the experiment is not recorded as downtime. Thus, the method of determining downtime used in this thesis suffers from a bias towards shorter downtimes, as was previously described in Section 2.7.2 with respect to session lengths. Again, any significant downtimes that are missing from the results are all extremely long, so long that we might as well consider the player in question to have permanently left the game. Compensating for the downtimes that might not have been recorded towards the end of the experiment also yielded negligible differences.

Short downtimes bode well for systems similar to the P2P framework. Since the time period over which a node is not accessible is small, persistent objects hosted on player nodes are more likely to be available. Of course, since session lengths are typically less than an hour, nodes are still going to be offline most of the time. The availability of a node, which will be discussed in the next section, focuses on how often a player is online.

The relationship between downtime and the time of day as shown in Figure 2.11 is the inverse of that in Figures 2.7 and 2.2, as expected. Nodes that leave in the early morning will not return for several hours, and nodes that leave in the evening will be back in a few hours.

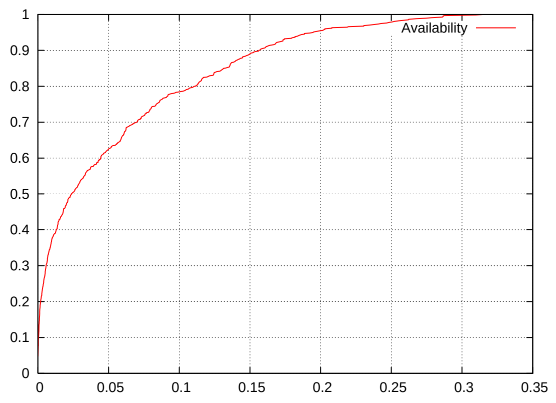

Figure 2.12: Availability of nodes over February.

Summary

The distribution of downtimes is similar to that of session lengths, with each player having both very short and very long downtimes. However, downtimes are generally longer than session times. Time-of-day effects are present in the downtime distribution, with the longest downtimes occuring in the early morning.

2.7.4 Availability

Availability is a measure of how often a node is online and available as opposed to offline. The availabilities of all players in the long-term experiment over the month of February (specifically from Feb 5 2007 to Mar 5 2007) were determined by finding the ratio between the total amount of time spent online and the total amount of time elapsed (28 days). The distribution is shown in Figure 2.12.

The median availability is 0.024, *i.e.* around 2%, about 30 minutes out of an entire day. The highest recorded availability was 31.5%.

An availability of 2% is not by any means high, and it indicates that the P2P framework must agressively replicate persistent objects in order to provide high availability. If we assume that sessions are independent, then at least 50 replicas of an object are required in order to achieve

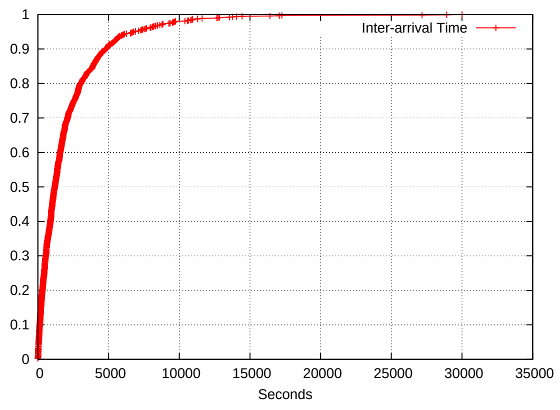

Figure 2.13: Inter-arrival times.

100% effective availability. This number increases if sessions are not found to be independent, which will be discussed later.

2.7.5 Lifetimes

2.7.6 Inter-arrival Times

Inter-arrival time is the time between two consecutive login events. This metric is useful in characterizing how frequently new nodes enter the system. A system where the inter-arrival time is very short must ensure that any overhead associated with adding new nodes into the system is low. If not, the system might be overwhelmed by the constant stream of new users. For example, the P2P framework should use a DHT with a low join overhead if inter-arrival times are short.

An important aspect about inter-arrival times is that they are likely to get shorter when the population increases. Thus, when measuring inter-arrival times over a large population such as with the long-term experiment, we are likely to see inter-arrival times on the order of seconds. However, since the long-term experiment has a polling interval of 5 minutes, it is not suited for gathering statistics at such a fine granularity. We thus focus on the results obtained by the detailed experiment instead.

Figure 2.13 shows the distribution of inter-arrival times as gathered from the detailed experiment. The median of the distribution is 1189s, around 20 minutes.

2.7.7 Churn Rate

The *session churn rate* refers to the rate at which login and logout events occur. It reflects the rate at which nodes enter or exit the system. In our measurements, churn rate is determined by counting the number of login and logout events that occur over a fixed period of time, referred to as the *bucket*.

A high churn rate has the same ramifications as a short inter-arrival time. Systems with high churn must be able to handle frequent changes to their membership sets. The overhead of adding and removing nodes should be low, and there should be minimal penalties to the system when nodes enter or leave. Suppose each time a node enters or leaves the P2P framework, it must exchange information regarding the objects it is hosting. Then in the face of high churn, a large amount of bandwidth might be consumed to facilitate this information exchange, causing poor performance to the rest of the system.

A useful metric to study along with churn rate is the *churn source set*, the set of players who cause churn within each bucket. A high churn rate could be caused by many players joining and leaving the system, or it could also be caused by a single player joining and leaving far more rapidly. By comparing the size of the churn source set with the number of events, the average number of events per contributing player can be determined. The effects of a single player causing most of the churn versus many players contributing to some of it can be different; in the former case, the guilty node could just be ignored or labelled faulty, resulting in the effective churn rate to become much lower. We subsequently refer to the ratio of churn over source set size (events found over players found) as the *churn/player ratio*.

In the long-term experiment, churn rate was determined by taking the total number of login and logout events occuring within 10 minute buckets.

The distribution of session churn is shown in Figure 2.14, which has a median of 11 events.

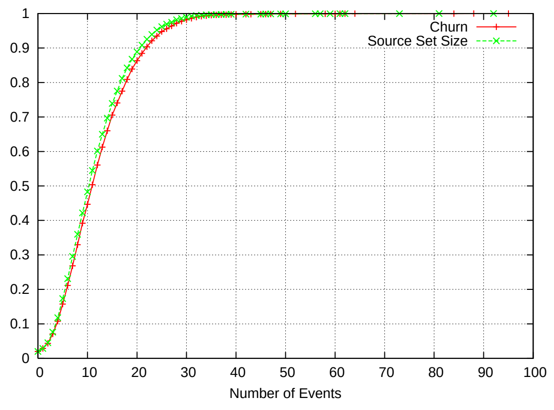

Figure 2.14: Churn rate and source set size.

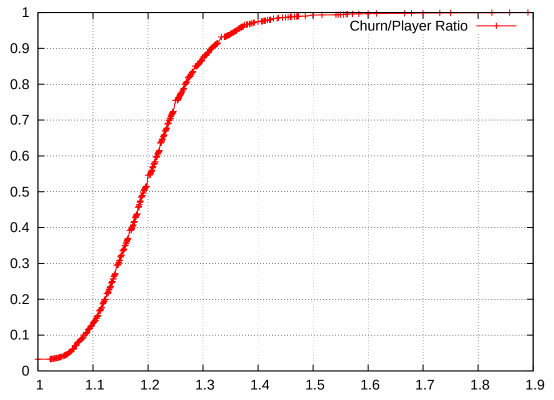

Figure 2.15: Ratio of churn rate and source set size.

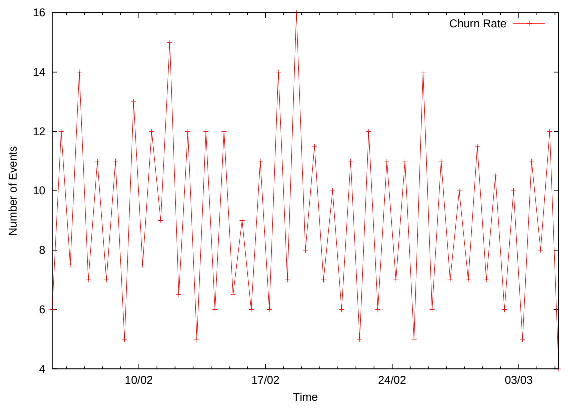

Figure 2.16: Median churn rate every 12 hours over February.

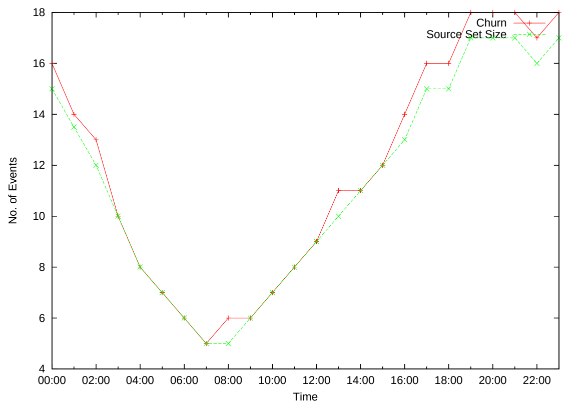

Figure 2.17: Relationship between churn and time of day.

The distribution of churn source set size is also shown in the same figure, and it is similar to that of session churn. This suggests that most login/logout events detected over the 10 minute bucket were all due to different players, *i.e.* few players logged in and out (or vice versa) within 10 minutes. The median churn/player ratio was 1 (with a mean of 1.06), further supporting this argument.

In order to understand how the source set size varies over larger periods of time, the churn/player ratio was determined over a dataset with 30 minute buckets. This gives us a measure of the average number of login/logout events per player over 30 minute intervals. The distribution is given in Figure 2.15, which has a median of 1.2. Thus, high churn is generally caused by a large group of players.

The median churn rate every 12 hours for the month of February was also determined, and is plotted against time in Figure 2.16. Using 12 hour bins allows the diurnal nature of churn to be seen, which is confirmed in the time-of-day graph shown in Figure 2.17.

Some further calculations are necessary to make sense of the churn rate. 11 events every 10 minutes gives a churn rate of 1.1 events per minute. Normalizing by the sample size,and assuming that each event was caused by a unique player, we get that 0.1% of the population either enters or leaves the game every minute.

0.1% does not seem like a large percentage, but recall that the median number of players online at any time is close to 6% of the total population. That equates to 1.6% of the online population logging in or out every minute. Note that this does not imply that 6% of the total population enters or leaves the game every hour; our assumption that each churn event was caused by a unique player only holds at shorter time-scales such as 10 minutes. Over an hour, we are likely to see pairs of churn events (a login and a logout) caused by a single player, since the median session time is less than an hour.

2.7.8 Are Players Independent?

2.7.9 Mobility and Location Churn

A player's *mobility* is rate at which the player moves through the game world. If we partition the game world up into locations (zones in WoW), then mobility can be expressed as the rate at which players change their location in the game.

The related metric of *location churn* is a measure of how frequently players enter and leave a location. In general, highly mobile players result in high location churn.

High mobility and high location churn will cause difficulties to both the P2P and federated frameworks. Mobile players stress the discovery component of the frameworks because they need frequent updates to their set of nearby objects, while at the same time changing the set of nearby objects of all other objects they pass by. If there are any overheads associated with adding or removing objects from the set, then high mobility will result in performance penalties.

2.7.10 Stay Time

The analog of session length with respect to locations is *stay time*, a measure of how long players stay within a particular location. The federated framework is affected by stay time, because a short stay time means that player objects will be hosted by servers for only a short while before changing hands. This might result in extra overhead. The same applies for any object placement strategy that favors placing nearby objects on the same node.

2.7.11 Location Density

The *density* of a location in the game world refers to the number of players within that location. Dense areas with many players generally require more processing and bandwidth usage, because there is likely to be more interaction between nodes. A particularly dense location might cause problems for the federated framework, by overloading the server associated with that location.

It is particularly interesting to observe the *stability* of location density in the game world. If dense locations tend to always be dense, then the federated framework can allocate more resources to those servers managing such locations. Conversely, if density tends to change, then such an approach would not be very beneficial, and might even negatively affect performance.

2.8 Discussion and Analysis

Chapter 3

Conclusion

Bibliography

[1] World of warcraft census statistics. URL `http://warcraftrealms.com/census.php`.

[2] Wine. URL `http://winehq.org/`.

[3] Wowwiki. URL `http://www.wowwiki.com/`.

[4] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM*, pages 104–113, 2001. URL `citeseer.ist.psu.edu/baughman01cheatproof.html`.

[5] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, Portland, OR, August 2004.

[6] A. Bharambe, J. Pang, and S. Seshan. A Distributed Architecture for Multiplayer Games. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[7] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 1–6, Lihue, Hawaii, May 2003.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas*

in communications (JSAC), 20(8):1489–1499, 2002. URL `citeseer.ist.psu.edu/castro02scribe.html`.

[9] Chris Chambers, Wu-Chang Feng, Sambit Sahu, and Debanjan Saha. Measurement-based characterization of a collection of on-line games, 2005. URL `http://www.usenix.org/events/imc05/tech/chambers.html`.

[10] Wu chang Feng, Francis Chang, Wu chi Feng, and Jonathan Walpole. Provisioning on-line games: a traffic analysis of a busy counter-strike server. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 151–156, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-603-X. doi: http://doi.acm.org/10.1145/637201.637223.

[11] Chris Gauthier Dickey, Daniel Zappala, and Virginia Lo. A fully distributed architecture for massively multiplayer online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 171–171, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-942-X. doi: http://doi.acm.org/10.1145/1016540.1016566.

[12] Kieron Drake. Xtest extension protocol. URL `citeseer.ist.psu.edu/230467.html`.

[13] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *Proceedings of ACM NOSSDAV*, June 2004. URL `citeseer.ist.psu.edu/gauthierdickey04lowlatency.html`.

[14] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945475.

[15] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua –

an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, 1996. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6⟨635::AID-SPE26⟩3.0.CO;2-P. Also see http://www.lua.org/.

[16] Blizzard Inc. World of warcraft, . URL `http://www.worldofwarcraft.com/`.

[17] Blizzard Inc. World of warcraft faq, . URL `http://www.worldofwarcraft.com/info/faq/`.

[18] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games, 2004. URL `citeseer.ist.psu.edu/knutsson04peertopeer.html`.

[19] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000. URL `http://citeseer.ist.psu.edu/kubiatowicz00oceanstore.html`.

[20] Linden Labs. Second life. URL `http://secondlife.com/`.

[21] ImageMagick Studio LLC. Imagemagick. URL `http://www.imagemagick.org/`.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001. URL `citeseer.ist.psu.edu/article/rowstron01pastry.html`.

[23] Timothy Roscoe Sean Rhea, Dennis Geels and John Kubiatowicz. Handling churn in a dht. Technical Report UCB/CSD-03-1299, EECS Department, University of California, Berkeley, 2003. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/6360.html`.

[24] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*,

pages 137–150, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-603-X. doi: http://doi.acm.org/10.1145/637201.637222.

[25] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-561-4. doi: http://doi.acm.org/10.1145/1177080.1177105.