WebHack: A Research System for Social Massive Multiplayer Online Games

Arnoud Bakker

Delft University of Technology

System Performance of a Platform for Social Massive Multiplayer Online Games

Master's Thesis in Computer Engineering

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Arnoud Bakker

16th January 2012

Author
Arnoud Bakker

Title
WebHack: A Research System for Social Massive Multiplayer Online Games.

MSc presentation
20th January 2012

Graduation Committee

prof. dr. ir. D. H. J. Epema (chair)	Delft University of Technology
dr. ir. A. Iosup (supervisor)	Delft University of Technology
prof. dr. ir. S. D. Cotofana (CE)	Delft University of Technology

Abstract

The most popular Facebook games are being played by millions of users, sometimes only a few weeks after introduction. Although several of these games and their users have been studied, there currently exists no open-source version of such a game.

In this thesis we present the design, implementation, and performance analysis of WebHack, a Facebook-integrated multiplayer game. WebHack is built upon the classic, but still popular, game of Nethack. We discuss the technical difficulties with file handles, process limits, communication networks, and handling failures.

Further, we consider aspects specific to legacy applications, for example legal issues and technical limitations. We propose methods to circumvent these issues, and show a successful integration of the legacy game Nethack into our Facebook-integrated game system.

We present the design of our system and evaluate the performance of the design in various scenarios. Among other results, we show that WebHack is a high-performance system, able to support over 300,000 concurrent players, handle arrival rates for up to 1,750 new players per second for over 60 seconds, and is able to recover within 10 seconds from a server failure.

Preface

I want to thank my parents, employer, and supervisor.

I am not new to the subject of multiplayer games; I have hosted my own multiplayer online RPG for over ten years. This led to the following tribute:

Rage of Vengeance masters!

Supreme Entity
Akasi Arwen Balin Foil Ignite Jrk Loesje Lucifer Vigo Zur

Posse
Fallout Mandor Morra Pygmy Sphere Summoner Uber Wacky Xenthar Xypher

Arnoud Bakker

Delft, The Netherlands
16th January 2012

Contents

Chapter 1

Introduction

1.1 Background

Gameplay is an old and important aspect of human interaction. It is so much part of our nature that even seems to predate the existence of human culture [1].

Some games played today were also played in a similar form centuries ago, but new games are also constantly being invented. This thesis focuses on one of the newer forms of gameplay, which is based upon the modern invention of social network websites. First, we give some background on computer gaming, and specifically on the game Nethack, then we discuss the Facebook platform and introduce our problem statement and research questions.

1.1.1 Computer Games

Computer gameplay was made possible around 1950 and is currently a popular daily activity in areas where electrical power is available, although games with boards and pieces, or with a ball or a piece of string, are also still popular.

Game development followed the communication trends and multiplayer games evolved for telephone modems, local area networks, e.g. Ethernet, and the Internet. Earlier Internet games often matched players with no common background against each other. Slowly, games shifted towards a model where players can keep track of their friends and favourite partners. This made Internet computer gaming less anonymous, and more of a social activity.

Game design was influenced by this social trend and the multiplayer online games (MOGs) became more diverse. In a field first dominated by fast-paced, aggressive, and competitive MOGs, games which were slower, more constructive, cooperative, and narratable gained popularity [2, 3].

This research project revolves around social MOGs (or SMOGs). For this purpose a basic, but fully functional, SMOG system, based upon the existing game of Nethack, is designed, implemented and analysed.

Nethack

Nethack [4] is a well-known computer game. Whereas the first version was released in 1987, the latest functional updates were made public in 2009. Updates to support modern platforms, such as smart mobile phones, are still being developed.

Nethack is a turn-based single-player game. The player controls a humanoid character, which is trapped in a large dungeon. The goal is to obtain the legendary Amulet of Yendor. For this purpose, the dungeon must be explored while fending off dangerous monsters and completing other challenges.

This game is the most popular and successful descendant of the text based role playing games (RPGs), a genre which started around 1975. Although the transition to advanced 3D graphics has been made, modern RPGs like World of Warcraft and Runescape are functionally similar in many respects [5].

Although being old, Nethack still enjoys a cult status and it is mentioned in many technical shows and presentations. The game is included in most Linux-based operating systems installations. A part of the cult aspect stems from the ruthlessness and complexity of the game. There are around a hundred different commands, but many can be used or combined in surprising and unpredictable ways.

Most of the time, a single event can result in the death of the character, which immediately ends the game, forcing the player to start over with a new character. This last aspect forms a bond between Nethack players, who amongst themselves seek comfort to complain about frustrating in-game deaths, but also marvel and enjoy the humour, clever design and remarkable attention to detail of this epic game.

1.1.2 Facebook

Facebook was founded in 2004, first operating as a Harvard university social website, and going open for the public in 2006 [6].

Since then its user base grew enormously. In August 2011, over 375 million people were logging in daily, over 2.5 million websites have been integrated with Facebook, with ten thousand more following each day [7]. These websites use the integration with Facebook to track what people like, advertise, offer services, and study or interact with their public.

Facebook's large collection of personal information is both a treasure and a liability, placing the company in the middle of a permanent struggle between economic factors, privacy laws, and user satisfaction.

The popularity of Facebook and other social websites gave rise to a new breed of online multiplayer games, called social multiplayer games. Similar to normal multiplayer computer games there are played online, but instead of playing with fairly unknown players from around the world, most interaction occurs within the social circle of the player. These social multiplayer games tend to be less aggression-oriented and more aimed at cooperation.

1.2 Problem Statement

Although many SMOGs exist, none of them is open-source and available for research purposes. It is not a trivial task to construct a computer game system that is capable of supporting thousands of concurrent players. In an SMOG, the strong interaction between the requests of different players creates additional technical challenges.

Massive Multiplayer Online Games (MMOGs) have been extensively researched, and this research has already produced algorithms, techniques, and system designs capable of supporting a very high number of players.

This knowledge will be used to construct an MMOG system based upon Nethack, integrated into the Facebook platform. We call your system WebHack, and we designed it as a realistic research SMOG capable of handling a massive number of players. The design of the system is presented in Chapter 3. The experimental results are shown in Chapter 4; notably, in Section 4.4 we show that WebHack is truly a massive multiplayer game, and that WebHack is capable of handling hundreds of thousands of players concurrently.

1.2.1 Research Questions

We focus on three main research questions. Other goals of this research project are discussed afterwards and include technical challenges and a creative way of handling GPL-licensed software within a larger system. We discuss the background of the research questions, and related work, in Chapter 2.

Q1. How can massive multiplayer online games be integrated into the social platform Facebook?

The Facebook platform offers integrators a large and complex API, which is updated regularly. One of the features, that is, the 'I Like' button, has become a well-known concept. Many other forms of integration are possible, but no standard best-practice method exists.

During our research a flexible method of Facebook integration is implemented and tested. A discussion of Facebook API performance, and analysis of the integration is included in Section 4.2.

Q2. What are efficient methods in supporting social MMOGs on a heterogeneous multi-cluster server system?

It is technically challenging to build game systems able to handle multiple dozens of concurrent players [8].

There are many different game system designs [9, 10, 11], and we are restricting our research to systems using a client-server model where the server role is distributed over a cluster of machines. The server cluster is able to dynamically change in size. This has the advantage of being able to add resources when more are required, and being able to remove them to avoid unnecessary operating costs.

The system requires a mechanism to distribute server tasks over the available server machines. Different algorithms to determine which server handles new games are implemented in the research system, as described in Section 3.3.2. We include an analysis of the performance of these methods in Chapter 4.

Q3. How can a MMOG system be made tolerant to simple system defects?

Distributed computer systems become quite fragile, if an insignificant failure on a single node can affect the outcome of the entire system. Such a failure might occur in a game system when a server unexpectedly becomes disconnected or when the server that handles all the login requests crashes.

We investigate methods to ensure continued operation of the game system, even when such a failure occurs. This could avoid system breakdown and possibly loss of income. In Section 3.2.7, we describe how a WebHack system without single points of failure can be constructed. An experiment which shows the performance during server failures, and an analysis of the results, is included in Section 4.5.

1.2.2 Technical Objectives

To answer the research questions Q1-Q3, we have designed and implemented a SMOG system, called WebHack. In this section we describe the technical objectives of this system. The design and implementation details are presented in Chapter 3.

T1. Construct a fully functional web-based SMOG system.

The qualify for this objective, WebHack should not only offer a fully playable game through a website, but also the secondary functionality of a gaming system. This includes basic support for logging, maintenance, security, and interaction with other players.

T2. Construct a system capable of running on a heterogeneous multi-cluster server.

Achieving this technical objective means that our system is able to handle differences in hardware, and differences between groups of machines.

A heterogeneous server cluster consists of machines with different hardware platforms. WebHack should not only be able to run on different types of hardware, but also to form one server cluster which such systems.

A large number of machines, forming a single system, is sometimes subdivided in smaller groups, called clusters. Such a subdivision can be part of a naming convention, or be based on the network structure. This last situations might lead to a system where there are differences between inter-cluster and intra-cluster communication. In many situations inter-cluster communication is slower and subject to more access restrictions. Our system should be able to operate on such multi-cluster systems.

T3. Construct a system capable of supporting over 250,000 concurrent players.

According to Wikipedia [12] a system has to be able to support hundreds or thousands of concurrent players to be considered an MMOG.

Achieving this objective enables our system to qualify as a massive multiplayer game, and even demonstrates that it is capable of handling an amount of players only found on popular SMOGs.

Legal Objectives

L1. Show a practical method of integrating legacy software into a modern platform.

Most software comes with a license, which determines acceptable usage. It is no surprise that users are not allowed to copy, resell or redistribute most software they have payed for, but some of those restrictions also apply to free software.

Although Nethack is freely available, the use of the source is subject to a license. If a SMOG system is constructed by expanding the original Nethack program, the use of the entire SMOG system would be subject to the Nethack license. We plan to use Nethack in our SMOG system, but to minimise the effect of the license restrictions.

There are a number of popular license schemes for free software. Software which is placed in the public domain is considered to be copyright-free, and can be used for almost any purpose.

Most free software is released under a derivative of the General Public License (GPL). Software released under this license can be used, modified, and distributed free of charge. It is allowed to distribute modified version of the software, but not to charge for their use, or to place restrictions on access to the source code.

By using the game of Nethack in our SMOG system, it is almost unavoidable that a part of our source code can be considered to be derived. We attempt to show practical usage of an unmodified version of Nethack-3.4.3 in our research system, and explain the implications of this approach in Section 3.3.5.

1.3 Thesis Outline

In the next chapter we discuss the background of the research questions and technical objectives, and talk about MMOGs, Facebook Applications, and related work. In Chapter 3 we present the design and implementation of our research game system, WebHack. Chapter 4 contains the description and results of our experiments. We finish with a conclusion in Chapter 5.

Chapter 2

Related Work

In the previous chapter we defined our research questions on Facebook integration, game-system scaling, and fault tolerant game-systems. In this chapter we will discuss the background of the research questions. Some of the aspects we introduce will be used in the design of our MMOG-system, which we introduce in the following chapter. Others topics are here to serve as a starting point for further research.

2.1 Research on Social Games

Many Facebook aspects have already been the subject of scientific research. In the next sections we will discuss a bachelor of science project about a Facebook integrated shooting game, the social aspects of game design, and discuss some of the statistics of Facebook Applications and their users.

2.1.1 Constructing a Facebook-Integrated MMOG

In 2010, two bachelor students F. Jutte and J. de Swart completed their bachelor of science project on a Facebook-based MMOG at TU Delft. They designed and implemented a prototype of a web-based, shoot-em-up game. The game system had a client-server architecture. Player commands and location updates where distributed through a messaging system.

In this approach the number of messages per second grows quadratically with the number of players, making the system unable to support a large number of players. Their research focused on investigating different methods to reduce the number of messages sent. The main idea was to reduce the area of the world on which the clients received updates. The part of the world a client was interested in is called the Area-Of-Interest (AOI).

The mock-up game was lightly integrated into Facebook. Their thesis described other possibilities for integration, which we further explore in this project.

2.1.2 Designing Social Games

Web-based games are not new, but Facebook-integration opens up new possibilities. In the introduction, we already mentioned that violent and fast-paced games did not fit the social genre very well. By looking at the first games that flourished on the social websites a number of similarities appeared. At least five different design aspects [3] were found to be important for producing social web games: physicality, spontaneity, sociability, narrativity and asynchronisity.

Social games contain actions which humans are physically able to do, but hide the complexities and effort of those actions. Metaphors should be used to make the game easy to follow. The common social relations between players are used in the game design. The game was designed while considering users who play a number of short sessions daily.

These aspects are simple to grasp. They are easy to integrate into existing games, so they can serve as a guidebook for developing Facebook-integrated games.

2.1.3 Statistics of Facebook Applications

The usage of a number of different Facebook applications has been traced and analysed [13]. This analysis gives insight into the geographical distribution of the users, the intensity of the interaction between users, and the distribution of network traffic. Most of the results are representative for reasonably popular applications studied over a long period of time. Among the important results is shown that application interaction with Facebook can incur multi-second delays, and worsen for more popular applications.

Because of the slow nature of most social games, such delays are not a direct problem, but they can become a source of annoyance for the players. With most applications, the number of users grows at a nearly exponential rate during the first days of operation, but remains fairly constant afterwards [14].

2.1.4 Current Status

Currently, the game developer Zynga is dominating the Facebook application arena [15]. The company, only 4 years old, was estimated to be worth around 7 Billion dollars. At least 25 percent of all Facebook users played one of their games. Their most popular game, FarmVille, services over 25 million players daily. Zynga uses both a private data centre and commercial cloud-based solutions to host their games. The clouds are mostly used to handle the large traffic spikes generated by recently introduced games.

Blizzard Entertainment still manages to attract over 11 million paying subscribers with their 7-year-old game ’World of Warcraft’. Blizzard constructed the server clusters out available hardware, but they are largely operated and maintained by partners because commercial cloud computing was then still in its infancy. In general, games which require fast responses are more commonly run on self-engineered server systems, rather than clouds.

2.2 Research on MMOG performance

The maximum number of players in an MMOG greatly depends on the game genre. Most high-paced games, e.g., first-person shooters, are not able to support more than several dozen players, but run into that limitation when a single room or area becomes very crowded [8].

For certain subtasks a client has to consider all other nearby clients. A few of these tasks, e.g., path finding, collision detection, and maintaining a consistent global state, consume an increasing amount of extra resources when considering an additional client. When the resources of one of the servers of a server cluster are exhausted, this can cause the game to slow down or even fail completely.

Although slower games do not suffer from the same problems, they can also be slowed down considerably when the number of concurrent players increases. Common limiting factors are the maximum number of database transactions per second, and the overhead caused by locking operations.

2.2.1 Load Balancing in Structured P2P Systems

Ananth Rao e.a. [16], compared different load-balancing approaches in a P2P-system where some nodes were overloaded with work, and others had resources to spare. They experimented with different methods and algorithms of swapping tasks between nodes in order to reach a state where all nodes are moderately loaded.

Their work showed that for a sufficiently large network a simple scheme, in which each overloaded node shed tasks to the least loaded node it was aware of, performed amongst the best ones, while requiring the least communication.

Although the work focusses on a P2P-system which did not have gaming functionality, but instead it offered access to a database, the results could be equally applicable, because of the functionality agnostic nature of the load balancing algorithms.

2.2.2 Modelling HTTP Network Traffic

Popular web-servers are visited daily by a large number of clients. The visits differ greatly in length and number of requests. A model of such a server can constructed using Queueing Theory.

If the arrival of clients is modelled as a Poisson process, the system can be easily analysed analytically, and the model can be made to closely resemble the traffic of an existing web-server. However, Ethernet network traces were shown to exhibit self-similarities, a feature which can be described by the Hurst-exponent. It was shown that the self-similarities occurring in network traffic can not be explained by an underlying Poisson process [18].

A difference between the Poisson model and reality is found when comparing traffic on a single Ethernet cable with the summed traffic of a number of cables, or by looking at a trace with different enlargement factors, as shown in Figure 2.1.

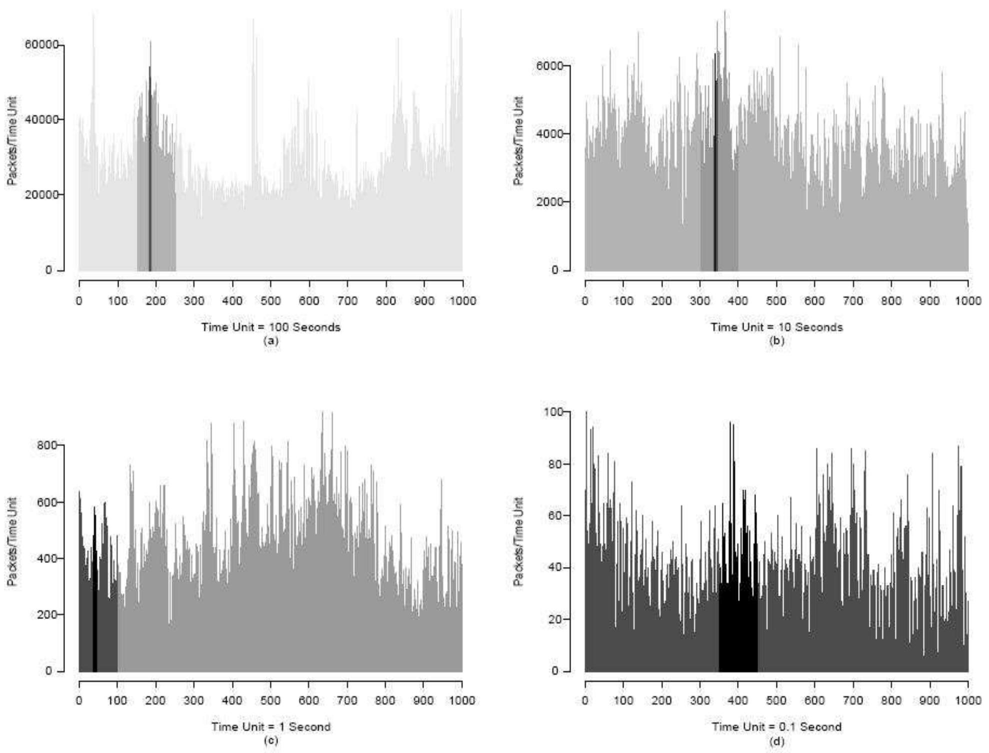

Figure 2.1: Self similar features of Ethernet traffic [17].

The Poisson-based model predicts the traffic to become more regular where in reality it will contain more and even higher spikes.

Various other queueing models were found to be able to produce traffic with self-similar features. In one of our experiments we use a model based upon the Weibull distribution. This distribution was selected over the simpler Pareto-distribution because the heavier tail more accurately describes web-traffic.

2.3 Research on Game System Failures

To avoid the effect of failures in a distributed game system, a number of techniques which where developed for P2P systems can be used.

A number of algorithms, originally used for P2P systems, have been proposed to reduce resource requirements for MMOG-servers [19, 20]. However, limitations to the trustworthiness of client-machines limit the possibilities of this approach. This is not only a problem to P2P-based game systems, traditional client-server based game systems are also vulnerable to a long list of cheating possibilities [21].

Inside a server cluster all the machines are trusted completely, which allows P2P-techniques like Pastry and PAST [22, 23] to be used to provide a persistent database. PAST is a file system, built on top of Pastry, a P2P-system. The file system handles requests to store and retrieve files. Stored files are replicated on multiple systems, allowing correct retrieval even if a small number of systems leaves the network. Requests for retrieval are routed to one of the systems holding the file, an attribute which also services as a load-balancing mechanism.

The P2P-system Pastry is not very relevant, since it is constructed for a highly dynamic environment of average Internet users, quite unlike the dedicated server cluster we are using. However, a PAST-like file system could be used to provide reliable data-storage for our SMOG system. With the game data stored on multiple hosts, there are possibilities to balance the load and reduce the effect of failing servers.

We implemented a number of P2P techniques in WebHack. A study into the possibility of adding more P2P functionality can be a topic of future research.

Chapter 3

WebHack: A Research System For SMOGs

In this chapter we describe the design and implementation of our Social Multiplayer Online Game (SMOG) system, called WebHack. We first present a functional description of the system, then the overall design and several major components. Last, we discuss the implementation of the system.

3.1 Functional Description of the System

The WebHack system has three main functional aspects. It is a complete game platform, it is usable for scientific research, and it has a working Facebook integration. We discuss these aspects, and their influence on the design, in the following sections.

3.1.1 Game Platform

WebHack is a web-game platform capable of delivering the performance of an MMOG. The platform runs on a cluster of server machines, and offers the functionality of a game-system to its users. This is not limited to offering a fully playable game, but also includes features like security, administration, and maintenance. These features are not only included for their functionality, but also to avoid working with an oversimplified game-system. Measurements on a system without these features might lead to optimistic performance estimations or unrealistic conclusions.

WebHack is a large-scale system, which operates on a cluster of machines. In Section 1.2.2 we listed the objective to make the system capable of handling 250.000 concurrent players. WebHack contains functionality to comfortably construct, configure, and modify clusters of dozens of machines, and to distribute the workload over them.

Since it is impractical to require a large number of identical machines to be available, WebHack is able to run on systems with different hardware configurations and operating platform. WebHack can also join such systems into a single server cluster, which is one of our technical objectives.

We operate WebHack on a cluster of machines during the experiments. If components are placed on a single machine, they experience greatly reduced communication delays. Such a setup would not lead to realistic measurements.

3.1.2 Research System

WebHack is a platform for scientific research. It offers the functionality to run diverse and repeatable experiments on clusters of server systems. Complex scenarios can be instrumented accurately and the system can be adapted during operation to handle new situations.

An experiment is created by defining an initial system state, the configuration options, and the arriving workload. Most changes to the configuration options do not affect the final state of the system, making it possible to repeat an experiment with different settings. The first run can be made in a slow, verbose mode, and the second run using settings for optimal performance. The output of the first run could be used to verify the correct operation of the system. Then, the performance can be measured during the second run.

Repeatability requires the careful storage of random seeds, and event time stamps. Nethack bases some of its decisions on the output of random functions or by looking at the local clock. When different values are used in different runs, the output of the game can greatly differ. The same output can be reached when the random values and time stamps are stored, or when they are derived in a reproducible manner.

New situations can be handled effectively by changing the configuration, and even the executable code. Most components of WebHack can be dynamically reconfigured, and large portions of the executable code can be reloaded without losing state.

The combination of these features allows the WebHack operators to add new configuration options within a running system and immediately start using them. This flexibility allows investigating complex problems without having to rebuild the conditions, by gradually adding code to first identify and later handle to problem.

The features are able to work on systems which use a large number of server machines. Configuration changes can be made on groups of machines and code updates are distributed automatically. The measurement data for all machines are collected onto a single machine, even if the server cluster suffers from failing machines. On this machine the information is combined, and further processed into images, of which we show some in Chapter 4.

3.1.3 Facebook Integration

WebHack is a social web-game. The social aspect is integrated into gameplay, and into the system.

The game offers social features, such as making players able to invite new players, share items with their friends, and observe others games. The system collects high-scores and other accomplishments, and allows players to compare and share this information. It is possible to add, and experiment with, new forms of interaction.

On a system level, the social aspect is accomplished by registering WebHack as a Facebook Application. It uses the Facebook API to communicate with the Facebook platform, but is able to operate even when this communication is slow or otherwise corrupted.

The Facebook integration makes the system part of the Facebook experience, and raises the users expectation of the systems availability. WebHack gracefully handles sudden server failures, and is able to run multiple reachable instances of all its components.

3.2 System Design

WebHack, our SMOG system, consists of three main components. The first component is called Nethack+, which is a Nethack process with wrapper code around it. The second component is the Nethack Handler Daemon (NHD). This daemon manages the local system and a group of Nethack+ processes. The third component is a website, called HackSite, which handles the communication between NHD and the players.

The secondary components are the graphing tool called 'Grapher', a computer player called 'Bot', and a daemon to bundle HTTPS connections, called 'Bundler'.

3.2.1 Nethack+

WebHack builds upon the existing Nethack game. However, the game needed to be heavily adapted to suit this project, since multiplayer features and web integration lacked. To add this functionality, a wrapper for the Nethack application was constructed.

The wrapper intercepts and handles some calls Nethack makes to the Operating System (OS) on which it runs, i.e., Linux. Some of these intercepts change the input and output behaviour of Nethack, for example, the terminal menu screens are redirected to a website with choice buttons. Other intercepts change the internal behaviour of Nethack, for example, when a multiplayer level is selected or an item is donated to a player in another game.

From here onwards we refer to a wrapped Nethack process by placing a plus-sign after the name, referring to it as Nethack+.

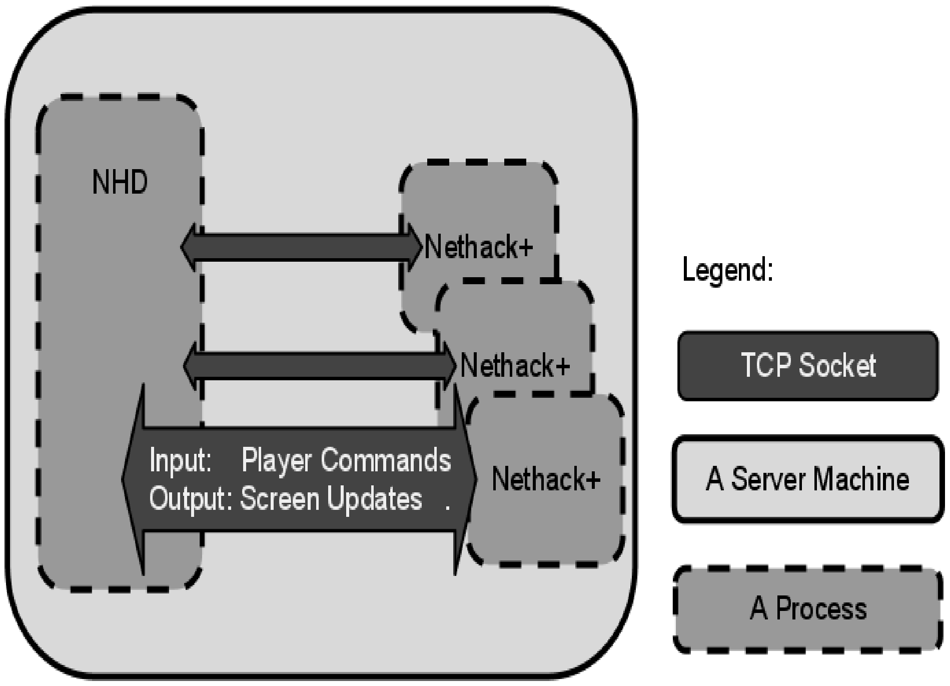

Figure 3.1: Schematic overview of a NHD with several Nethack+ processes.

3.2.2 Nethack Handler Daemon (NHD)

The NHD is a stand-alone daemon process, which is started on a number of server machines. Its primary task is communicating with Nethack+ processes, other instances of NHD, the Facebook API, the website, and humans.

The communication with Nethack+ processes is shown in Figure 3.1. Besides the communication, NHD is also responsible for creating and destroying Nethack+ processes, handling their file access, recording their input requests, storing the current game screen, and handling player commands.

To communicate with other instances of NHD they have to located, and connections have to be set up. These connections span up a communication network through which all the components in the system can be contacted. We discuss different methods to locate and connect in Section 3.3.2.

Contact with the other NHDs is used to distribute, among other things, information about games, configurations, and statistics. Distributing game information allows the system to handle situations where a server fails. The information is also used to assign new games to the correct NHD.

These requests have to be distributed in such a way that none of the systems run out of resources. This is complicated by the fact that due to hardware differences between servers, resources on one system are not necessarily comparable to an equivalent amount of resources on another system.

Distributing the knowledge of the games allows the system to handle failing servers gracefully and find the correct place to place new games.

NHD also uses HTTPS to communicate with the Facebook API. Through this API it can retrieve detailed information on users and their friends, or post updates to the public message board of users.

For operators it is also possible to interact with a NHD directly, through a terminal or the network. Commands can also be forwarded by other instances of NHD, or through the website.

3.2.3 HackSite

HackSite is the name of the website for this system. It can be accessed directly or through the Facebook website, and can be used by both players and maintainers. For direct access, the URL *https://webhack.nl:21592/* can be used. To access the site through the Facebook website, one has to search for the 'WebHack' Facebook Application.

The web pages are delivered to the clients browser by one application. The application uses NHDs as its information source, and is mostly responsible for formatting this information. It also adds web scripts to support hot keys and handles the clickable game interface.

The website itself contains no game logic and understands little of the information it is showing to the user. For example when a player clicks on a selection box which is shown to him by HackSite, the mouse click is converted to a keystroke, which is injected in a Nethack+ process. Nethack+ reports an updated menu, containing a check mark in the selection box to NHD. NHD transfers the updated menu page to HackSite, which renders it in the player's browser. Although this method requires a lot of communication steps, its generic approach makes it suitable for all the game screens that might arise, greatly reducing the number of game situations which have to be tested.

3.2.4 The Bot

Although WebHack is fully playable, we also created a simple computer player, called Bot, for research and experimental purposes. By using a computer player we are able to create, and recreate, a wide range of input patterns. The Bot does not need to be a very proficient player, as long as the workload generated by a group of bots resembles the workload created by a group of human players.

The Bot can emulate different arrival patterns for the players. We made use of three different patterns, one where all the players arrive instantly, one where they arrive in a constant rate, and one where the arrival of the players mimics the daily pattern of an average, but somewhat simplistic, human.

Because of similarities in the communication, the robot is able to connect to a NHD directly, or connect to the HackSite. The direct option allows specific components of the system to be tested, and the latter setting allows for more accurate measurements for a complete system.

3.2.5 The Grapher

When working with a large WebHack setup, there are hundreds of instances of NHD running concurrently. They send commands to hundreds of thousands of different Nethack+ processes. Such a setup generates a huge number of statistics. To gather and process all these statistics a separate component was constructed. It is called the Grapher, and can be run both from the command line, and as a CGI script. The last option makes it possible to refine complex graphs with a web browser.

The Grapher connects to a NHD and continually queries it for new events. An event can signify many different things. For example it can be an entry from a log file, a number which describes CPU usage for an application, or a collections of measurements from the last second. Each events is created by a NHD, and then communicated to the other instances.

It is also possible for the Grapher to connect to multiple NHDs at the same time. Such a setup is mostly useful for tracing WebHack while running tests where systems can suffer from failures. It that case multiple connections to the server cluster ensure that event information is always able to reach the Grapher.

3.2.6 The Bundler

To save file descriptors on certain machines, and to allow a more convenient handling of internal firewalls, we created a simple connection bundling application.

In Figure 3.2 we show how the Bundler is placed on the gateway and internal machines. It is set up to bundle the incoming HTTPS connections into a single connection. In this setup only the gateway machine, or machines, require the rights to receive inbound HTTPS connections from the Internet. This setup is more secure because unrestricted access from the Internet is considered a security risk. By using this tool, the risk can be avoided for the internal machines.

The use of the Bundler reduces the resource usage on the machines running HackSite, and makes it possible to run the WebHack system when only user accounts with limited privileges are available on the Gateway machines.

3.2.7 A Complete System

The components described so far in Section 3 can be combined to form the WebHack system. A complete system requires several dedicated machines, called *Gateways*, and a number of machines determined by the number of concurrent players. We call the last type of machines *Workers*. In the following text we describe the arrangement of components on these machines, the required network structure between the machines, and the steps required to activate the system.

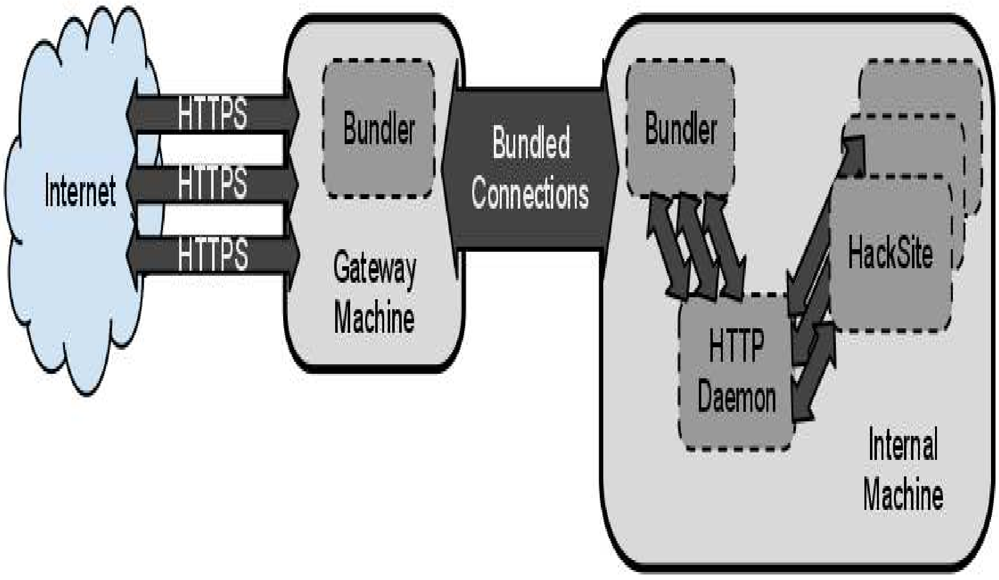

Figure 3.2: Schematic overview of the HTTPS-Bundler.

A WebHack system requires at least one Gateway machine; using multiple Gateways is preferable, since it allows to the system to function even if one of the Gateway machines suffers a failure.

All Worker machines run an instance of NHD. Gateway machines either run the Bundler, or the HackSite and an instance of NHD. A setup with 2 Gateway machines which uses a Bundler is shown in Figure 3.2. At least one instance of HackSite is required, the use of the Bundler is optional.

To improve security, network access rights can be restricted for the server machines. Gateway machines require the rights to receive inbound HTTPS traffic, and the rights to connect to the Worker machines. Worker machines require the rights to make outbound HTTPS connections to the Facebook website, and the rights to connect to many other Worker machines. It is not required to be able to contact all other Workers directly, as long as an indirect path from every Worker to every other exists.

Starting and configuring the Bundler is simple. The only configuration option is the port number to listen on. The application is started from the command line, and the same method applies for most other components.

The HackSite is activated by starting a HTTP-daemon application. We used a small and simple HTTP-daemon called ’Mongoose’, which we downloaded from *http://code.google.com/p/mongoose*. After placing the HackSite executable in the documents directory, the website is operational. However, it will only show a page stating that an important component could not be reached.

The NHD is bundled with a small configuration file, which is used to prevent a NHD from starting Nethack+ processes on specific machines, and to correctly identify broadcast addresses.

No further configuration is needed. After starting up, the NHDs automatically locate other instances and construct a communication network. When one instance of NHD on a Worker node connects to a NHD on one of the Gateway machines, the system is ready for operation. The discovery process usually takes only a few seconds.

After the system is brought online, players can connect to the system over Internet, or a workload can be applied by running the Bot software. The Bot software runs on separate machines, which, depending on the settings, either require the rights to set up HTTPS connections to the Gateway machines, or the rights to set up TCP connections to the Worker nodes. The Bot accepts a number of configuration options. Among other things, they can specify the number of players that will be emulated, the delay between their commands, and the pattern in which the players arrive.

If graphs are required, a separate machine can be set up to run the Grapher. This machine requires the rights to set up TCP connections to one or more machines running NHD. Again, there is a benefit in using multiple connections in the situation that a machine fails. The Grapher stores the trace information to a database, which can be queried by accessing the Grapher with a web browser.

3.3 System Implementation

In this section we describe the implementation of the WebHack components, in the order they were introduced in Section 3.2.

3.3.1 Nethack+

Nethack+ is formed by placing a wrapper around the normal Nethack application. In this section we first explain how the normal flow of execution is diverted to activate our wrapper. Then we discuss the implementation of the wrapper in more detail and show how the dungeon map and game menus and handled.

Right after startup, Nethack uses the file system to check for the existence of a saved game. The call is intercepted by a mechanism explained in Section 3.3.5, and triggers the initialisation of the wrapper. The wrapper uses information contained in the environment variables to establish a connection to to a NHD. The connection is then used to query the saved game and multiplayer status. After startup the connection is used to transfer files, logs, information on the screen, and keystrokes.

The wrapper intercepts calls to the file system, allowing the wrapper to control which saved game Nethack+ uses as input. However intercepting all file system access is complicated by the systems standard library, which has weakly documented and complex file system requirements. These problems are avoided by adding filters capable of separating file accesses made by the application and the system libraries. Calls to output characters on the terminal are intercepted by the wrapper. From the location on the screen, and the contents of the process stack, the wrapper is able to distinguish between symbols depicting the game map, showing a menu, or displaying background narrative.

Understanding symbols on the map can be challenging. A single letter from the alphabet can indicate the presence of both a feeble and a powerful monster, which may or may not cause the letter to be displayed in a different colour. Adding descriptions for the known symbols would improve the understandability, but constant additions would have to be made to cope with new symbols.

Our solution is to use the in-game help system for this purpose. Whenever a new map symbol is reported to NHD, NHD sends a batch of keystrokes to Nethack+, which activate the help system and request the explanation of the symbol. The request for information, and its reply, are hidden from the player, but the description of the symbol is intercepted and used in the depiction on the website.

In-game menus and prompts show an overview of the players choices, and also list available hot keys for each option. This consistent behaviour allowed us to include knowledge about keyboard shortcuts in the website, greatly improving the interaction.

3.3.2 NHD

The NHD is implemented as a single-threaded, high-performance, Linux daemon process. Most of the code is placed inside a library, consisting of many different modules, which can be dynamically reloaded. We discuss three of the most complex parts of this library in the following subsections.

Connecting to Other NHDs

In the WebHack system, the NHD plays the role of communication hub. An instance of NHD maintains connections to other components running on the local machine, and communicates with the other NHD instances running on remote machines. To announce its presence, a NHD broadcasts messages using UDP traffic. When other instances are discovered, the NHD forms connections to a limited number of them. The connection scheme ensures all instances form a single network which cannot be easily split.

Broadcasting functionality is not offered by all network systems, and careless use of broadcast messages can congest networks. As an alternative to broadcasting, a list of hosts running NHD could be constructed and included in source code or configuration files. However, maintaining a long list of addresses would be cumbersome and error prone.

Another way of deriving the addresses automatically is by scanning the local network, although this option can be slow, and might require some knowledge of the network topology.

We chose to use small broadcast messages, and limited the number of broadcasts for each NHD-instance to one per second. This gives us the advantages of automatic detection of new hosts, while avoiding problems with network congestion. In Section 4.5 we analyse the behaviour of this mechanism.

After a NHD locates other instances, direct connections can be set up. A simple scheme is to have one NHD instance form a connection to every other instance, forming a star shaped network. This structure is fragile, since a failure on the central host cause the entire system to fail. Another simple scheme connects every instance to every other, creating a full mesh. Although this construct is not fragile, and messages never have to travel more than a single hop, the resource consumption is impractical; with thousand hosts this scheme requires close to a million connections.

NHD makes a limited number of connections, which still leads to a full mesh with a low number of instances. With a larger number of NHD instances, each picks its neighbours randomly from the broadcast messages received. In the rare case this scheme leads to disconnected groups of instances, NHD ignores the connection limit and adds connections until all instances form a single network. To avoid network structures which can be easily split NHD occasionally adds a connection to a host which is connected to a relatively low number of its neighbours.

Distributing New Games

When a new player arrives, HackSite sends a request for a fresh game to a predefined NHD. That instance of NHD might host the game itself, or relay the request to one of its neighbours. The system is capable of distributing games based on different algorithms. A simple algorithm places new games on random hosts. Hosts can also be selected in a round-robin fashion, or the least busy host can be selected. We leave the design, implementation, and evaluation of other algorithms for future work.

Different methods of determining machine workload are supported. The number of active games is a good indicator for the weight of the expected workload, but it ignores the difference in activity between players. Gauging the CPU usage gives insight in the summed activity of all games. Lastly, it is possible to look at command delays or number of commands processed per second. These sources of information give insight in the performance of both the local machine and the individual games.

After a new game is created on a NHD, the event is broadcasted to all other hosts. The broadcasts include the game ID and the address of the NHD which is hosting it. This information allows each NHD to deliver an overview of all existing games to WebHack. The URLs in this overview encode the information, allowing WebHack to contact the correct NHD for game interaction.

Handling Failures

Hosts can fail, for example because of a power failure. If a system runs on a single host, there is little to be done but restart the system after the failure has been resolved. Distributed systems have a higher probability of experiencing failures, but also have more ways of dealing with them. This system includes several mechanisms to reduce the negative effects of a failing host.

Even when one server fails, web requests are able to reach the cluster of NHDs through a different server. The cluster is able to communicate with all its members regardless of the loss of a single server.

Players connect to the system with a web browser. The URL of the main site points to multiple machines, each running WebHack. Those machines also run the NHD. NHD forms a communication network which cannot be split by removing a single machine. A single failure of one of these machines does not prevent requests from the players to reach the cluster of NHDs.

Each active game is assigned to one NHD. This NHD selects two of its neighbours as its backups, and broadcasts this information. All game-related information, e.g., saved files, inputted commands, and the random seeds, is shared with the backup hosts. When input arrives, WebHack sends it to both the primary NHD and his backups.

Normally, the primary NHD responds quickly after input arrives with an update. If it does not respond in time the backup hosts notice and attempt to replace the primary host is made. The backups recreate the game to its last known state, by applying the same input. If they obtain a result before the primary host sends his, they broadcast a message informing the cluster they are the new primary NHD for the game. In some cases, both backup servers claim ownership, but a protocol is in place to favour one of the claims over the other.

Performance

The performance of a NHD is important because of its central role in the system. We improved the performance of NHD over that of simple networking daemons by using three key features: Using asynchronous IO-functions, working with a single thread, and employing zero-copy packet parsing functions.

A NHD handles network traffic asynchronously. Synchronous IO-requests can block, causing a stall in execution. A NHD uses a single execution thread; thus, using synchronous networking code would cause the program to spend most of its running time waiting for the completion of IO-requests.

Linux offers different API-calls to implement asynchronous network handling. We chose the more modern 'epoll' variant, which not only made the code more responsive, but is also able to handle a much larger number of concurrent connections. Running a single-threaded daemon with one of the other asynchronous calls results in an extra 25% CPU-usage when using over 2,000 concurrent connections. The downside of the asynchronous network code is the added complexity, but the improvement in performance is noticeable.

NHD uses a single thread of execution. When external tools are run by NHD, they are placed inside their own process. The single thread approach has limitations. Due to efficient handling of player commands and incoming packets, and due to delaying some tasks until the CPU is idle, only one thread is required to perform all NHD tasks. An advantage of this approach is the lack of locking. Many high-performance daemons apply multiple threads in order to improve performance. Such applications also tend to require locking for correct operation, and to copy information between threads to separate buffers. These actions can cause them to spend a significant amount of additional resources. To handle more then 10,000 users a single thread would not suffice. We would then need to split the operation of NHD in multiple threads, while simultaneously trying to avoid installing locks for as long as possible.

Zero-copy packet parsing functions handle incoming packets without copying them before parsing. This is accomplished by copying pointers to strings, and handling substrings more efficiently. Removing the operations which copy strings around reduces the memory usage of the application and reduces the number of operations required. The usage of zero-copy packet parsing was common in traditional Linux daemons written in C, but many high-level languages do not offer the type of control over buffers which is required to avoid making copies.

In the optimal case, an operation reads network traffic into a buffer. Then, functions handle all the messages which are completely stored in the buffer, without making copies. Afterwards, the handled bytes are removed from the buffer. If ring-buffers are used to store the incoming traffic, all these operations can be performed without copying any memory blocks.

We developed a library to conveniently parse packets, and the structures within these packets. It uses specialised code to handle read-only strings and substrings. All the strings and substrings derived from one network buffer point to the same memory area, and only store the beginning point and length of the string. With this approach, many operations are trivial to implement and do not require copying of memory.

3.3.3 HackSite

HackSite is a CGI-application that creates the website for this system. These websites allow the users to control the system, and play the game. The HackSite is not aware of any game-information, and holds no state. To access information, HackSite queries the local NHD.

The website for the project is implemented as a CGI-application. This approach is not very popular anymore; most websites are implemented in a scripted language. However, by constructing the website as an CGI application, it can be implemented in the same programming language as the rest of the system. This allows us to reuse many functions and use the same libraries.

Using compiled applications also comes with another advantage; they execute a lot quicker than interpreted scripts.

The HackSite can also be used to navigate the system. Players can invite their friends, see the games they played, and start new ones. Maintainers can use the website to view the logs, analyse system performance, or change the configuration.

All the gameplay happens through a single page. This web page contains javascripts to make dynamic updates to the screen, and to register player input. Commands can be issued through keystrokes and mouse clicks, and are sent to the HackSite-application inside HTTP-requests. These request occur in the background and do not trigger a reload of the game-screen. Rebuilding the entire screen would be distracting to the player, and require too much time.

When the HackSite receives the HTTP-request, containing the player command, it sets up a connection to the correct NHD, and forward the command there. The website waits for a response. The response can contain error information, or updates to game screen. If a response is received, it is returned immediately, otherwise an empty response is returned after 200 ms. This limits the request duration and frees up resources for new commands.

When no new commands are available, the HackSite also sends requests to HackSite. These polling requests can pick up the slower updates, and updates caused by other activity.

Ajax web scripts can be used to send actions to the server in small web requests, without reloading the complete web page. The server could reply the web request with a situation update, greatly reducing the amount of information transfered. However the update can be incomplete, if an action has a delayed effect or results in an animation. To avoid missing updates the website periodically checks for them even when no input is available. We also experimented with an alternative method which uses a single Ajax connection for all updates. Although that method is cleaner and more responsive, it was abandoned because long term connections consumed an impractical amount of resources in current browser implementations.

3.3.4 Facebook Integration

The Facebook integration of WebHack is implemented in two different system components; the HackSite handles cookies and redirections from the Facebook website, and communication with the Facebook API is handled by a NHD. We also used the Facebook Developer website for our implementation.

Registering a Facebook Application

The Facebook Developer website offers a portal for application developers. With the site developers can register, configure and delete their Facebook Applications. The most important settings are an application name, the application secret, and a URL for the redirection.

We chose the name ’WebHack’ for our Facebook Application. We used a random application secret, and registered `https://webhack.nl:21592` as our website. The strange port number in the URL is the result of TU Delft policies, which requires students to use ports not registered by IANA.

Implementing the Facebook Integration

When users navigate the Facebook website, a small piece of text, called a cookie, is placed in their browser. The cookie allows the Facebook website to remember which Facebook account is associated with the connection. The same strategy is used with Facebook Applications.

When a Facebook user selects an application, the application website is shown inside a panel on the Facebook website. The application website is able to derive the Facebook identity of the user by parsing a cookie. Also included in this cookie is a token which can be used to query the Facebook API.

The application parses the cookie by first decrypting it with a static decryption key. This key is only known by Facebook, and the maintainer of the Facebook Application, thus preventing others from constructing and intercepting these cookies. The authenticity of the cookies can also be verified by looking at an electronic signature, another mechanism to prevent forgeries.

To simplify logging in, it is also possible to navigate to the HackSite directly. If the user is not yet logged in to Facebook, the site shows a button. Clicking the button results in a Facebook login. The javascript code which is called by this button is part of the Facebook API, and is described in detail on the Facebook Developer website.

Calling the Facebook API

The HackSite is able to derive the Facebook identity and an API token from a cookie. The token is a piece of which allows access to the Facebook API for a limited duration. HackSite reports the Facebook identity, and the token, to the local NHD. This instance of NHD shares this information with the rest of the cluster.

The NHD handles all calls to the Facebook API. It uses the token to access the basic account information of the users, and to gather a list of their friends. The information is then stored in a local database.

The Facebook API calls are placed inside HTTPS requests. GET and POST methods are used to query and store the Facebook account information. NHD spawns an external process to execute these HTTPS requests. In Section 4.2 we analyse the design, and the performance of our Facebook API interface.

The local database which stores the Facebook account information is able to quickly respond to requests. In the background, the information is kept up-to-date by executing Facebook API calls. This mechanism reduces the amount of API calls, and quickly handles requests. However it may occur that pages are constructed with outdated or missing information. As a result, pages created within the first 5 seconds of using the application can fail to show the full name of the user, and adding a new friend with the Facebook website can take up to a minute before being propagated to the WebHack overviews.

3.3.5 Handling Legacy Code

In this section we describe the implementation aspects of L1, as defined in Section 1.2.2. First, we discuss the Nethack license, and its relation to the General Public License (GPL). We then explain the license implications of various methods of modifying Nethack.

Software Licenses

The GPL is currently the most used license for open source software. It was created in 1989 by combining other licenses which shared a common goal. Many prominent applications release their sources under the GPL, e.g., Firefox, the Linux kernel and MySql.

These licenses prevented companies from taking publicly available source code, modifying the code slightly, and then selling the resulting product without revealing their changes. The GPL was updated twice to prevent certain methods of circumventing the license.

Nethack was released in 1987, before the first version of the GPL was created, and its license is based upon one of the predecessor of the GPL. The intention of the license is the same. It allows sharing Nethack and making modifications, but does not allow to hinder others who attempt to share the modified version further.

When the Nethack source is modified to allow the game to be used in a SMOG, these modifications would be subject to the license. The license would not allow to distribute the modified version of Nethack without also allowing access to the modified source code.

If features are added to Nethack without modifying the source code, the license can also apply. The license states that works that are derived from Nethack are only allowed if the derived work is also subject to the same license agreement.

The situation is more complicated when considering a system which acts as a portal to play a modified version of Nethack. If the portal is not derived from the Nethack code, it can use a different software license. That license might allow users to be charged a fee for usage of the portal.

The updated versions of the GPL describe the allowed usage of the protected work in more detail, which prevents bypassing the GPL with such portals.

Layers of Abstraction

We mentioned the possibility of altering the behaviour of Nethack with making modifications to the source code. This can be done by altering the environment in which the Nethack process executes.

The environment could be modified by a framework. Portions of the framework code that specifically handle Nethack would be considered to be a derived work. If the framework uses a high level of abstraction, it becomes capable of handling similar applications, and contains less Nethack-specific source code.

Our framework intercepts calls to the terminal, the file system, and the system library. These three subsystems are commonly used in other applications, and we handle them in a way which is not specific to Nethack.

The code which specifically handles game information from Nethack is placed in a separated module. If our usage of Nethack code is considered to be subject to the software license, only this part of the code would have to be disclosed upon request. This addresses the legal challenge L1 (see Section 1.2.2).

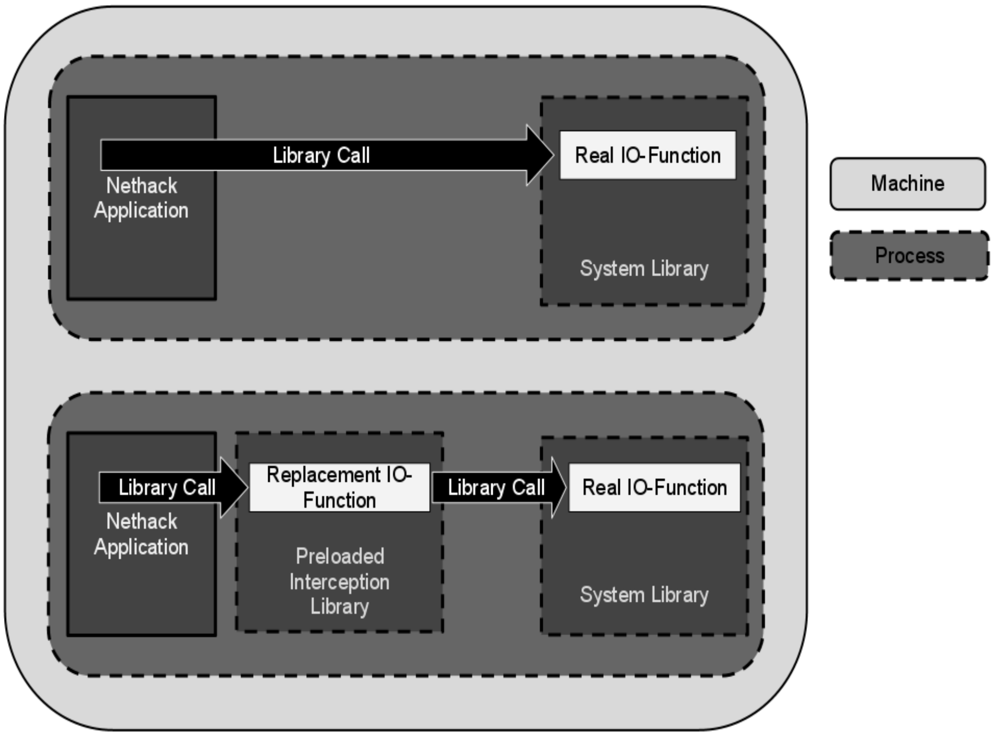

Figure 3.3: Example of library preloading.

Implementation

The Nethack+ wrapper relies on a technique called ’library preloading’. This technique is available on many different platforms, including Linux-based systems. An example of the approach is shown in Figure 3.3.

Most applications are linked to a set of libraries. When the application is started, the application and the libraries are loaded into memory. The application is then linked to certain functions which reside in the libraries, and it is executed.

With library preloading, functions from a library can be replaced without re-compiling the application or the libraries. The replaced functions can also access the original functions, which makes this technique suitable for building filters into already existing applications.

With this technique it is possible to start an unmodified version of Nethack, while preloading one of our libraries. The library overwrites all functions which access the terminal, the file system, and several others.

When our library is preloaded, Nethack still communicates with the player by calling the terminal. However, the calls to the terminal are not forward to a screen, but to a NHD, which relays them to the HackSite.

Chapter 4

Experimental Evaluation

This chapter contains a selection of experiments and their results. In the following two sections we will describe our experiments, and present our main findings. Then, we present the individual experiments which show the performance of the Facebook integration, the behaviour of a single Worker, the limitations of process creation, and the behaviour of a large-scale setup. Last, we show the effect of simple failures on performance of the system.

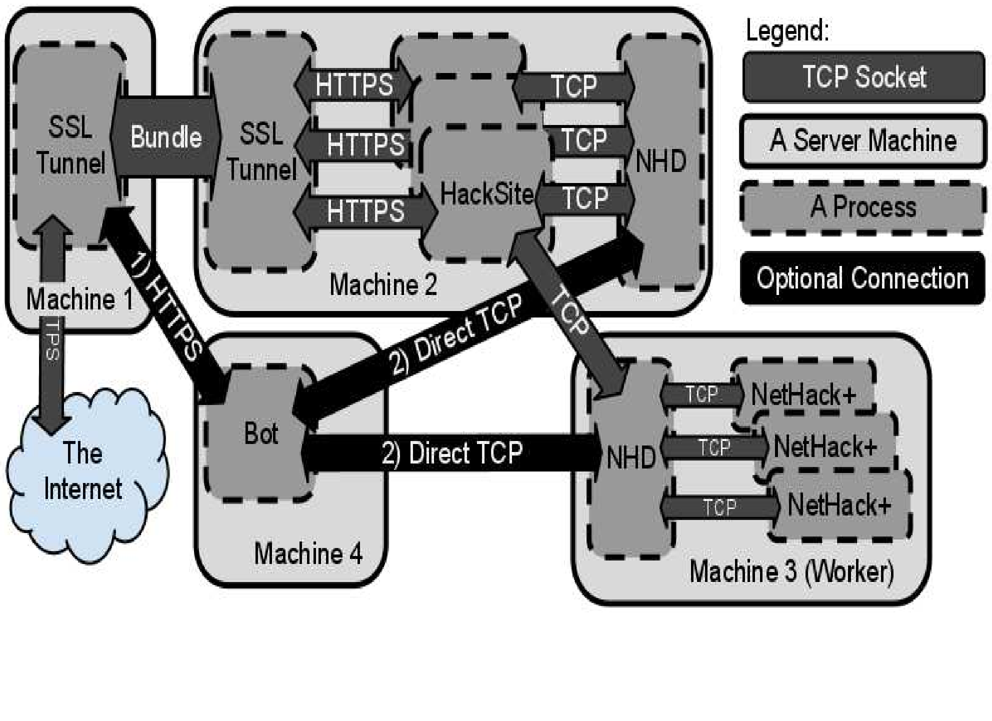

Figure 4.1: WebHack system with one Worker node.

4.1 Overview

We ran a number of experiments with the WebHack system. During those experiments, the basic system setup was similar; the number of Workers and the reported performance metrics varied.

4.1.1 The Basic Experimental Setup

In Figure 4.1 we show a basic WebHack setup. This setup has only one Worker, but still requires four different machines. In Section 3.1.2 we described why it is not desirable to place all components on a single machine.

Machine 1 is dedicated to accepting HTTPS connections and routing the requests to the server cluster. This machine is the only one that has to be reachable from the Internet, it serves as an entry point to the rest of the system. It is one of the Gateway machines, as mentioned in Section 3.2.7.

Machine 2 handles the bundle of incoming connections and delivers them to the local HTTPS daemon, where they will be handled by the HackSite.

An instance of NHD runs on this machine, but no Nethack+ processes are started here.

Machine 3 runs an instance of NHD, and a number of Nethack+ processes. This machine is called a Worker and hosts the actual games. In larger setups, this machine might be duplicated a number of times, to allow the system to handle more concurrent players.

Machine 4 runs the Bot software, described in Section 3.2.4, which emulates human players. In some experiments the Bot connects to a NHD over HTTPS, but in others it bypasses the HTTPS layer and communicates with a NHD directly over TCP. The normal and direct method are numbered 1 and 2, respectively, in the figure.

4.1.2 Measurements

We ran a number of experiments with the basic setup we described in Section 4.1.1. During those experiments, we took measurements which we use to show the performance and correct behaviour of our SMOG system.

To evaluate the the responsiveness of the WebHack system, we measured the delay of game commands, Facebook API calls, and HTTPS requests. The delay of a game command is the time between receiving the input, and receiving the first results of the command, both measured at the NHD.

The delay of a Facebook API call is measured at NHDs, and is the time between sending the HTTPS request, and receiving the complete results. In Section 2.1.3

we mentioned that the API request delays of an older version of the Facebook API could be significant, and average at around 1 second.

The delay of HTTPS requests is the difference between the moment of receiving the request, and the moment HackSite is finished creating the response. We expect this delay to be not much larger than 200 ms, as described in Section 3.3.3.

With a single Worker system we tested the effect of the players arrival pattern on the performance, and investigated the limits to the arrival rate.

To evaluate the performance of the system, we show the number of processes, recently updated games, and process-game pairs in the system. The number of processes shows the number of Nethack+ processes; one is created for every active game. The number of recently updated games shows the number of games, which received new commands during the last 10 seconds. The number of process-game pairs shows the number of Nethack+ processes that is attached to a game.

The combination of these three values give an impression of the resource usage (processes) and performance (updated games). We traced the same three values for a large setup with over 70 Workers.

In our final experiment, described in Section 4.5, we the effects of failures on the WebHack system, and the effect of the code that distributes the workload.

4.1.3 Main Findings

Our experiments showed that WebHack is a responsive system, and capable of delivering a high performance. The system is capable of handling over 300,000 concurrent players, exceeding technical challenge T1, and is able to handle failing servers.

The Nethack+ processes handle 95% of the game commands in 200 ms. Delays of over 1 seconds occur, but not infrequently ($< 0.1\%$). The larger delays are masked by the GUI, which is shown by plotting the HTTPS request delays. All HTTPS requests, even with high workloads, are handled in 200 ms, with an average request duration of only 20 ms.

We show that the system performs well when the emulated players follow a daily pattern, but tendency to play short games increases resource consumption and could lead to performance degradation.

The system can experience minor problems, and temporary reduced performance, if the arrival rate is raised above a certain limit. Traces of system operation containing these temporary problems are shown in Sections 4.3.5 and 4.4.

WebHack is able to perform as a large-scale SMOG system. While using the multi-cluster DAS-4 supercomputer with over 70 Workers, the system was able to service 309,000 concurrent users, satisfying our technical objectives T2 and T3.

In the experiment in Section 4.5, we show that WebHack is correctly able to function even if repeated machine failures occur. This happens without loss of game or logging information.

4.2 Facebook Integration

In this section we analyse the Facebook integration of the WebHack system. We discuss the structure of the API, and present the results of our experiment, where we measured the API request delay.

4.2.1 Application Construction

The Facebook website offers construction guidelines and examples for a wide range of website applications and programming languages. Although the information is extensive, multiple versions of the API are in use, creating a diversity of options. The guidelines mostly supply small snippets of code, which can be combined to form the required website functionality. This allows developers to quickly build functional websites, but it lacks a structured uniform approach.

The fragmented structure of the API is partly caused by the face the interface is under constant development. One of the recent API changes greatly affected this project; Facebook started to require Applications to use secure connections, rather than preferring them. These secure connections use the HTTPS protocol, and require SSL certificates to operate.

4.2.2 Facebook API Performance

In a study on network footprints [13] it was shown that a Facebook Application can suffer significant delay on its requests to the Facebook API. The delays differ between applications, and the difference is mostly related to the application's popularity. They show a delay of 0.6 seconds for one application, and a delay between 0.6 and 4 seconds for another.

The WebHack system measures the duration of the Facebook API requests it executes. In Table 4.2.2 the distribution of the delays of basic account information requests is shown. These requests require a single HTTPS GET operation to be performed. We obtained the measurements by logging all Facebook API delays for several weeks.

Table 4.1: Facebook API Delay.

What	Min	25%	Mean	75%	99%	Max
HTTPS GET-Request duration	0.693	0.891	0.983	1.120	1.219	1.268

With this information we confirm the notion of Nazir e.a. that these delays are significant [13]. Their research shows that the delays are larger for more popular Applications. Even those delays would not lead to problems in WebHack.

The main reasons for this is the fact WebHack queries the Facebook API in the background and caches the results. The HackSite constructs correct information even when the Facebook account information is incomplete. A Facebook Application which does not handle the Facebook API requests asynchronously would suffer from intolerable delays.

Requiring the use of asynchronous handling of Facebook API calls, the use of cryptography in the session tokens, and forcing the use of encrypted HTTPS connections, makes implementing a high-performance Facebook Application a challenging task.

4.3 Performance of a Single Worker Node

In this section we will analyse and discuss a WebHack setup which uses a single Worker. We first look into the duration of executing Nethack+ commands and web-requests in Sections 4.3.2 and 4.3.3. Then, we look into the effects of players which follow a basic daily pattern in Section 4.3.4 and into the effects of high arrival rates in Section 4.3.5.

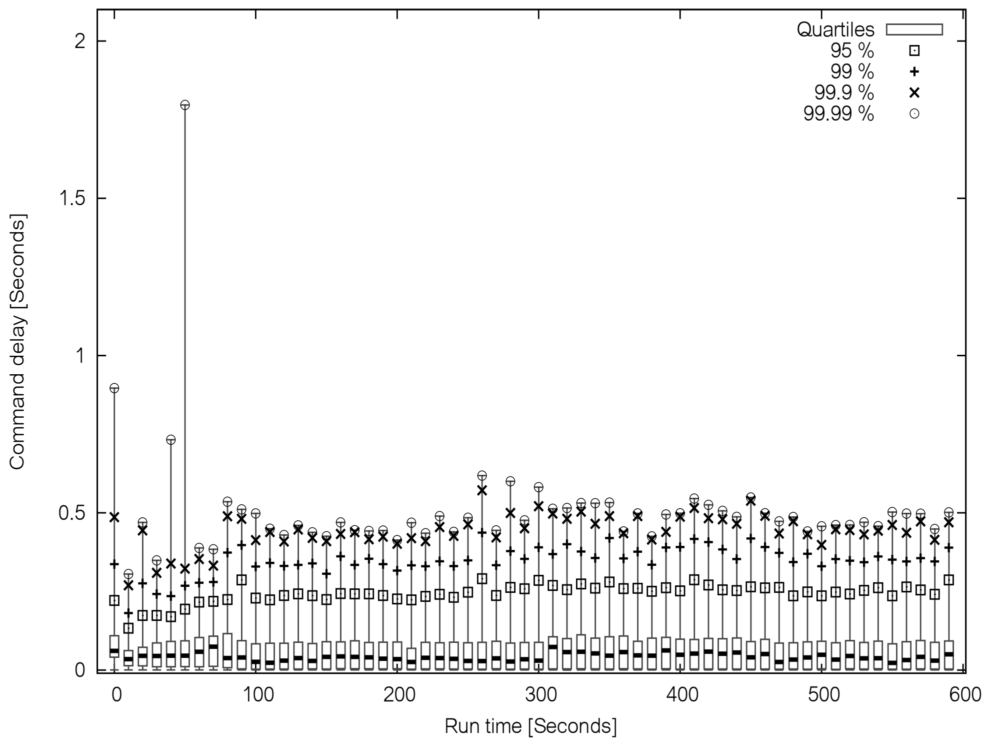

Figure 4.2: Distribution of Nethack+ command delay, 1000 Users.

4.3.1 Experimental Setup

We used the setup described in Section 4.1.1. The Bot uses HTTPS connections in the the first two experiments, described in Sections 4.3.2 and 4.3.3, and direct TCP connections in the other experiments.

4.3.2 Nethack+ Command Delays

The Nethack+ application executes a main loop where it repeatedly waits for commands, and processes them. We measured the delay between the moment of arrival of a command, and the moment the first updates to the game were made. Nethack+ receives the commands through a connection to an instance of NHD. In this experiment each instance of NHD computed and stored the command delays for their Nethack+ processes.

Game commands vary in delay and complexity. Some commands have a simple result, e.g. a single message describing the current room. Commands can also have complex results which consume more resources, e.g. travelling down a flight of stairs to a new level. Such a command triggers the creation of a whole new dungeon level.

Figure 4.2 shows the distribution of the delays in a box-plot. Each box represents the delays from a 10 second interval. The boxes at the bottom of the figure depict the first and third quartile ranges, with a bar at the mean value. The top and bottom bar show the minimum and maximum value, and various symbols depict the 95%, 99%, 99.9%, and 99.99% ranges. Most commands are processed within two tenths of a second, but larger delays occur. The largest delay comes from a command which took over 1.5 second to handle. This delay is too long for comfortable game play, but it only represents 0.1% of the commands which were handled in those 10 seconds. All other commands completed within 0.3 seconds. Large delays only occur when one of the more complicated commands is executed on a machine which runs many instances of Nethack+. Since the complicated commands are not commonly executed, and the probability of a large delay is low, the situation is acceptable for gameplay.

4.3.3 HTTPS Request Delays

Our next measurement focusses on HTTPS requests. We measured the delay of these requests, in order to show that the system is responds quickly, even when handling high workloads or slow commands.

The HTTPS requests are handled by the HackSite, which functions as a front-end to the system. It serves web pages which allow players to navigate WebHack, interact with their Facebook friends, and start new WebHack games. Nethack+ gameplay is also handled through the HackSite.

When a player command arrives at HackSite, the command is forwarded to an instance of NHD. HackSite then waits for a limited amount of time. If game updates arrive during this period they are immediately returned, otherwise HackSite

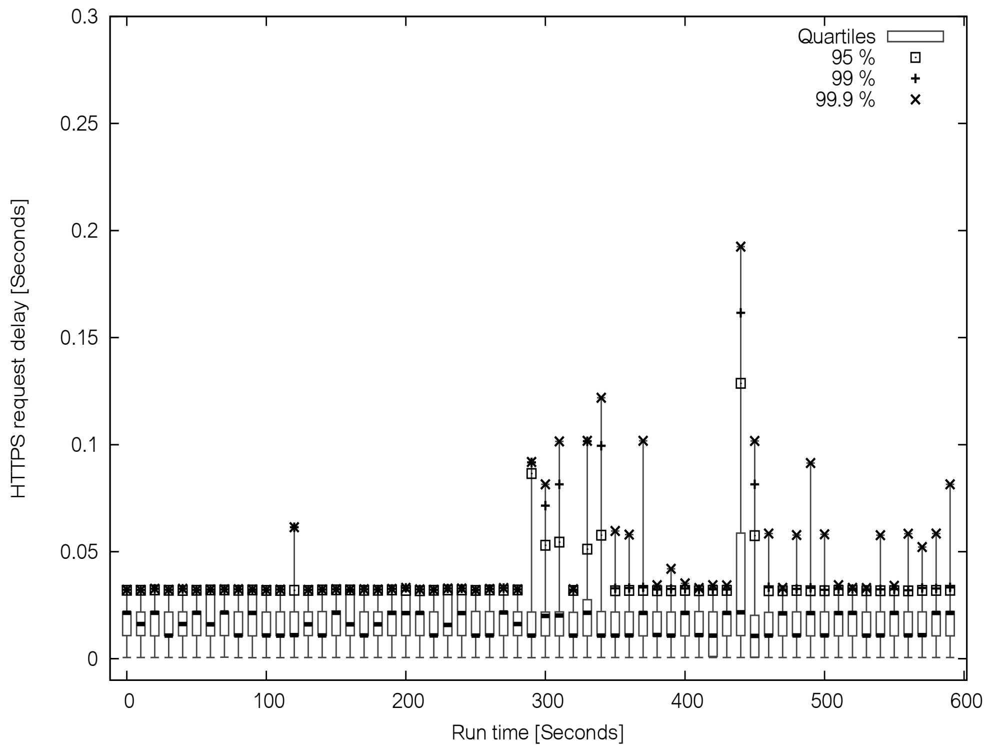

Figure 4.3: Distribution of HTTPS request delays, with 100 Users, using SSL.

returns a successful result without any updates. The complete mechanism is described in Sections 3.2.3 and 3.3.3.

The HackSite uses SSL, an encryption protocol which turns plain-text HTTP requests into encrypted HTTPS requests. Facebook requires applications to use encryption, to prevent simple eavesdropping attacks from capturing sensitive information.

The usage of SSL requires the generation of cryptographic certificates, and requires that communication, both inbound and outbound, is translated with various cryptographic operations. The cryptographic operations can require a significant amount of computation.

We used the setup shown in Figure 4.1.1, and emulated 100 players which executed one command per second. HackSite measured the HTTPS request durations. We show the distribution of these durations in Figure 4.3, using the method described in Section 4.3.2.

The results show that almost all requests were handled in less than a tenth of a second. Many of the larger delays are caused by the absence of updates, but these requests are also handled within 200 ms. This was expected, since HackSite only waits for a limited duration when handling game commands.

After running the system with for an hour with 100 users, we raised the number

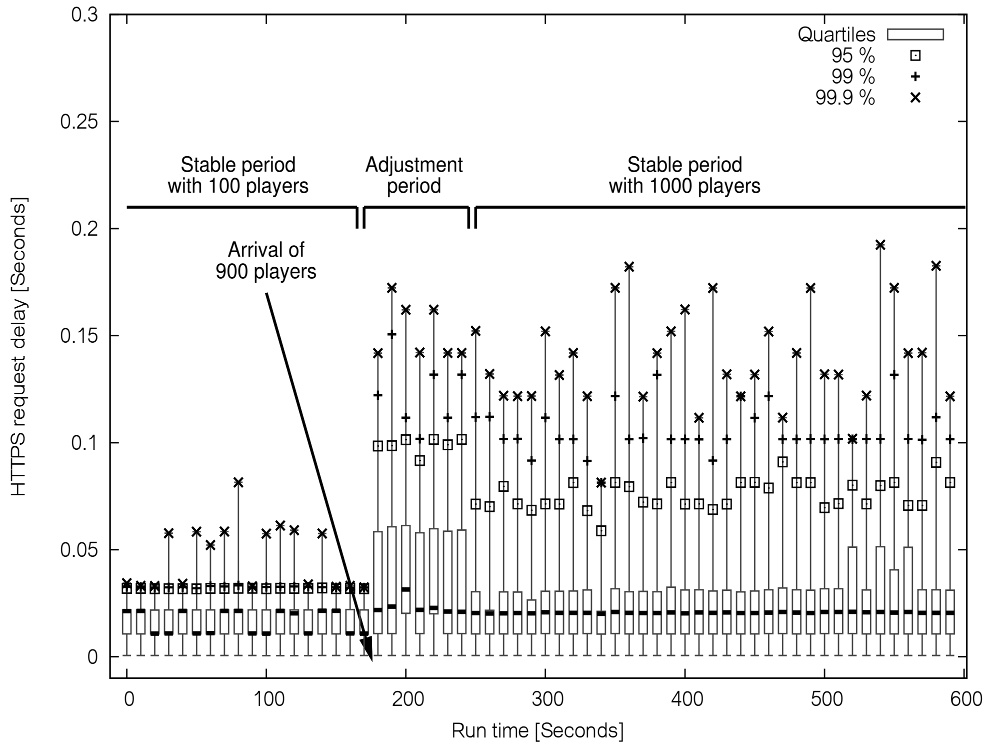

Figure 4.4: Distribution of HTTPS request delays, with up to 1000 Users, using SSL.

of players to 1,000 while measuring the HTTPS request duration. The arrival of new players places an extra burden on the system, because creating a new Nethack+ instance consumes more resources than normal gameplay, and because the system has to find an eligible Worker to handle the new games.

We show the HTTPS request duration in Figure 4.4, in which we annotated the moment where the number of players changed. Right after this moment an adjustment period begins. During this period the delays show a much higher average value, but the distribution of the peaks is similar to the stable period that follows.

Raising the number of users from 100 to 1,000 causes the web-requests to be handled slightly slower, increasing the duration to handle 95% of the requests from 40 ms to 75 ms. However, the mean request duration remains at 20 ms, and all requests are handled within 200 ms. This shows that WebHack remains responsive when handling 1,000 users which are using encrypted HTTPS connections, and that the average HTTPS request requires only very little (<20 ms) processing time.

4.3.4 Effects of a daily pattern on system performance

For this experiment we use the Bot to emulate players, as before, but in a more realistic manner. We added a daily rhythm and the tendency to play short games to their behaviour.

We use the Weibull distribution, for the reasons described in Section 2.2.2, to determine the arrival times of the players.

The Weibull distribution function in given by:
F(t) = $1 - e^{(t/\beta)^{\alpha}}, t > 0$

The Weibull density function is given by:
f(t) = $\alpha.\beta^{-\alpha}t^{(\alpha-1)}e - (t/\beta)^{\alpha}, t > 0$

The α parameter defines the shape of the distribution. With $\alpha = 1$, this distribution is a normal exponential distribution. With $\alpha < 1$ the weight of the function shifts to the beginning, and with $\alpha > 1$ more weight will move to the tail. The β parameter defines the scale. Increasing it will increase the mean value of the function, and vice versa.

From studies on Weibull parameters for HTTP traffic [24, 25] the value 0.7 was selected for the shape parameter. The average duration of a visit was selected to be 5 minutes, although there was no strong basis available for selecting this parameter. Estimates for web server visits ranged from 2 to 30 minutes.

The daily rhythm of our emulated players is based upon their local timezone. We used information of the distribution of Facebook users across the world, shown in Table 4.3.4, and mapped the geographic locations upon time zones.

WebHack can be played on every moment of the day, but the emulated humans mainly play during their evening hours. This leads to a situation where humans can be in three states: Sleeping, Idle, and Playing.

Table 4.2: Distribution of Facebook users across the continents. Source: O'Reilly Research.

Continent	Number of users	Percentage of users
North America	162.5	32.5%
Central America	6.5	1.3%
South America	46	9.2%
Europe	137.5	27.5%
Middle East / North Africa	42.5	8.5%
Africa	9	1.6%
Asia	88.5	17.1%
Oceania	11.5	2.3%

When a human is in the playing state, it sends a command every two seconds, the length of their games is determined by the Weibull distribution, or occasionally on the death of their character. In order to reduce the length of the experiments we used a virtual clock which runs 20 times faster than a normal one, leading to virtual days of 4320 seconds.

In Figure 4.5, we show the number of Processes, Recently Updated Games, and

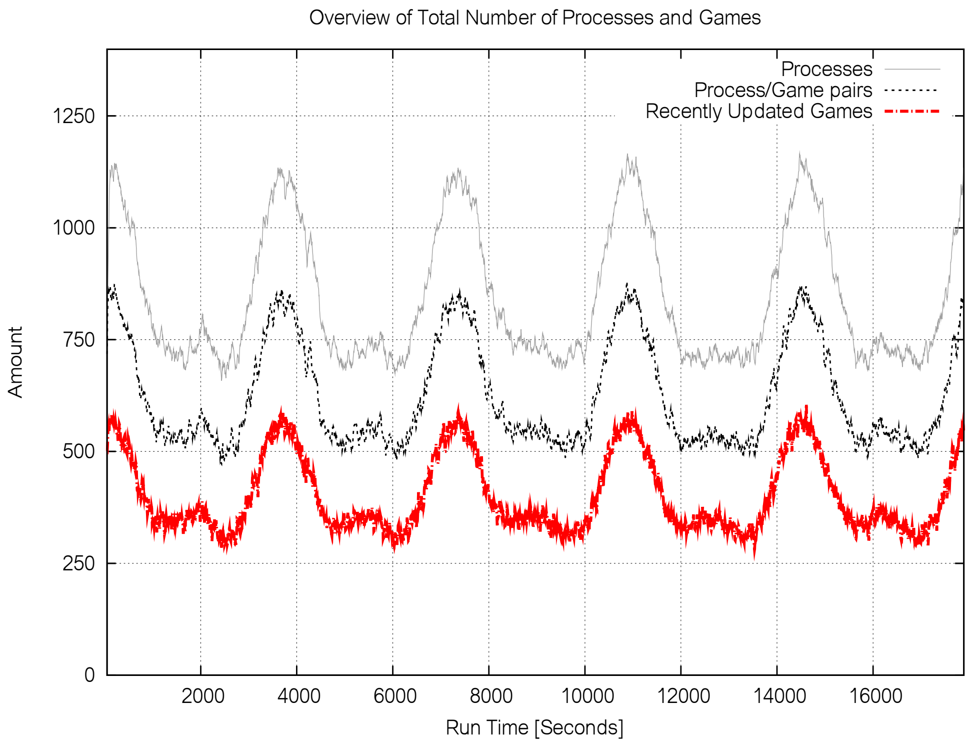

Figure 4.5: Daily pattern.

Hooked Processes. The number of active games in the system is roughly equal to the number of hooked processes. The number of recently updated games is slightly lower, because players tend to play games for a short time. When they move on WebHack receives no notification, and the game lingers for a while until WebHack stops the process and saves the game.

In realistic situations, only a small percentage of the user base will be actively playing at a given moment. The large fraction of short games forces the system to allocate a substantial amount of resources to games which no longer receive any input.

When dealing with a large number of players which sporadically play short games, reducing over-allocation becomes more important. The situation can possibly be improved by adding a mechanism to detect when an active player navigates away from the website, or deselects the web browser as the active application.

4.3.5 Process Creation

In our next experiment we show how the WebHack system responds to changes in the arrival rate of new players, and in particular the problems which arise when the arrival rate is increased too much.

In Figure 4.6 we show a trace where a WebHack system with a single Worker

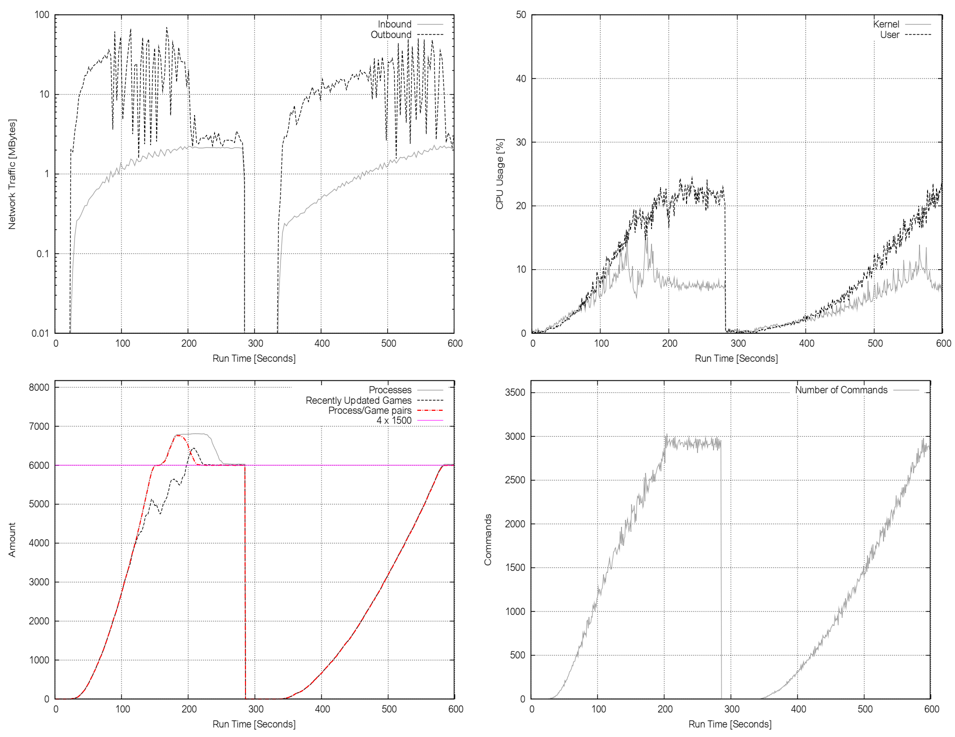

Figure 4.6: Using two different arrival rates with 6,000 players.

received a workload of 6,000 players. There are two runs, started at the first and the 300th second. The maximum arrival rate of players is 60 players per second in the first two runs, and 30 per second in the last.

When a run starts the arrival rate of players is increased from zero to the maximum value over a period of ten seconds. After the initial phase the arrival rate is kept constant until all players are active, at which point new games are only created when another ends.

The second run shows no problems. The number of processes, the number of recently updated games, and the number of processes which are connected to a game are very close to each other. In the first run the high arrival rate leads to a slowdown, which caused some impatient players to create multiple games. After about a minute the system stabilised. The slowdown was caused by the significant amount of resources required to start up a new game, however the exact cause has not been determined.

Using the second run as a known good result, we looked back at the resource usage of the first run. The only clue is shown in the CPU usage graph around the 150 - second mark. Time spent inside the kernel suddenly drops, however the cause is not known. We speculate WebHack's usage of kernel resources decreases because the kernel is spending its resources on other tasks. The situation results in a large delay when handling game commands, which caused the Bot to retry some of its operations.

4.4 Scalability of a large-scale distributed setup

In the previous section the attributes of a WebHack system with a single Worker were discussed. In this section we will look at a large-scale setup, operating on a multi-cluster supercomputer. We discuss the the arrival rates, command delays, and the maximum number of concurrent players.

To scale up the system for a large amount of users, we used the DAS-4 supercomputer, which recently achieved 14th place on the Graph500 list [26]. The supercomputer is split into 6 clusters, of which we will use only two, located in Amsterdam and Delft.

4.4.1 Experimental Setup

The setup shown in Figure 4.1 was modified slightly. For this experiment Machine 2, which serves as a distributer, was replicated 5 times, and Machine 3, the Worker, was replicated over seventy times.

Each Bot was set up to use direct TCP connections to NHD, bypassing the HTTPS layer. This frees up resources for gameplay, avoids the complexities introduced by the SSL layer, and works around certain problems caused by the firewalls between the DAS-4 clusters.

This setup consists of almost hundred machines, and each of the Workers handled thousands of games. The number of machines was limited by availability, not by the system. This was shown by the resource consumption levels on the Distributer machines, which did not reach critical levels during these experiments.

Due to fair-use policies, we limited our usage of the DAS-4 supercomputer to two of its six clusters. We used 72 machines from the Amsterdam cluster, *fs0*, and 20 machines from the Delft cluster, *fs3*. The inter-cluster traffic was reduced by allowing the Bot to directly connect to nearby NHDs.

4.4.2 Worker Setup

Workers are able to handle up to 6,000 games. At that point the OS enforced a limit on the maximum number of processes. CPU and memory usage were significant, which suggests an even higher number of games per Worker can be made possible by further tweaking the OS parameters.

The maximum number of open files per user, and for a complete system, is also limited. On our request, these limits were temporary raised for parts of the DAS-4 supercomputer. The Nethack+ code was modified to use the number of open file descriptors more efficiently.

Nethack+ opens file descriptors to access files on disk, communicate to NHD, and access the console. Although Nethack+ cannot operate without a virtual console, it was proven that it is able to share the virtual console with all other Nethack+

instances. Files on disk can be virtualised in memory, and the other descriptors can be kept open only when directly needed.

Using a combination of these techniques, the average number of open file descriptors per instance of Nethack+ can even drop below 1.0, which practically removes the file descriptors limit as a limiting factor.

Our setup did keep the TCP connection to NHD open at all times, as well as spending several file descriptors on file system access. One of those descriptors was used to open a large level datafile. By using the file system, rather than the wrapping layer for access, a system-wide memory cache is used. The file is then only loaded into the memory once, which significantly reduces memory usage.

4.4.3 Workload Generation

In this experiment Workers were running multiple instances of NHD. Depending on OS parameters either two or four instances of NHD, each with a process limit of 1,500, were started.

The OS-parameters were related to the DAS-4 clusters. The machines from the Amsterdam cluster each hosted 3,000 Nethack+ games, and the machines from Delft hosted 6,000 games each.

The first wave of arrivals consisted of 50 machines from the Amsterdam cluster, which simulated a steady arrival rate of 1,500 new players per second. The arrival rate was kept constant for a period of 100 seconds, resulting in 150,000 concurrent players.

The second wave of arrivals came from 21 machines from Amsterdam, and 16 from Delft. The player arrival rate was kept constant at 1,000 new players per second for 153 seconds. The second wave added another 153,000 active players. This wave was handled by a smaller number of machines, which results in a higher arrival rate per machine. Also, the machines from the second wave were attached to two different clusters from the DAS-4 and are located 70km apart.

4.4.4 Experimental Results

Figure 4.7 shows the total number of processes during the experiment. Every instance of NHD reported its local statistics every second, and the Grapher combined those numbers into these graphs.

The first and second wave of arrivals are visible as the upward slopes in the graph. At the start of the trace, there were already 6,000 players in the system. With the 150,000 players from the first wave, and 153,000 from the second, the total number of players in the system is 309,000. The graph shows a horizontal line at this level.

During the second wave of arrivals, the system shows signs of stress, which can be seen as downward spikes. The spikes occur when NHDs miss the deadline to report their statistics, or when the Grapher is unable to correctly identify the 1-second interval for a report.

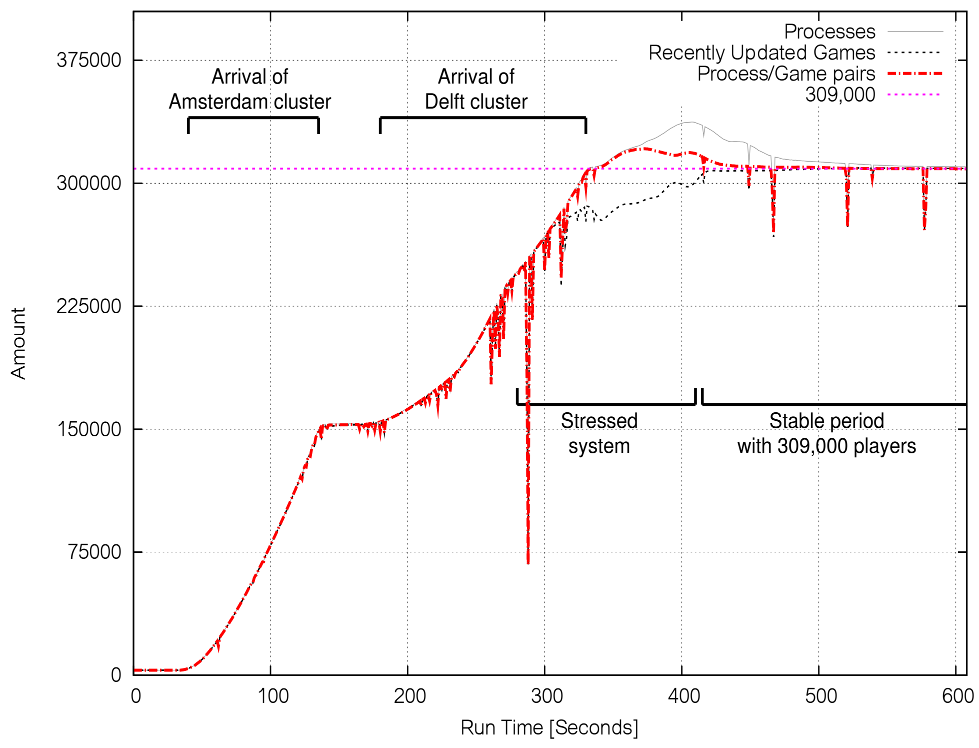

Figure 4.7: Number of processes.

Only the correct reports are included in the summation; delayed or missing reports result in a temporary drop of the totals.

Around the 300-second mark the stress caused some inpatient players to create multiple games. This results in a surplus of processes spawned by the system. After the system stabilises, the surplus processes are reused or terminated.

At the 425-second mark all the system is stable again. All the players are actively playing, and the extra games and processes are slowly being reclaimed.

The amount of stress on the system, can be seen in Figure 4.8, where the command delays are shown.

During the first wave of arrivals WebHack shows good performance. No extra games are created and the command delays indicate good responsiveness. However, half-way into the second wave the performance deteriorates, resulting in larger delays for the players. After about a minute the system recovers and continues to operate normally.

The arrival rate during the first wave was around 1,500 players per second, for a duration 100 seconds. The most popular Facebook Application today, The Sims Social, received an averaged amount of 100 new users per second during the first week of operation [27].

This experiment shows that the WebHack system is able to handle a large amount

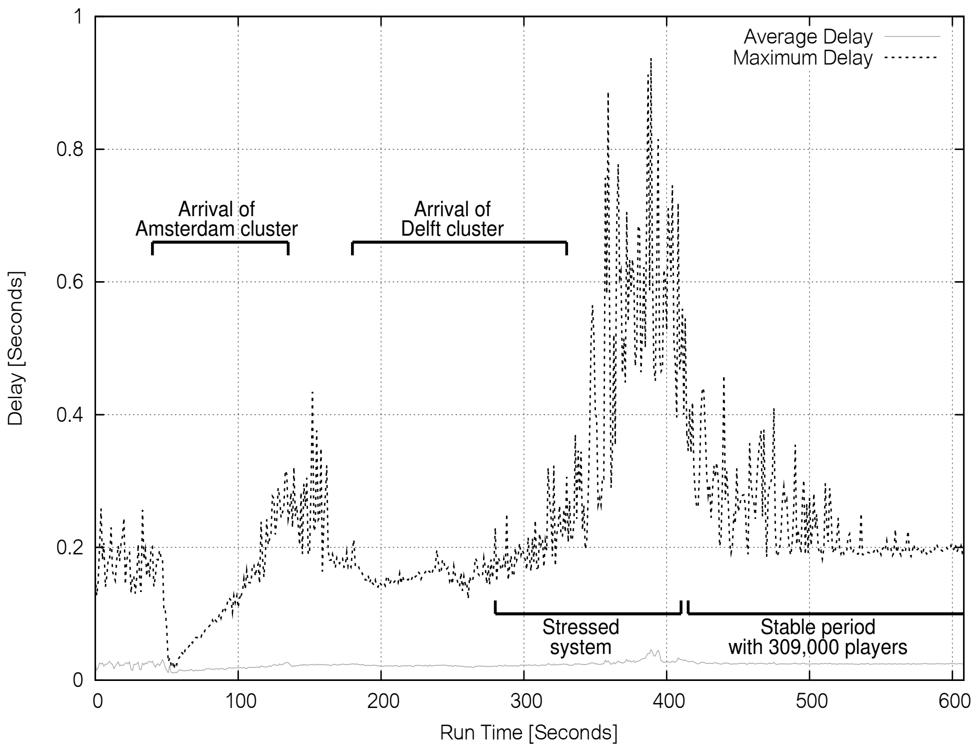

Figure 4.8: Command delay.

of concurrent users, and to be capable of handling high arrival rates, such as those seen at the introduction phase of new SMOGs.

4.5 Handling Failures

In our last experiment we look into the effects of simple failures on the WebHack system. We used a setup with multiple Workers, and repeatedly simulated a complete machine failure. Such an event triggers the functionality described in Section 3.3.2. This functionality causes a backup NHD instance to take over in the event where the primary NHD is not responding to input quickly enough.

In the following sections we describe the experimental setup, and the results. After looking at the functional performance, we discuss the resource consumption of the failure tolerance features.

4.5.1 Experimental Setup

We used a slightly modified version of the setup shown in Figure 4.1. We used three Workers, and configured all of the instances of NHD to operate as primary and as backup game host. The Bot emulated multiple players, and used direct connections to communicate with the NHDs.

The Worker machines suffered from failures at random intervals. When a failure occurs, the NHD forgets all game state and stops communicating for two minutes. At that moment it will restart, and rejoin the server cluster. After starting up, the rejoined server picks up new games again as they arrive.

The Bot runs on a separate machine, and emulates 100 players, which execute one command per second. Two traces were collected with this setup. In the first run the failure tolerance features enabled, and servers occasionally suffered from failures. During the second trace, these features were disabled and the servers did not suffer from failures.

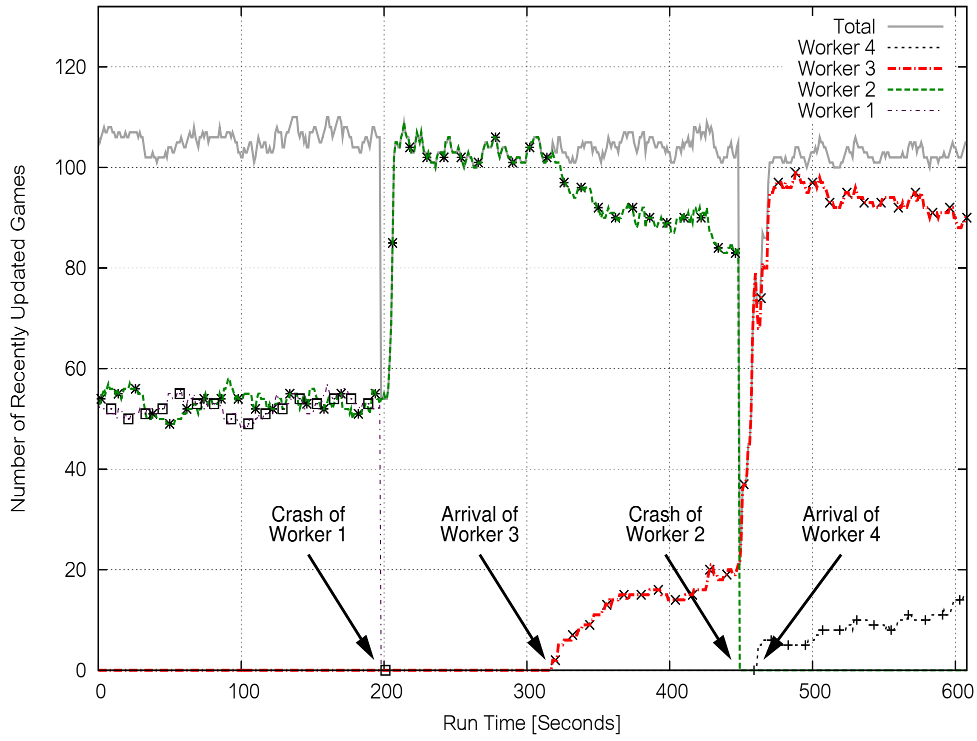

Figure 4.9: Number of recently updated games.

4.5.2 Experimental Results

In Figure 4.9 we show the number of recently updated games, broken down per Worker. We annotated the arrival and departure of the different Workers.

At the beginning of the trace, there are two Worker machines active in the system. The functionality described in Section 3.3.2 results in a relatively equal distribution of work between the two machines.

Around the 200-second mark one of the two initial Workers, Worker 1, suffers from a failure. Within 10 seconds Worker 2, which acts as the backup game server,

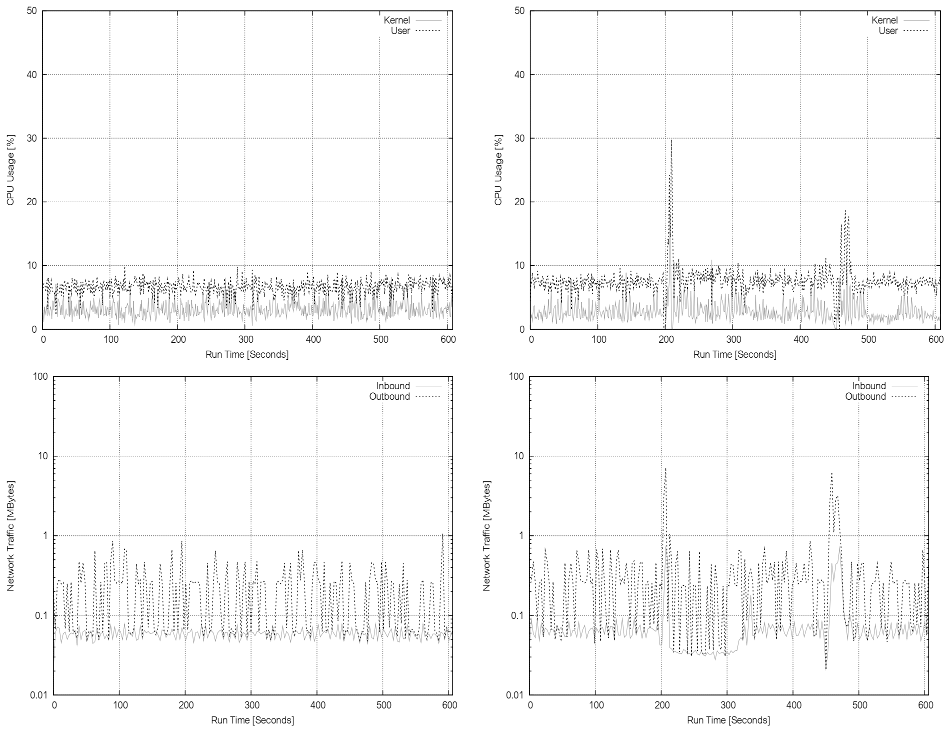

Figure 4.10: CPU Usage and Network Traffic.

replaces the failed machine, and handles the combined workload. A little over 300 seconds into the trace a new Worker, Worker 3, joins the cluster. This Worker starts to quickly pick up new games. Since the total number of concurrent players remains constant, a decline of the workload handled by Worker 2 can be seen. During this period Worker 3 registers itself as the backup game server for the games handled on Worker 2. The effects of this can be seen at around the 450-second mark, where Worker 2 suddenly fails. Worker 3 takes over the workload and the system recovers from the failure a second time.

Due to use of clocks and the random function, as described in Section 3.1.2, it is not trivial to recreate games correctly on another machine. During this experiment we traced the process which recreates the games on the backup machines. We verified that the random seeds and game states before, and after the transfer, where identical.

The failure tolerance features cause a large amount of extra messages to be sent. These messages notify backup game hosts of attempts to contact the primary hosts, and transfer game information from the primary hosts to the backups. Processing and sending these messages costs CPU and network resources. In Figure 4.10 we placed the resource usage of the two runs side-by-side. The left and right side show the trace without and with the failure tolerance functionality, respectively.

The spikes in CPU and network usage during the two recovery periods can be clearly seen in the right hand figures. The differences in resource usage are small, but significant. Notably the User CPU-usage and the Inbound network traffic are a bit higher.

The resource costs consumed by the failure tolerance features are small in comparison with the advantages. The features allow the system to continue operating even when multiple failures occur. The primary game host is replaced quickly, leading only to small discomfort for the players. The inevitability of system failures, and the need for maintenance, make these features a powerful addition to a game platform.

Chapter 5

Conclusion

In the previous chapters we discussed MMOGs, Facebook, and our WebHack system. In this chapter we will look back at our most important results and suggest ways to expand this research in the future.

5.1 Summary

We looked into the process of integrating a MMOG into an online social platform, specifically Facebook. Although the integration process was more complex than expected, a functional and expandable integration was achieved.

We built an efficient SMOG system, using a multi-cluster architecture, asynchronous IO, and zero-copy packet parsing. The workload is handled by many small components, which can be easily distributed over a large number of machine.

We showed that our system is capable of functioning when suffering from multiple failures. The recovery is fast and automatic, and the features consumes an insignificant amount of extra resources.

WebHack is a fully functional web-based game-system, which allows Nethack games to be played over the web. The old game is expanded with social and multi player features. The game platform includes important secondary features like security and maintainability.

In Section 4.4 we show the system to be capable of supporting 309,000 concurrent players, well over our established goal of 250,000. The system is capable of handling the large arrival rates, which are seen when popular games are introduced.

The system is able to operate on the DAS-4 supercomputer using dozens of machines from multiple clusters. The maximum number of machines was limited by the size of the supercomputer, not by our system.

We have build our social platform upon legacy software, which we integrated in a way that allows adding support for other legacy applications, and does not involve rewriting the program.

We did not ignore the software license restrictions of the legacy software. Instead, we reduced the size of code which is considered to be a derived work.

5.2 Future Work

We created a functional, but basic SMOG system. The basic functionality leaves many areas of optimisation and interesting design choices untouched.

Further research could expand the functional possibilities of the Facebook integration, or provide better understanding of the effects of the different social aspects of the interaction.

Our MMOG system uses many different processes and file descriptors to operate. Research into different construction methods or new techniques might lead to improvements in efficiency.

A study of possible improvements to workload distribution and fault tolerance is another avenue for future research.

Although realistic scenarios were used in the experiments, some uncomfortable implementational issues were avoided by restricting the use of HTTPS connections, and by limiting the scope of communication for certain messages. Removing these simplifications is a topic for further research.

A number of ideas, based on P2P techniques described in Section 2.3, were used in the WebHack system. The possibility of expanding these ideas to include a fully distributed database engine in the system can be researched.

5.3 Finally

We hope that in the future, WebHack allows many players to finish a game of Nethack and become the proud owner of an Amulet of Yendor.

Bibliography

[1] Johan Huizinga, *Homo ludens*, (1938, publisher=Beacon Press, Boston).

[2] *http://internetgames.about.com/od/gamingnews/a/trendsdecade.htm*.

[3] Aki Järvinen, *Game design for social networks: interaction design for playful dispositions*, Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games (New York, NY, USA), Sandbox '09, ACM, 2009, pp. 95–102.

[4] *http://nethack.org*.

[5] Matt Barton, *Dungeons and desktops: The history of computer role-playing games*, (2008).

[6] *http://nl.wikipedia.org/wiki/facebook*.

[7] *https://www.facebook.com/press/info.php?statistics*.

[8] Arnoud Bakker, *Survey of system challenges when increasing the amount of players in a virtual game world*, (2011).

[9] Schaap, Charite, Doorn, and Pang, *System architectures for massively multiplayer online games - an overview*, 2008.

[10] van Ee, van den Heuvel, Hooikaas, and Rens., *A survey of system architectures for massively multiplayer online games*, 2009.

[11] Smit Tak, Jutte and de Swart, *A survey of mmog system architectures*, 2010.

[12] *http://en.wikipedia.org/wiki/massively_multiplayer_online_game*.

[13] Atif Nazir, Saqib Raza, Dhruv Gupta, Chen-Nee Chuah, and Balachander Krishnamurthy, *Network level footprints of facebook applications*, Internet Measurement Conference, 2009, pp. 63–75.

[14] Atif Nazir, Saqib Raza, and Chen-Nee Chuah, *Unveiling facebook: a measurement study of social network based applications*, Internet Measurement Comference, 2008, pp. 43–56.

[15] *http://www.appdata.com*.

[16] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica, *Load balancing in structured p2p systems*, (2003).

[17] W. Willinger and V. Paxson, *Where mathematics meets the internet*, (1998).

[18] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson, *On the self-similar nature of ethernet traffic*, IEEE/ACM Transactions on Networking **2** (1993), 1–15.

[19] Thorsten Hampel, Thomas Bopp, and Robert Hinn, *A peer-to-peer architecture for massive multiplayer online games*, Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (New York, NY, USA), NetGames '06, ACM, 2006.

[20] Abdennour El Rhalibi, Madjid Merabti, and Yuanyuan Shen, *Aoim in peer-to-peer multiplayer online games*, Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology (New York, NY, USA), ACE '06, ACM, 2006.

[21] Sieteng Soh Steven Webb, *A survey on network game cheats and p2p solutions*, (2007).

[22] Antony Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, Middleware 2001 (Rachid Guerraoui, ed.), Lecture Notes in Computer Science, vol. 2218, 2001, pp. 329–350.

[23] Peter Druschel and Antony Rowstron, *Past: A large-scale, persistent peer-to-peer storage utility*, Workshop on Hot Topics in Operating Systems (2001), 0075.

[24] Hyoung-Kee Choi and J.O. Limb, *A behavioral model of web traffic*, Network Protocols, 1999. (ICNP '99) Proceedings. Seventh International Conference on, oct.-3 nov. 1999, pp. 327 – 334.

[25] Ryen W White, *Understanding web browsing behaviors through weibull analysis of dwell time*, Work (2010), 379–386.

[26] *http://www.graph500.org/nov2011.html*.

[27] *http://www.allfacebook.com/sims-social-is-facebooks-fastest-growing-application-2011-08*.