

SYMPTOP: A Simulation Toolkit for Peer-to-Peer Networks

Gijs van der Ent

1st June 2005



SYMPTOP: A Simulation Toolkit for Peer-to-Peer Networks

Master's Thesis in Computer Science

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

Gijs van der Ent

1st June 2005

Author

Gijs van der Ent

Title

SYMPTOP: A Simulation Toolkit for Peer-to-Peer Networks

MSc presentation

6th June 2005

Graduation Committee

prof. dr. ir. H. J. Sips (chair) Delft University of Technology

ir. dr. D. H. J. Epema Delft University of Technology

drs. P. Garbacki Delft University of Technology

drs. dr. L.J.M. Rothkrantz Delft University of Technology

Abstract

This thesis describes the design and implementation of the SYMPTOP system, a toolkit for the distributed simulation of Peer-to-Peer (P2P) file-sharing networks. We discuss the performance of P2P systems, and from this discussion derive P2P performance parameters and metrics which we include in the SYMPTOP system. SYMPTOP uses existing P2P clients in its simulation, as opposed to systems that require the user to implement the protocol in the simulation system, in order to enable us to simulate as many different P2P systems as possible, including those that are closed-source. Performance metrics of the simulated P2P network are derived by SYMPTOP from the network traffic between peers in the simulation. Results from sample SYMPTOP simulations of the Gnutella and Overnet networks are presented and discussed, as well as the performance of the SYMPTOP system itself.

Preface

This report is my Master's thesis at the Parallel and Distributed Systems group of the Delft University of Technology. It is the result of the design and implementation of the SYMPTOP simulation system. From the Parallel and Distributed Systems group I would like to thank my advisers ir. dr. D.H.J. Epema and drs. P. Garbacki. I would also like to thank ing. P.V. Anita for the DAS-2 support throughout the realisation of the project, and last but not least my family and girlfriend for their support.

Gijs van der Ent

Delft
1st June 2005

Contents

| | | |
|----------|----------------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Simulating Peer-to-Peer Systems | 3 |
| 2.1 | Peer-to-Peer Performance Aspects | 3 |
| 2.1.1 | Protocol Performance | 3 |
| 2.1.2 | User Participation | 4 |
| 2.1.3 | Content Distribution | 5 |
| 2.2 | Parameters | 5 |
| 2.3 | Metrics | 6 |
| 2.4 | Peer-to-Peer Systems Simulated | 7 |
| 2.4.1 | Gnutella | 7 |
| 2.4.2 | Overnet | 7 |
| 3 | System Design | 9 |
| 3.1 | System Requirements | 9 |
| 3.2 | Peer-to-Peer Client Implementation | 10 |
| 3.2.1 | Own Implementations | 10 |
| 3.2.2 | Existing Clients | 11 |
| 3.2.3 | Conclusion | 11 |
| 3.3 | System Structure | 12 |
| 3.3.1 | The Simulation Generator | 12 |
| 3.3.2 | The Coordinator | 13 |
| 3.3.3 | The Simulation Nodes | 14 |
| 3.3.4 | The Peers | 14 |
| 3.3.5 | The Result Processing Tool | 14 |
| 4 | System Implementation | 16 |
| 4.1 | SYMPTOP Architecture | 16 |
| 4.1.1 | The Simulation Generator | 16 |
| 4.1.2 | The Coordinator | 17 |
| 4.1.3 | Simulation Nodes | 18 |
| 4.1.4 | Peers | 20 |
| 4.1.5 | Result Processing Tool | 20 |
| 4.2 | Gnutella | 21 |
| 4.2.1 | Configuration | 22 |
| 4.3 | Overnet | 22 |

| | | |
|----------|----------------------------------------------------|-----------|
| 4.3.1 | Configuration | 22 |
| 5 | Example Simulations and Results | 24 |
| 5.1 | Small Simulation | 24 |
| 5.1.1 | Simulation parameters | 24 |
| 5.1.2 | Simulation Results | 25 |
| 5.2 | Large Simulation | 25 |
| 5.2.1 | Simulation Parameters | 25 |
| 5.2.2 | Simulation Results | 26 |
| 5.3 | Special Scenario: Search Traffic | 26 |
| 5.3.1 | Simulation Parameters | 26 |
| 5.3.2 | Simulation Results | 27 |
| 6 | Conclusions and Future Work | 34 |
| 6.1 | Conclusion | 34 |
| 6.2 | Future Work | 35 |
| | Bibliography | 36 |
| A | Simulation Scenario File Format | 38 |
| B | SYMPTOP Installation and Usage | 40 |
| B.1 | Installation | 40 |
| B.1.1 | The <i>mysrc</i> Directory | 40 |
| B.1.2 | The <i>bin</i> Directory | 41 |
| B.1.3 | The <i>tcpp</i> Directory | 42 |
| B.2 | Required <i>sudo</i> Usage Rights | 42 |
| B.3 | Usage | 42 |
| B.3.1 | Simulation Scenario File Generation | 42 |
| B.3.2 | Starting the Simulation | 43 |
| B.3.3 | Starting the Simulation on Other Systems | 44 |
| B.3.4 | Result Processing | 45 |

Chapter 1

Introduction

Peer-to-Peer (P2P) systems have been growing in popularity rapidly since their introduction. The most popular type of P2P systems are file sharing networks, which are used to share various types of content over the Internet [2]. Due to the increase in popularity of P2P systems, the network performance of these systems has become a very important issue in the design and realisation of these networks. Network topology and routing have been the subject of a great number of studies in recent years [23, 14, 22, 10], and this has resulted in the development of many P2P networks using different techniques. We are interested in the impact these new developments have on the performance of P2P Systems.

In order to study the performance of P2P systems, we are going to use simulations of these networks. Using a simulation environment allows us to simulate specific scenarios, and gives us detailed data on the individual peers. This kind of data is hard to get from a deployed P2P system, since it is difficult to get information from all peer connected to a large network. Also, it would be impossible to measure the impact of certain conditions on the performance of the systems, since we have no control over the network. The goal of our simulations is to enable us to make predictions about the performance of a P2P system as the number of connected peers grows, and to test new routing algorithms and network topologies.

In order to perform simulations of P2P systems, we have designed and implemented SYMPTOP, a creative abbreviation of "Simulating Peer-to-Peer Networks", which will be discussed in this report. Its main difference with existing simulation systems is that SYMPTOP will allow the user to use actual P2P clients in the simulation. This ensures that the user will need less time preparing a simulation, and simulation results will be more realistic, when compared to traditional simulation systems that require the user to implement the P2P protocol himself in the simulation system. This approach does however cause some other problems when implementing the simulation system and obtaining useful information about the simulated network, which will be discussed in this report.

SYMPTOP should be capable of simulating as many P2P systems as possible. Initially we will be simulating the Gnutella [28] and Overnet [30] networks. Gnutella is one of the first true P2P systems, and was very popular in the initial years of its existence. It became so popular at some point that the network performance collapsed because of the flooding algorithm that was used [2, 10]. Overnet is a recent P2P system

that is part of the popular eDonkey [26] client, which uses it for file location within its network. We are interested in the differences between these networks in terms of message routing and network topology.

During the initial tests of SYMPTOP, we encountered a problem related to the timing of commands to peers within the simulation. These commands have a timestamp associated with them. Using the exact time specified for commands resulted in commands being executed before other commands that were specified to execute before the first commands. While this only occurred when timestamps were close to one another and when the load on the system was high, this posed a threat to the consistency of our simulations. A more detailed discussion of this problem is given in Section 4.1.2.

In Chapter 2 first an overview will be given of the aspects affecting the performance of P2P systems, and from these aspects performance parameters and metrics will be derived. In Chapter 3, the system design of SYMPTOP and problems and decisions made are discussed. Chapter 4 deals with implementation details and specific problems and their solutions encountered during the development of SYMPTOP. In Chapter 5, the results of initial simulations of the Gnutella and Overnet network will be shown. Finally conclusions and recommendations on the future development and usage of SYMPTOP will be given.

Chapter 2

Simulating Peer-to-Peer Systems

The aim of this chapter is to introduce the various aspects of P2P system performance. From these performance aspects we derive the parameters and metrics we will be using in our SYMPTOP simulations. In Section 2.1 we discuss various aspects that influence P2P network performance. Sections 2.2 and 2.3 deal with the parameters and metrics of our simulations respectively. Finally, Section 2.4 gives the reader the two examples of P2P networks which we will be simulating initially

2.1 Peer-to-Peer Performance Aspects

In P2P systems, peers collaborate in order to achieve some common goal. The defining attribute of a true P2P system is that of equality between peers. True P2P networks do not have central nodes that are single points of failure because of their central role in the system. Because of this, P2P systems are very resilient to the failure of single or multiple nodes. However, this decentralised nature of P2P systems makes it hard to assess the performance of these systems, as there are no central measurement points.

P2P file-sharing performance is defined differently than performance in other computer applications. Whereas in other systems the speed at which calculations are done or at which frames are rendered are satisfactory benchmarks, this does not work for P2P file-sharing systems, since the system will generally not do many calculations. In P2P file-sharing systems the users care about getting the files they want as soon as possible. This however depends on too many variables about which no reliable prediction can be made, and thus we cannot use this as a reliable metric in a simulation environment. In our simulations, we will be looking at network performance of P2P systems, under the influence of various parameters which describe the behaviour of peers.

Below, the factors that influence the performance of P2P systems are given, divided into three groups: Protocol Performance, User Participation, and Content Distribution.

2.1.1 Protocol Performance

The message routing protocol of a P2P network determines how messages, such as searches for content and messages related to content publication, are routed through the network, and in what manner files are transferred.

The network protocol ultimately determines the *network scalability*, and what the network load is on peers that are connected to the system. If the load per peer increases as the total number of peers in the system increases, the system will only scale to a certain point, at which peers will be unable to forward any more messages in the system. This particular problem was encountered in early Gnutella [28] versions, which initially worked well, but as the popularity of the system grew, the network became congested with search messages, and the performance of the system collapsed. In order to maintain a good performance, P2P systems should take very great numbers of users into account.

Another part of the P2P protocol determines how files are transferred within the system. Early systems allowed a user to download a file from one source only. Nowadays, new P2P systems employ transferring smaller parts, or blocks, of a file from different sources [24, 26], which may or may not possess the complete file at the time. This not only gives better *download performance*, but with a smart distribution of these parts, also makes sure that it is more likely for a file to remain available in the system for a long time.

2.1.2 User Participation

While a good P2P protocol ensures good scalability and overall efficient operation of the P2P network, a P2P network depends on its users to provide new content to the network, and to keep the content available on the network.

A good P2P System will need to deal with *heterogeneity*. Peers, the individual computer systems connected to the network, have very heterogeneous properties. Some are machines connecting to the Internet through a modem and only stay connected for as long as it takes to obtain a file, others have permanent high-speed connections and stay connected to the system almost permanently, and there is everything in between. Storage space of individual peers affects the time a peer will store the content, and thus the lifetime of the content within the system. Measurements have shown that of all peers connected to a BitTorrent [24] network, over 40% had a firewall, or were otherwise incapable of accepting incoming connections [17]. By taking these kinds of heterogeneity into account, or taking advantages of heterogeneity where this is possible, P2P systems can greatly increase their performance. An example of this is the KaZaa network [29], which uses the peers with higher uptime and better connections to index files stored on peers and thus decrease the time and the message complexity it takes to search compared to Gnutella [28].

Another factor in how fast users can obtain files in a P2P environment is how willing users are to contribute their resources (especially upload bandwidth) to the system. A study [2] showed that at some point in time, as much as 70% of the peers connecting to the Gnutella network were not contributing to the system at all, just taking resources away from the system. This sort of behaviour hurts the performance of the system tremendously, and thus P2P developers have designed different mechanisms to give the user some *incentive to share* resources with the rest of the network. These approaches have had some results, but there will always be people who try to get the most out of any system, while contributing as little as possible. Another problem is that in true P2P networks with no central servers, verifying system contribution is hard. The BitTorrent network has a central server, called a tracker, which enables monitoring of

user contribution, and some sites [31, 27] have used this to encourage users to contribute to the system. Integrating these ratios in P2P systems has been the subject of studies [15, 7, 16] recently, but these systems have not yet been integrated into actual P2P systems yet.

2.1.3 Content Distribution

Besides the P2P system used and how many users participate in the system, performance of the system also depends on the distribution of content within the system. When content can be obtained from various peers at the same time, this can speed up the downloading of that content.

Generally, the more popular content is, the more widely spread it will be within the system, and the longer the lifetime of the content will be. Popular content distribution is problematic at initial publication however. Similar to websites being unreachable during heavy interest from the Internet community (such as the CNN site during the 9/11 attacks), P2P systems also have problems serving requests for popular content. How P2P systems handle this load is another aspect of P2P performance. Where early Gnutella versions did not have the elaborate algorithms for dealing with these situations, recent protocols like BitTorrent [24] seem to flourish under these conditions, because as soon as a peer has downloaded a small part, it can start uploading this part to other peers that are downloading, thus generating a lot of upload capacity for popular files.

Another *data distribution* problem is seen in Distributed Hash Table (DHT) networks [5, 11, 12], where information about files available in the network is stored on specific peers, which, with the DHT routing algorithms, allows for deterministic of search messages. In these DHTs, it is imperative that this data is distributed and replicated in such a way that this data is not lost when peers leave the system, since obtaining this data from scratch would severely damage the performance of the system. Also, in distributed P2P file systems, distribution plays an extremely important role in ensuring the survival of data.

2.2 Parameters

Of the performance aspects mentioned in the previous section, we are interested in the protocol performance of P2P systems. This means we are going to investigate scalability of P2P networks as the number of peers connected to these networks grow. Network evolution and performance is dependent on many aspects, but most of all on the behaviour of individual peers in the system. With SYMPTOP, we are going to simulate scenarios, in which we are able to specify the behaviour of each individual peer. Below we list the parameters of SYMPTOP:

1. **Initial Number of Peers:** The initial numbers of peers that are part of the network at the start of the simulation. These will form the initial network to which other peers will connect.
2. **Join Rate:** The rate at which peers that have not been connected to the network join the network for the first time. This is an indicator of how fast the peer population will grow.

3. **Lifetime:** The distribution of the time which peers stay connected to the network during one session. After this time the peer will disconnect, and may later reconnect to the network depending on the rejoin rate.
4. **Rejoin Time:** The time after which peers that previously disconnected from the network reconnect to the network.
5. **Search Rate:** The rate at which peers connected to the network query the network for files.
6. **Publish Rate:** The rate at which new files are inserted by peers into the network.
7. **Client Settings:** P2P Clients allow the users to modify the settings of the client. This modifies the network structure or routing efficiency. These settings should be set by SYMPTOP. An example of one of these settings is the TTL counter in Gnutella, which modifies the number of hops a message will travel before it is discarded.

2.3 Metrics

The results of our simulations should give us useful data on the performance of a particular P2P system, given the scenario we want to simulate. We are interested in the topologies of the P2P networks, as well as bandwidth usage the system uses for search and other messages. We are going to extract the following metrics from the data acquired by the simulations:

1. **Messages complexity of actions:** Actions like joining, leaving, and executing queries in the simulated P2P network all cause network traffic. The message complexity of these actions determines the scalability of the protocol.
2. **Pathlength:** The pathlength between peers connected to the network. This shows how far a message will have to travel at least from one peer to another.
3. **Connectivity:** The number of active connections peers in the P2P network maintain to other peers.
4. **Clustering Coefficient:** An indicator of how much peers tend to cluster within the network, the clustering coefficient of a peer equals the number of existing connections between the peers it is connected to, divided by the total number of possible connections among these peers.
5. **Bandwidth used:** The number of bytes sent and received by peers in the network. This includes maintenance overhead that peers are subjected to in order to stay in the system, by making sure the peers they are connected to are still alive.

2.4 Peer-to-Peer Systems Simulated

In this section we introduce the two P2P Systems we will be simulating initially: Gnutella and Overnet. Our reasons for choosing to start with these P2P networks, as well as the structure of the networks will be given.

2.4.1 Gnutella

We chose Gnutella [28] as first client to be implemented due to the status it has. Gnutella was one of the first fully decentralised P2P systems, all peers are completely equal, no central servers are used for anything, thus no single points of failure exist. Gnutella therefore is the classical example of a P2P system, which has been studied before many times [19, 20, 10]. This means that a lot of information and measurements of Gnutella performance and statistics are available to which we can compare the results of our simulation system.

The Gnutella network structure and routing protocol are very both very simple. A new peer connects to another peer that is already in the Gnutella network. After this the new peer starts getting new references to peers, to which it maintains active open connections until it has about 4-6 of these connections. Peers that are not connected to actively are kept in a buffer for further reference, they are used when one of the peers the peer is connected to disconnects from the system. Search messages in the Gnutella routing protocol are routed using a flooding algorithm. The peer initiating the search sends out a message to every peer it maintains an active open connection to, containing the search string, and a Time To Live (TTL) counter. Every peer receiving a message like this then forwards it to other actively connected peers, decreasing the TTL, and discarding the message when either the TTL of the message reaches 0, or the peer had already seen the search message before.

Because of the simple flooding algorithm for searching, the performance of Gnutella started to waver when the population of peers grew. As the connections of individual peers became saturated with search messages, the system became sluggish. Improvements to the original protocol were made, but we will be using this older protocol since it has become a classic example of a system that worked well initially, but then collapsed due lack of scalability.

2.4.2 Overnet

Overnet [30] is a closed-source client which has recently become very popular in conjunction with eDonkey [26]. Overnet is a DHT network, based on the Kademlia network [13, 3]. A Kademlia network is constructed as follows: each peer generates a random identification of 160 bits long, which is from then on used every time the peer is started. Every peer maintains a neighbour map that is used to deterministically route a message from one peer to another peer, based on their identifiers.

The neighbour map is filled using the following principle: Each peer knows the peers that are close to it, but the fraction of peers it knows decreases with distance. The notion of distance in the Kademlia network is defined as the integer that one gets by taking the exclusive or (XOR) between the 160-bit identifiers of two peers. This means that peers that differ in the last bits are considered closer to one another than

peers that differ in the first bits of their identifiers. Neighbour lists are kept in 160 k -buckets, where each k -bucket contains neighbours at a predefined distance interval; k -bucket i will contain peers at distance 2^i to 2^{i+1} , for each $i = 0, 1, \dots, 160$. The maximum size of these buckets is denoted by k , where a system with a higher k will be more resilient to peer failure, as these will have more connections to fall back on when other neighbours fail.

At lower values of i , the k -bucket will be nearly empty, as the peers will only use a small fraction of all the 2^{160} possible identifiers. When i gets higher, the number of possible identifiers, and thus peers that qualify for a place in the bucket, grows. At this point Kademlia peers will prefer to keep active neighbours in their list, as peers that have been in the system longer are more likely to stay in the system [20], thus keeping neighbours that have a higher uptime ensures a less transient population of the neighbour map.

In the resulting network of peers, each peer routes messages by forwarding them to the nearest neighbour it knows of the target identifier. As soon as there is no closer neighbour known, the message has arrived at its destination (this may not be the actual target, as because of the sparsely filled identifier space, the target may not exist). More details about the network construction and routing can be found in [13].

We have two reasons for including the Overnet network in the initial test of SYMP-TOP. First, it allows us to show that we can simulate networks that have a closed source protocol and client, something that has not been possible in the past without reverse engineering the client and/or reverse engineering the network traffic. Second, DHTs have become very popular in the research community the past years, and are currently considered the most scalable routing algorithms for P2P networks.

Chapter 3

System Design

In this chapter we present the design of SYMPTOP and its components in detail. Section 3.1 deals with the system requirements of SYMPTOP, Section 3.2 discusses the considerations made with respect to the implementation of the P2P clients, and Section 3.3 presents the structure of SYMPTOP.

3.1 System Requirements

We want to do simulations to observe the network performance of P2P systems. These systems are designed in such a way that they can handle a very large number of users at the same time. In order to accurately measure network performance and make sure that the P2P system behaves in a realistic fashion, we need to simulate networks with a great number of peers at the same time.

There are two kinds of network traffic in P2P systems: system control and maintenance traffic, and download traffic. We are not interested in the download traffic of P2P systems, as this mostly depends on the users of the system, and the availability of content in the network, rather than the routing performance of the network protocol. The system and maintenance traffic we are going to study consist of messages that relate to peers joining, leaving, searching, publishing, and checking whether other peers are still connected to the network. Since these messages are not the actual transferring of files between peers, they can be used to determine the performance of P2P systems [8].

SYMPTOP should enable users to simulate many P2P systems in an easy way. We are also interested in the performance of P2P systems which are based on closed protocols, which is problematic, since the protocols these systems use are not publicly available, and so cannot easily be simulated.

SYMPTOP has to satisfy the following requirements:

1. **Capable of simulating many different P2P systems:** We want to be able to simulate as many P2P systems as possible, in order to be able to make comparisons between the different systems that are currently being used and developed.
2. **Individual control of peers:** We want to be able to control the behaviour of individual peers: when they join, leave, search, and publish content.

3. **Peer behaviour should be reproducible:** The user should be able to run the same scenario multiple times, for instance with different P2P systems. Being able to guarantee that peers behave in the exact same fashion between simulations allows for more accurate simulation results.
4. **Parameters:** The parameters given in Section 2.2 should be implemented in the system.
5. **Simulation results:** Useful result that show the network structure and performance should be logged. At least the metrics defined in Section 2.3 should be determined.

In order to simulate many peers at the same time we decided to design and implement SYMPTOP to run in a distributed computing environment. The capability to run on multiple computing systems at the same time should be taken into account during the design of the simulation system, to prevent problems with the implementation of the system. These computing systems are assumed to be homogeneous.

3.2 Peer-to-Peer Client Implementation

In order to simulate P2P networks, we need to include the behaviour of the appropriate client in SYMPTOP. This can be done in two different ways: implementing client behaviour directly into our simulation system, or using the existing clients for P2P systems in our simulations. Both options have their advantages and drawbacks, which will be discussed below.

3.2.1 Own Implementations

One option to simulate P2P systems is to implement the protocol the system uses ourselves into our simulation system. This can be done with P2P systems that have their protocol specification publicly available, or have an open source version of their client out of which the protocol can be extracted.

There are two advantages to this way of implementing the client. First, implementing the clients ourselves into the system will result in better performance of the system, since less overhead is required per peer because no user interface, and error handling (short of that included in the network protocol) would have to be included in the implementation. This would result in the ability to simulate more peers per SN, thus making the number of peers we can simulate in total larger.

The second advantage is that making adjustments to the protocol is easier once we have implemented the system in SYMPTOP. This would allow us to study the effect of modifying the protocol in ways that could benefit performance of that protocol.

There are two drawbacks to this way of implementing client behaviour into the simulation system however. The main drawback of this method is that making our own implementation of a P2P client would take a considerable amount of time per protocol implemented.

The other is that many of today's popular P2P file-sharing systems have closed-source clients, and protocol specification is kept secret for various reasons. The fact that protocol specifications are not available to us would mean that we are either unable

to simulate these systems, or reverse engineer the client or network protocol. This last option would take us a lot of time, and small changes in the implementation provided by the creator of the client would result in another long period of both network and code reverse engineering to determine what exactly changed.

3.2.2 Existing Clients

Using existing clients means taking an original client for a P2P system, and modifying its environment so we can manipulate its behaviour, thus gaining full control over the client, so we can simulate the P2P system. Using this alternative, we have to devise a way to simulate user behaviour to generate certain events in the system. This will involve manipulating the user interface like a regular user would in an automatic way. The easiest clients to do this would be with command line clients. However, as soon as only a graphical user interface (GUI) client is available, this becomes much harder.

Another problem we face when using existing clients for our simulations is these clients possibly creating connections to the outside. Clients usually have some initial server(s) they use to create a connection to the existing network. These connections will have to be blocked in order to prevent the peers in SYMPTOP from integrating themselves with the regular P2P network on the Internet. Instead of blocking these connections, we might also be able to redirect these connections to a local server providing the initial connection to the simulated network. If the only means for peers to get an initial connection to the P2P network is through a dedicated server application that is not publicly available, we will have to create our own bootstrapping server.

The performance of SYMPTOP using existing clients will be worse compared to clients implemented specifically for the simulation. This will reduce the number of peers we can simulate simultaneously. How much this number is reduced will depend on the specific client simulated.

3.2.3 Conclusion

We decided to use existing clients in the implementation of our system, since this will allow us to simulate as many different P2P systems as possible, including those which are closed-source, and thus hard to simulate. This will also allow us to test clients without implementing the protocol ourselves.

Using existing client we will also get a more realistic idea of what the performance is of the actual system, rather than the performance of the protocol in theory, since implementations may differ from the initial protocol specification, and some clients might make minor improvements to the protocol without these being integrated into the official protocol specification.

Using our own implementation of a P2P protocol will still be possible, by either implementing a complete client, or by implementing it in SYMPTOP. In order to allow this last option, we will have to take it into account during the design and implementation of SYMPTOP.

3.3 System Structure

SYMPTOP consist of five main components: the Simulation Generator, the Coordinator, the Simulation Nodes, the Peers, and the Result Processing Tool. These components are related to one another as shown in Figure 3.1, and each component will be described in the following sections.

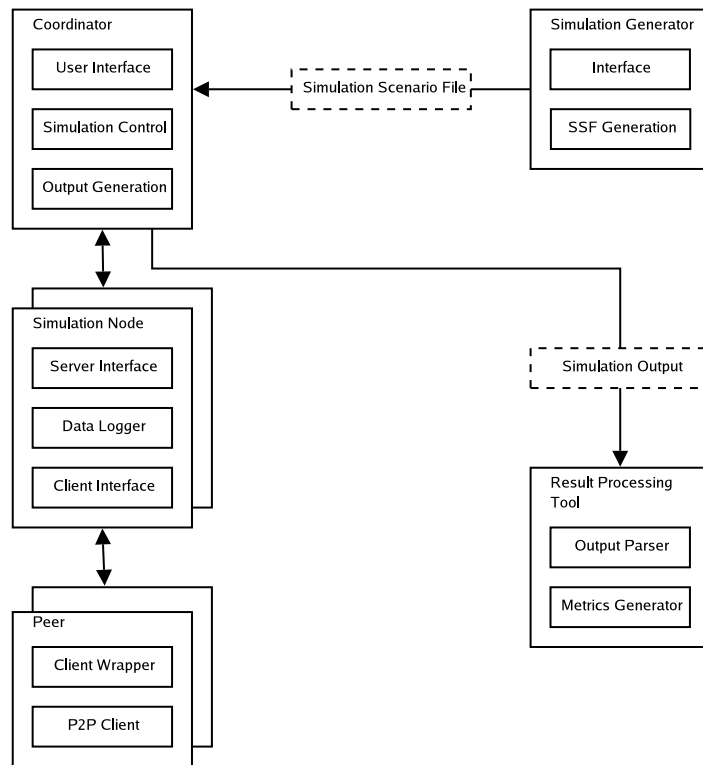


Figure 3.1: General system structure of SYMPTOP.

3.3.1 The Simulation Generator

The main function of the Simulation Generator (SG) is to create Simulation Scenario Files (SSFs) for use with the SYMPTOP simulation environment. These SSFs consist of commands that the simulation will use to control peers, telling them when to do what. The commands in these files can be specified by hand, but as the number of peers in a simulation grows this is no longer feasible, as the SSF needs to contain a command for every action every peer takes during the simulation. The SG consists of two parts: the user interface which allows the user to adjust the parameters of a simulation scenario, and the SSF Generation part which generates commands that specify peer behaviour in a simulation from these parameters.

The user interface is provided to allow the user to specify the parameters that define peer behaviour, and are derived from the parameters specified in Section 2.2. In order to allow for groups of peers with different characteristics, the user can specify different peer types, each with different behaviour parameters. The behaviour parameters are:

initial number of peers of that type, and the rates at which they join, leave, rejoin, search and publish. Each of the rates has two attributes: a distribution the user can choose, and the average of that distribution.

The SSF Generator uses the distributions and averages of the different peertypes to generate commands that define the behaviour of peers. Initial peers of each type are given the command to connect to one another before the actual simulation with peers that join at a later time (specified by the join rate). All commands are stored in an SSF, which is used by simulations.

3.3.2 The Coordinator

The main function of the Coordinator, which is the central server in the SYMPTOP system, is controlling the simulation and providing centralised services such as peer lookup and data storage. The Coordinator consists of three parts: the user interface, simulation control, and output generation.

SYMPTOP simulations are designed to be ran as batch jobs. The user interface thus consists of the command line, at which some simulation specific properties can be specified. During the simulation, information about the simulation is printed on screen that allows the user to track the progress of the simulation. All peer behaviour and simulation events are specified in the SSF generated by the SG, and thus no interaction with the user is required during the simulation.

Simulation control takes care of executing the commands given by the SSF, and starting and ending the simulation. All Simulation Nodes (SNs, described in Section 3.3.3) are controlled by the Coordinator, and through them the individual peers are addressed. Commands to individual peers consist of join, leave, rejoin, search and publish commands. Commands directed at the SNs consist of information-logging, peer-creation, and simulation-termination commands. The Simulation Control part of the Coordinator also provides some services to the SNs, which are listed below:

- **Finding Simulation Nodes:** SNs need to communicate with each other when connecting peers, to find out each other's IP, and the ports peers are listening to.
- **Finding Connected Peers:** When peers want to connect to the network we are simulating, they will in most cases need to bootstrap using a peer that is already connected to the network. The server keeps track of which peers are connected to the system, and which SNs these peers are running on.
- **Data Storage:** The SNs log data about the simulation, whether from peer output or network monitoring utilities. This data is submitted to the Coordinator for storage and later processing.

All data send to and logged by the Coordinator is stored for later processing by the Result Processing Tool (described in section 3.3.5). Processing of the data generated by the SYMPTOP simulation is not done during the simulation, as the amount of data logged can become very large. This data would require substantial computational power to process, and this would mean we would not be able to simulate as much peers at the same time.

3.3.3 The Simulation Nodes

The Simulation Nodes (SNs) constitute the interface between the Coordinator and the peers (described in Section 3.3.4). As such, a SNs consists of three parts: An interface to the Coordinator, an interface to the peers it controls, and a data logger.

The interface to the Coordinator and the peers makes sure that peers controlled by the SN get the commands the Coordinator sends to them. The SN will also take care of special system operations that may be required for certain commands, such as the creation of files which peers can publish in the network. New peers are started by the SN, and terminated when the simulation is done.

The data logger takes care of logging relevant network traffic data to and from the machine the SN is running on. The network traffic of peers is logged on the lowest level, enabling us to later analyse these packets and determine the complexity of actions peers take in terms of network complexity.

3.3.4 The Peers

The peer component of SYMPTOP takes care of all the client-specific details of the system. Peers consist of two parts, the actual client, and the Client Wrapper.

The Client Wrapper is the interface SYMPTOP uses to communicate with the P2P client that is being simulated. It converts commands coming from the SNs to commands which can be used on the P2P client. In case the P2P needs any specific files or directories in order to function correctly, the Client Wrapper takes care of creating these. The Client Wrapper also takes care of the starting and terminating of the actual P2P client.

Different clients will have different methods of establishing an initial connection to the network. This has to be implemented by the wrapper too, since in our simulations, we can not use server lists or other external sources of initial peers to connect to. If clients with hard-coded initial connections points are tested in our simulation setting, these connections will have to be intercepted and either relayed to a similar service within the simulation system, or dropped, to prevent the simulations peers from becoming part of the actual network on the Internet.

Commands the peer can give to the P2P client are those specified in Table A.1, and some additional client specific commands and output parsing, where possible, to keep track of the state of the client. These commands and output parsing are not required in order to do simulations, and do not have to be implemented for every client in the system.

3.3.5 The Result Processing Tool

The Result Processing Tool is used after a simulation has finished, and its purpose is to extract useful information from the network traffic logs created during the simulation. As such, it consists of two parts: The output parser, and the metrics generator.

The output parser takes care of parsing the data created by running a SYMPTOP simulation. This data consist of network traffic from and to the different machines the simulated peers were running on. The output parser will relate the network traffic to the appropriate peers.

The metrics generator uses the data the output parser provides in order to generate the metrics specified in Section 2.3. The computation of some of these metrics, like the average pathlength and clustering coefficient, take a long time because of the complexity of the algorithms used in the computation. In order to reduce the number of times these calculations have to be performed, the interval at which these values are calculated can be increased.

Chapter 4

System Implementation

In this chapter various aspects of the implementation of the SYMPTOP system will be discussed, including the implementation of the P2P clients used. In Section 4.1 we present the implementation of SYMPTOP. Sections 4.2 and 4.3 deal with the integration of the Gnutella and Overnet P2P clients in the simulation environment.

4.1 SYMPTOP Architecture

Most of SYMPTOP is implemented in Java [6], using Remote Method Invocation (RMI) for communication between different parts of the system over the interconnection network.

Our implementation is deployed on the DAS-2 system [1]. This is a distributed computer system, consisting of 200 dual Pentium III nodes. The nodes are distributed among five clusters, which reside at a different locations in the Netherlands. These clusters are interconnected using the network that is used to interconnect Dutch universities. For our experiments and initial implementation, we will employ the TU Delft DAS-2 cluster, consisting of 32 nodes with two processors each, and one file server, which is used for programming, compiling, and starting and stopping jobs.

4.1.1 The Simulation Generator

The Simulation Generator (SG) is implemented in Java, using the CHARVA [25] library to provide a text based interface with menus. The user is able to start the SG with a previously saved simulation parameter file, or start from scratch to create a new SSF. The SG requires the user to specify the length of the simulation. Due to the problem of timing commands in SYMPTOP simulations, the time specified is relative, not absolute. The user can specify different peer types with different properties. At least one peer type should be specified to get a useful simulation. Additionally, users can specify the random seed of the simulation generation. This can be used to generate different simulation scenarios based on the same behaviour parameters.

When all parameters are provided to the SG, it allows the user to generate the SSF, which the Coordinator can read to get peer commands from. All commands are generated randomly, using the values provided by the user. First all types of peers have their initial number initialised. Then the distributions provided are used to generate join,

leave, restart, search, and publish events. At the end of the simulation the drawnetwork command is given to visualise the network, followed by the quit command to terminate the simulation.

Simulations generation parameters can be saved by the user for later use. Saving will create a *.sim* file which contains all the information about the simulation. This allows the user to make adjustments to the simulation parameters at a later date, or generate a simulation scenario with similar parameters but with a different random seed.

4.1.2 The Coordinator

The Coordinator which is the heart of SYMPTOP, commands the rest of the system, and stores the data of the simulations. The Coordinator should be the first to connect to the Java RMI Registry and register itself there so the SNs can find the Coordinator when they connect. The RMI registry can be running on a different server, but it can also run on the same machine as the Coordinator, since it does not take many resources.

Command line arguments to the Coordinator are the SSF file describing the current simulation, and the number of SNs that will participate in the in the simulation. After initial startup, the Coordinator reads the specified SSF in order to get a scenario for the simulation. It then proceeds to wait for all the SNs to have signed on with the system. When all SNs have connected to the Coordinator, the Coordinator proceeds to execute the first command(s), and the main simulation loop is entered.

Once the Coordinator processes a quit command, it first terminates all the peers, and proceeds to send all the SNs a quit message before quitting itself.

Command timing

Commands specified in an SSF (of which the format is given in Appendix A) have a timestamp associated with them. These timestamps are specified in milliseconds, and because of this rather coarse granularity, it is possible that multiple commands have the same timestamp. During the implementation, we encountered some problems using timers exactly in the way specified in the design.

Executing commands exactly on time became problematic as the number of peers running on a single SN increased. As the load on the system increased, the time required to execute a command increased. This resulted in consecutive commands to a peer sometimes being executed in reverse order. This, for instance, leads to commands being given to peers before the client is actually started, or search queries being sent while a peer is not yet connected to the network.

There are two viable solutions to this problem in our system. One solution to this problem would be to give all commands to peers sequentially. The difference between the timestamps of two consecutive commands would be the delay the Coordinator should wait after the execution of the first command, before executing the second command. This implies that commands that have the same timestamp would be executed sequentially. This would guarantee that commands would be given to the peers in the specified order. The main drawback of this solution is that executing commands sequentially would prevent us from simulating parallel events in the simulation. Also,

we would only be making use of one of the SNs at a time, while other SNs would be idle.

The second solution, which we implemented, is a system that guarantees sequential behaviour if the timestamps of the commands are different, but allows for parallel processing of commands with the same timestamp. Commands with the same timestamp are started at the same time, as separate Java threads on the Coordinator. The Coordinator then waits for all threads to terminate, thus making sure all the commands are given to the peers. It should be noted that this does not mean that the peer will have finished execution of the command, as the peer implementation does not check for peers finishing commands. After all concurrent commands are given to the respective peers, the Coordinator will continue execution. Between two consecutive commands with different timestamps, the delay after executing the first command equals the difference between the timestamps of the two commands, like in the first solution. The drawback of waiting for the commands to be actually given to the peers is that we can no longer exactly predict how long our simulation would take to complete, unlike in the original situation where simulation time equals regular time. However, it is more likely that the behaviour will be correct, since no commands are given before commands that should be previously given.

Drawing the P2P network

Our test clients contain commands that allow us to list the connections they maintain to other peers in the system. The *drawnetwork* command, as specified in Table A.1, is implemented using this property of the P2P clients. When the *drawnetwork* command is encountered in the command list, the Coordinator sends each peer a command to list its active connections to other peers. This allows us to create a picture of the network, which can be used in small network to observe the topology and evolution thereof.

4.1.3 Simulation Nodes

The SNs run on the different machines that participate in the current SYMPTOP simulation. Each machine can have multiple of these SNs running, which might be beneficial when a machine has more than one CPU available. SNs take care of the different peers assigned to them, which are started as separate Java threads by the SN.

Each SN maintains a list of peers running on the respective machine taking part in the simulation. The SN forwards commands from the Coordinator to the respective peers, and in the case of *publish* commands, it assists by creating files containing random data for the peer to publish in the simulated network. SNs also help peers with finding peers already connected to the network, which can be used to create an initial connection to the simulated network.

Finally, SNs take care of logging network traffic between peers in the system. This is achieved by two logging subsystems: the *TCPDump* Logger, and the *Lsof* Logger.

TCPDump Logger

TCPDump [21] is the utility we use to capture the network traffic associated with peers. It can log all data of packets that reaches the machine through various network

interfaces. Since multiple SNs can be running on the same machine, and logging network traffic of the same machine multiple times is both redundant and costly in terms of data storage, not every SN needs to log network traffic. SNs from the same IP that connect to the Coordinator get the task of logging network traffic assigned on a first come, first served basis.

If the SN starts a TCPDump Logger, a separate Java thread is started, which takes care of the starting and terminating of TCPDump. TCPDump is started using the Java runtime environment, which allows us to interact with the program using the default input, output, and error streams. In its default mode, TCPDump logs all data coming through the network interfaces of the computer system. This means that not only communication between peer clients would be logged, but also control messages from the coordinator to the SNs, and all other traffic that reaches the computer system the SN is running on. In order to reduce the size of the log files, and the amount of work that has to be done later when parsing these logs, we limited the amount of data sorted by only storing messages with either sending or receiving ports that are in the range of ports used by the peers during simulations. To reduce the amount of data logged even further, we only log the packets sent by the particular machine the SN is running on. This ensures that network traffic between different SNs is not logged, and thus parsed, twice.

The output of TCPDump is written to a local file on the computer system it is running on, and is at the end of the simulation moved to the file server for storage and later parsing.

Lsof Logger

The peers initiate connections amongst themselves using the regular TCP/IP networking protocol. This means that each connection has an IP/port combination at the initiating and receiving end of the connection. Each peer in the system uses a different, predefined, port for listening to incoming connections (see Section 4.1.4) from other peers. Thus, we can identify the peer that a connection is initiated to by checking the port number of the receiving peer. However, when a peer initiates a connection to another peer, it uses a random port, assigned by the operating system the machine participating in the simulation, to initiate the connection from. This means that we are not able to identify which peer the connection is initiated from by means of analysing the IP/port pair.

This problem is solved by making use of the Lsof [9] tool. This tool lists all open files in a Unix/Linux environment, including all open sockets. The way in which information is logged allows us to determine which port is in use by which peer. This allows us to identify the initiating peer of a connection.

Lsof shows open connections at a single point in time. In order to gather as much information as we need during the simulations, Lsof is ran at a fixed rate. Due to the discrete nature of the data, it may occur that a connection is too short lived for us to notice in the Lsof data. This prevents us from identifying one of the two parties involved in the communication. The number of connections of which we cannot identify one of the peers can be reduced by increasing the rate at which Lsof is ran, thus increasing the data set. This does however mean that the amount of data that is logged will increase, and thus parsing time at the end of the simulation will increase as well.

4.1.4 Peers

Every P2P client we are going to simulate in the system will have a different interface. This is the direct result of using existing clients to simulate P2P networks. In order to allow SYMPTOP to communicate with the individual clients, a Java interface has been specified, which client wrapper classes should implement. This interface includes methods to create peers, start peers, make peers search, and all other actions we want to perform on peers.

Every different client will have a different way to configure its behaviour. Some requirements SYMPTOP currently has in order to ensure correct behaviour by peers in the system, is that a P2P client has a command line interface for Linux, and that the P2P client has configurable ports (or ranges thereof). Forcing clients to use different ports will allow us to distinguish them in the network logs, as explained in the previous section, and make sure clients behave correctly.

Client wrapper classes are the classes that create an instance of a peer on a system running the simulation, and after creation allow for the control of that peer. They should generate the required configuration options (such as a local port this peer should be listening on), and pass these to the P2P client we are currently testing.

Commands are passed to the P2P Client through text commands. The peer wrapper class will send the command to the P2P client, and return. This means that the wrapper will not wait for the command to finish execution with the peer. The reason we chose not to wait for this is that it would increase the complexity of the P2P client, and since we are interested only in message complexity of searches and the likes, simulations will not be based on responses from the P2P network, but rather on predefined values.

One exception is the *drawnetwork* command. After this command the client wrapper will wait for the output from the P2P client. This output is then parsed and the resulting list of open connections to other peers is returned to the Coordinator in order to generate the network layout.

4.1.5 Result Processing Tool

The task of the Result Processing Tool is to parse all the data logged during one simulation and extract metrics, as specified in Section 2.3, from the parsed data. The Result Processing Tool is written in Python [18], and works in two phases: data acquisition, and metrics generation.

Data acquisition consists of parsing the logs created during the simulation: the TCPDump logs, the Lsof logs, and the Coordinator log. All packets logged in the TCPDump logs are read, size is stored, and header information is parsed. This header information consists of the following items: A date, a source and destination IP, and a source and destination port. The data stored in the Lsof logs is used to reconstruct a list containing data on which port was owned by which peer at which times. The Coordinator log is used to get information about the state of the peers in the system, and the times at which searches and publish events happened in the system.

In the metrics generation phase the previously parsed data is used to generate useful data about the simulation. The identifiers of the peers engaged in communication logged are either directly resolved using the identifying port number, or looked up in the data that was logged by the Lsof Logger. A special case is that of the com-

munication using the UDP protocol, which the Overnet client engages in. Both the sending and receiving port in this case are the identifying port numbers, unlike with TCP communication where one of the port numbers will be randomly assigned.

A problem with UDP traffic is that there are no actual connections, like is case with TCP traffic. This means that we are unable to tell when a peer "disconnects" from another peer. Especially in the case of Overnet, a DHT, it is not reliable to assume that a connections between two peers ceases to exist when the last data is send between them. In order to verify whether a connection between two peers still exists would be to trigger network traffic between these peers. We can generate network traffic by making peers take actions like connecting, leaving, searching and publishing. However, since routing in a DHT is deterministic, we are unable to predict whether such an event would activate a certain connection between peers. In order to model UDP connections existing between peers, we use the following simplification: a connection is considered started as soon as the first data is send between the two respective peers. A UDP connection ends as soon as one of the peers leaves the system.

From the TCP connections logged in the Lsof logs and the connections derived from the UDP traffic, the following metrics are derived: the average pathlength, the clustering coefficient, and the average number of connections per peer. These values vary over time, and are calculated at a set interval. Especially the pathlength and clustering coefficient algorithms have a high complexity, and to keep the parsing time acceptable, this interval can be increased.

Output of the metrics generator is written to file, in a format compatible with Gnu-plot [4], to allow for easy plotting of the data. Each line represents one point in time, and the following data is logged per line: The number of peers, average pathlength, clustering coefficient, and average number of connections per peer. Also given are: the number of bytes sent, search queries issued, and published performed in the measurement interval preceding the point in time the line represents.

4.2 Gnutella

The Gnutella client consists of an open-source text-based client. Its interface is simple and straight-forward, in that all commands giving to peers by the Coordinator can be directly translated into text commands for the Gnutella client. Configuration of the client can be done at run-time, by using the appropriate commands. The port the client uses to listen on for incoming connections from other peers is specified as a command line argument for the client.

A problem we encountered with the Gnutella client is that it would not connect to a peer that ran on the machine the client was running on. The client is capable of distinguishing between peers on the same IP number by using the unique port number, but it ignores peers with the same IP it has itself. This would limit the number of peers any given peer could connect to in the system. The solution to this problem was simple because the client was an open-source client, all it required was a modification of the client its code to allow for these local connections.

Should this same problem have occurred with a closed source client, our solution would not have been available. In this case, there are 3 options: try to work around the problem with virtual machines, run one peer per IP number, or ignore the problem.

Starting one virtual machine per peer, each with a different IP number, would solve the problem, but take a lot of system resources, and thus reduce the number of peers the system is capable of simulating. The second option, running one peer per IP number, is easier than the first solution, but reduces the number of peers the system is capable of simulating even further. The third option, ignoring the problem, would have implications for accuracy of simulation results. Should the user choose this last option, the number of different machines partaking in the simulation should be as big as possible, in order to reduce the fraction of peers a single peer is incapable of connecting to.

4.2.1 Configuration

Configuring the behaviour of the Gnutella client is done in 3 ways: through the command line at startup, through the client itself using the "set" command, and through the *gnut_hosts* file.

We use the command line options to set the port which is used by the client for listening for incoming connections. A single Gnutella client uses only one port to listen on, and we specify this to be $10000 + client_id$, where *client_id* is the integer we use internally in the SYMPTOP simulation to identify the peer.

The options set through the client itself modify the behaviour of the client, and the interface of the client. The most interesting settings to us are the *TTL*, *min_connections*, and *max_incoming* settings. The *TTL* setting specifies which TTL the client will assign to its own search messages. The *min_connections* setting specifies until which number of connections the client will keep looking for new peers to initiate connections to. The *max_incoming* setting determines how many incoming connections from other peers the client will accept.

The *gnut_hosts* file contains IP and port numbers of all peers the client has ever known. It is used upon startup to initiate initial connections. When a client is terminated, it writes its host cache to the *gnut_hosts* file. In order for us to correctly simulate peers leaving the system, and the same peer reconnecting to the system at a later date, we had to use these files. The problem we encountered was that there was no way to specify which file was the *gnut_hosts* file of the particular peer, as each peer uses the exact same hard-coded filename. This meant that we had to backup this file after a peer left the system, and make sure the file got restored to the file that a particular peer left upon disconnecting from the network. Both these actions have been implemented in the client wrapper.

4.3 Overnet

We integrated the most recent Overnet client available at the Overnet website [30] into SYMPTOP. The Overnet client is designed to allow for multiple clients running at the same machine, as the command line arguments to the client include a means to load different configuration files.

4.3.1 Configuration

All configuration files and host caches are stored in a directory by Overnet. The client does not allow us to specify which port(s) to listen on for incoming connections at

startup, but instead reads these values from the configuration file. Our Overnet client wrapper thus had to generate a configuration file for each peer SYMPTOP is going to simulate.

Overnet uses two different ports to listen for incoming connections, one TCP port, and one UDP port. Both can be specified in the configuration file, and we have specified these to be the following: $10000 + 2 * client_id$, and $10001 + 2 * client_id$, respectively, where *client_id* is the identifier of the peer we use internally in the SYMPTOP simulation.

Chapter 5

Example Simulations and Results

In this chapter we present example SYMPTOP simulations, their results, and some observations about the simulations. Each section deals with one simulation scenario, simulated with both the Gnutella and the Overnet client. Section 5.1 shows a small simulation with a short scenario generated by the SG as an introduction. The simulation presented in Section 5.1 is longer, and contains more peers than the first simulation. Finally, in Section 5.3 we show a simulation with a custom simulation scenario, which shows network traffic resulting from peers searching in the network.

5.1 Small Simulation

The aim of the small simulation is to introduce the reader to the simulations, by discussing a simple simulation with a small number of peers. The simulation parameters and the results of both P2P networks will be discussed and a comparison of the two will be made.

5.1.1 Simulation parameters

We used the SG to generate a simulation scenario for a simulation running for 60 seconds. The parameters of peer behaviour are given below, with their distributions and averages:

| | |
|-------------------------------|--------------------------|
| Initial Peers | 10 |
| New Peers | 5 seconds (exponential) |
| Session Time | 60 seconds (gamma) |
| Time Between Sessions | 20 seconds (gamma) |
| Time Between Searches | 30 seconds (exponential) |
| Time Between Publishes | 50 seconds (exponential) |

The *Initial Peers* parameter defines how many peers are connected at the start of the simulation. *New Peers* defines at what rate new peers are added to the system. The *Session Time* is the time which peers stay connected to the network in one session. *Time Between Sessions* defines how long a peer stays disconnected from the network between sessions. *Time Between Searches* and *Time Between Publishes* define the times between search and publish events for a single peer.

5.1.2 Simulation Results

The number of peers in the system the parameters have resulted in are shown in Figure 5.1a, which is of course a direct result from the parameters. The metrics, and how these vary over time, obtained by running the simulation for both the Gnutella and Overnet networks are shown in Figures 5.2a, 5.3a, 5.4a, and 5.5a. It is evident that because of the short length of the simulation and the very limited number of peers in this simulation, the fluctuation of the metrics over time is somewhat erratic. The clustering coefficient of 0 for the Overnet network in Figure 5.4a is correct, this is a coincidence. Simulations that last longer and include more peers give more accurate results.

As for performance of the SYMPTOP system itself, this simulation took 111 seconds for the Gnutella network and 148 seconds for the Overnet network. Parsing the network logfiles took 4 seconds for Gnutella, and 0.8 seconds for Overnet on a 3GHz desktop PC.

5.2 Large Simulation

The aim of this simulation is to show a simulation that lasts longer, and contains more peers, than the previous one. This should show more accurate data about the used protocol, with less variance in the metrics like that seen in the small simulation. We also made use of two different peer types for generating the simulation scenario, which will be explained in the next section.

5.2.1 Simulation Parameters

The simulation is set to run for 300 seconds. There are two different peer types involved in this simulation: *publishers* and *regular peers*. The names of these peers should be self-explanatory, the former publish files in the system, while the latter only search for files, and hardly ever publish.

The parameters of the *publishers* are as follows:

| | |
|-------------------------------|--------------------------|
| Initial Peers | 10 |
| New Peers | 10 seconds (exponential) |
| Session Time | 100 seconds (gamma) |
| Time Between Sessions | 200 seconds (gamma) |
| Time Between Searches | <i>no searches</i> |
| Time Between Publishes | 10 seconds (exponential) |

The parameters of the *regular peers* are:

| | |
|-------------------------------|----------------------------|
| Initial Peers | 100 |
| New Peers | 1 second (exponential) |
| Session Time | 200 seconds (gamma) |
| Time Between Sessions | 100 seconds (gamma) |
| Time Between Searches | 50 seconds (exponential) |
| Time Between Publishes | 1000 seconds (exponential) |

5.2.2 Simulation Results

The results of the large simulation are shown in Figures 5.1b through 5.5b. Interesting about these results are that even though the average number of connections maintained by Gnutella and Overnet peers is about the same, the average pathlength differs a great deal.

The clustering coefficients shown in Figure 5.4 shows some interesting developments. First, the Gnutella network seems to have a pretty high clustering coefficient when compared to the Overnet network. This implies that a Gnutella peer is more likely to connect to peers that its neighbours are already connected to than an Overnet peer. The relatively high clustering coefficient at the start of the simulation in the Overnet network seems to imply that Overnet peers first connect locally to find out information about the network, and later discard these connections in favour of longer distance connections.

The large simulations took 20 minutes for the Gnutella network, and 22 minutes for the Overnet network. This is about a factor 4 higher than the simulation was specified to last, which is caused by the high rate of events during the simulation. Parsing took about 2.5 hours for the Gnutella network, and 55 minutes for the Overnet network. The difference in parsing time is caused by a higher number of connections between Gnutella peers. Although the active number of connections at any point in time is about the same between the two networks, the Gnutella network had a total number of unique connections of about 15 times higher than the Overnet network had. Gnutella connections thus seem to be much more short lived.

5.3 Special Scenario: Search Traffic

The aim of this simulation is to show that it is possible, and useful, to be able to specify the exact scenario executed by SYMPTOP. In order to analyse the network traffic caused by a search in the system, we increase the number of peers in the system, and observe the effect this has on network traffic when a constant number of queries is sent through the network.

5.3.1 Simulation Parameters

Unlike the previous simulations, the scenario for this simulation was not generated using the SG. Instead, the scenario is really simple, and was entered manually: Every 200 seconds, starting at the start of the simulation, 100 peers are added to the simulation. At 100 seconds after each addition of peers to the simulation, a total of 100 peers

issue a search query in the system. This simulation lasts for 1000 seconds, thus giving us 5 insertion events of 100 peers and 5 search events consisting of 100 searches.

5.3.2 Simulation Results

Figure 5.1c through 5.5c show the results for this simulation. Figure 5.2c reveals some interesting information about the two protocols simulated.

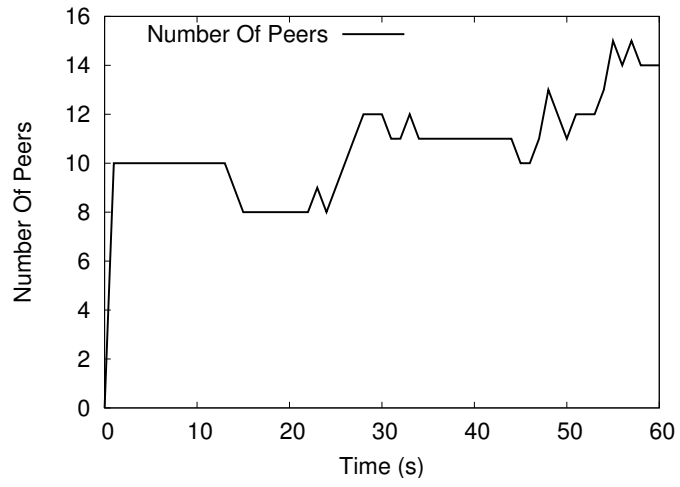
The Gnutella network retains an almost constant average number of connections per peer as more peers are added. In the Overnet network the average number of connections per peer increases fast with the first few additions of new peers to the simulation, but with later additions of peers this growth is less. It should be said that the number of connections in this simulation is not accurate for the Overnet network, because of its heavy use of the UDP protocol. Since the peers never leave the system, once a UDP packet has been sent between two peers, the connections is considered to be there until the end of the simulation. This property of our metrics generation will have to be taken into account.

The average pathlength shown in Figure 5.3c does not show us spectacular results. The much lower pathlength in the Overnet system is not surprising when taking the number of connections in the system into account. The clustering coefficient of both networks, shown in Figure 5.4c, converge as the simulation progresses. This means that the ratio of long-range and short-range connections in both systems is about equal.

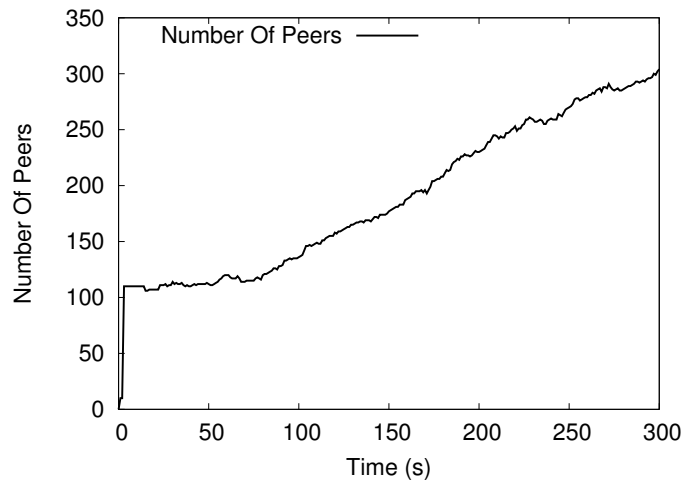
In Figure 5.5c network traffic of both networks is shown, although that of Overnet is hardly visible. What is immediately visible however is when the connect and search events happen. In order to get a better idea of the network traffic of both P2P systems, we have generated another visualisation of the network traffic, showing the cumulative traffic up to a certain point in time, given in Figure 5.6. What can be observed is that the total network traffic after a search event rises faster as more peers are connected in the Gnutella network, while very little network traffic is used when searching in the Overnet network.

These simulations took 17 minutes and 18 seconds, and 18 minutes and 52 seconds for the Gnutella and Overnet network, respectively. The small difference between the time the simulations took and the time they were specified to last, 16 minutes and 40 seconds, can be explained by the fact that this simulation scenario did not issue as many commands per second as the previous simulations did.

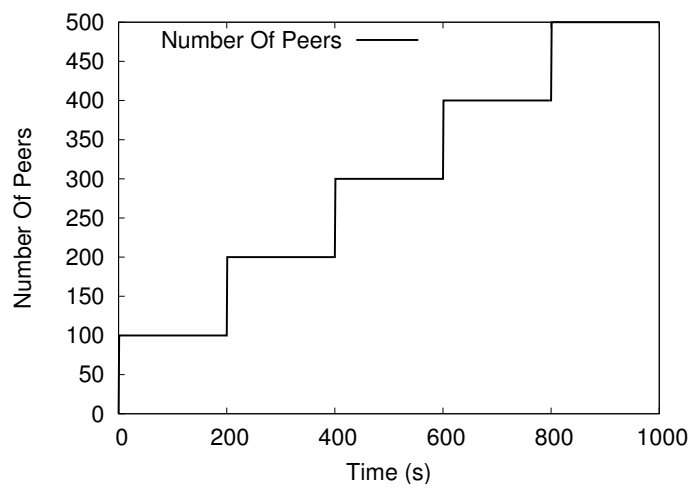
Parsing and metrics generation took more than 14 hours for the Gnutella data, while the same processing took 3 hours and 50 minutes for the Overnet data. These measurements are not accurate, as the parser was ran in the idle time of an actively used desktop workstation, but no accuracy error can account for this long execution time. The performance of the Result Processing Tool will have to be improved in order to make the parsing of data logged by simulations that last longer, and contain more peers, viable.



(a)

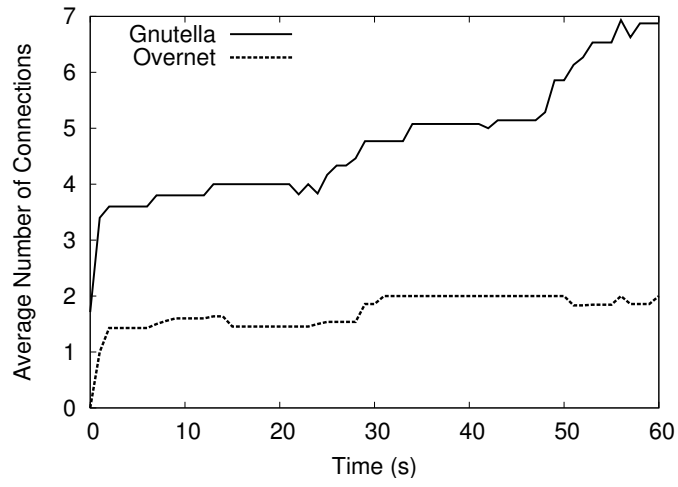


(b)

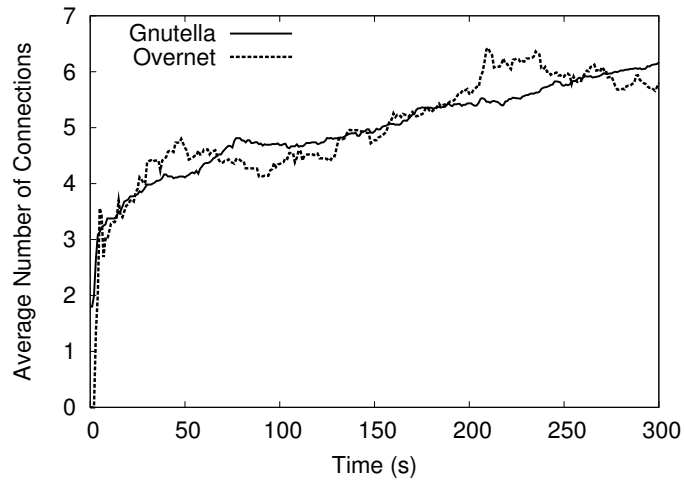


(c)

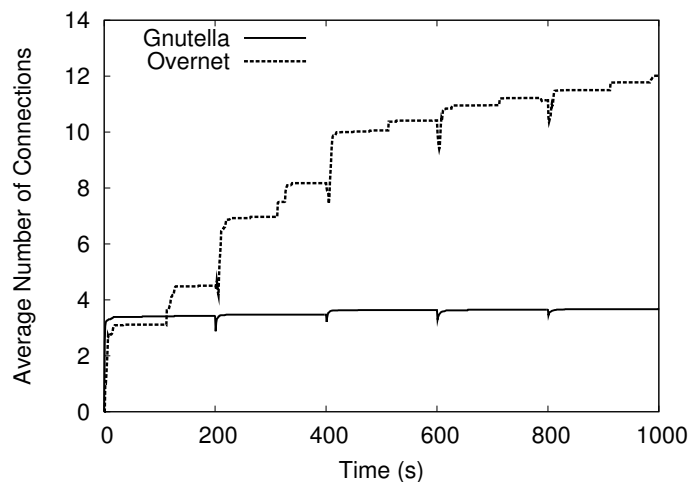
Figure 5.1: The number of peers in the a) small, b) large, and c) search traffic simulation.



(a)

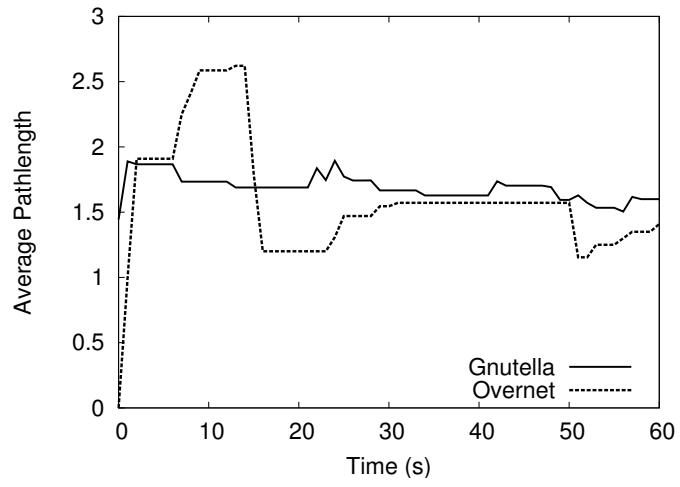


(b)

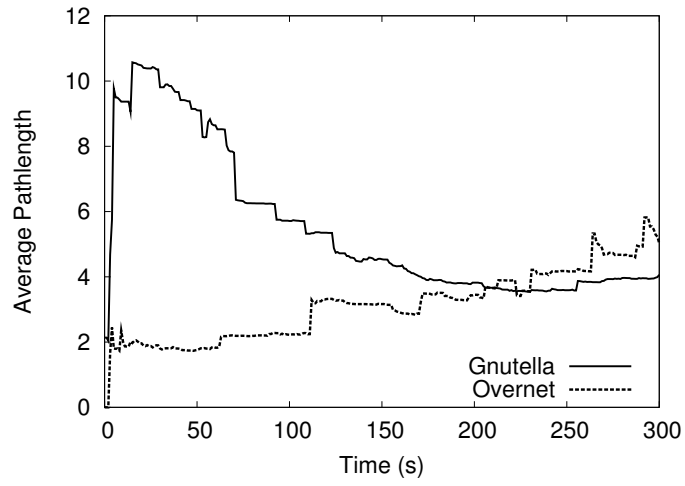


(c)

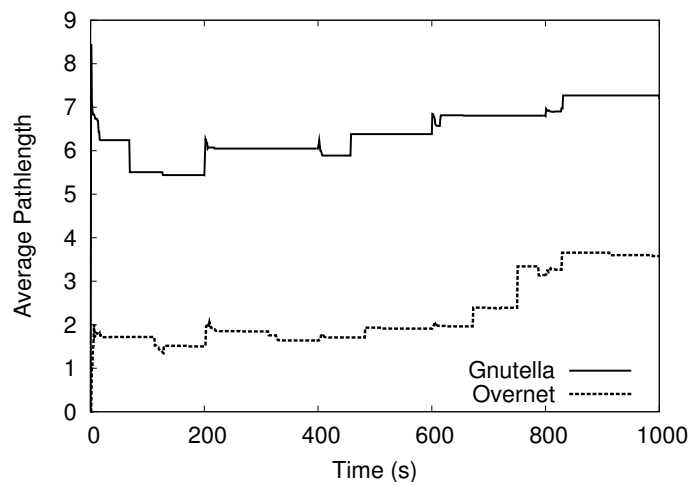
Figure 5.2: The average number of connections in the a) small, b) large, and c) search traffic simulation.



(a)

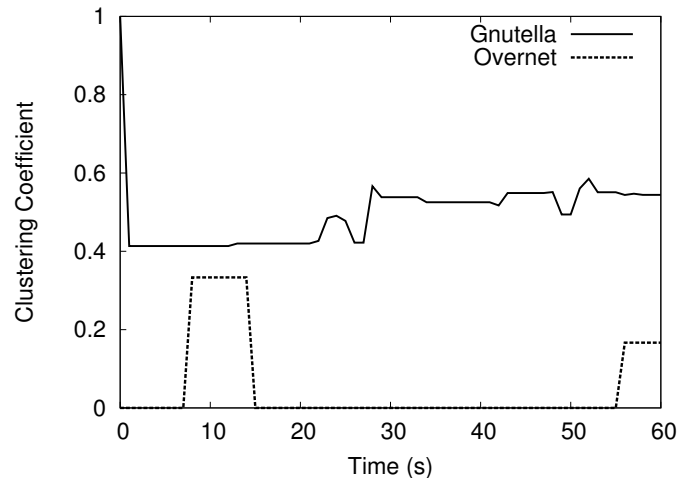


(b)

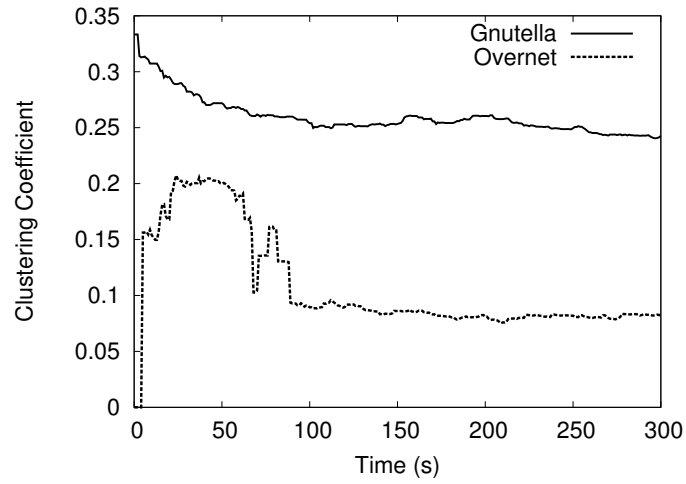


(c)

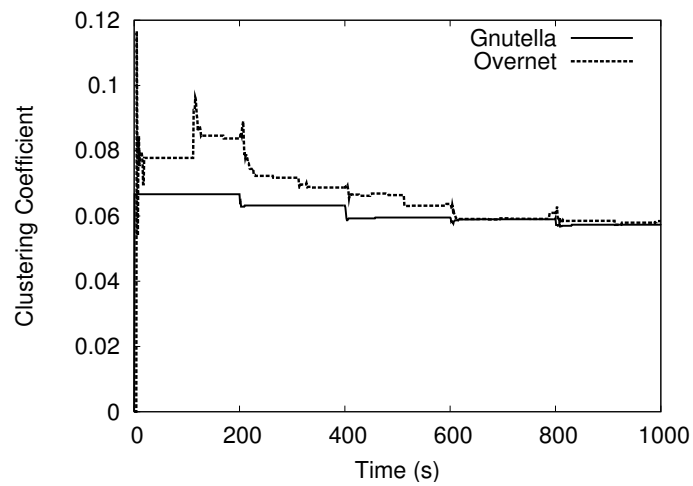
Figure 5.3: The average pathlength in the a) small, b) large, and c) search traffic simulation.



(a)

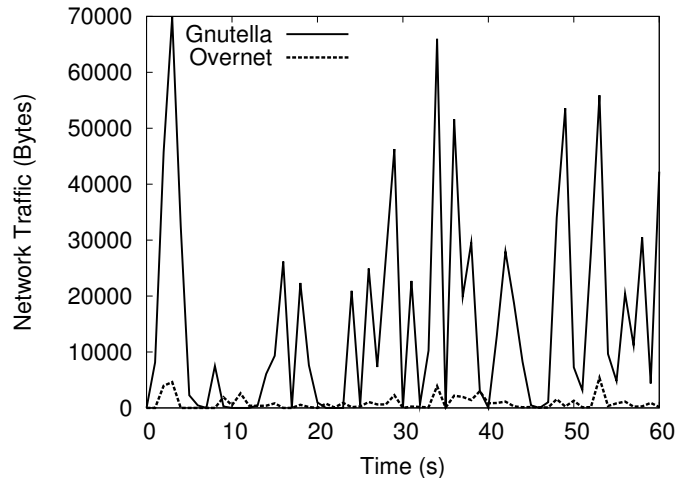


(b)

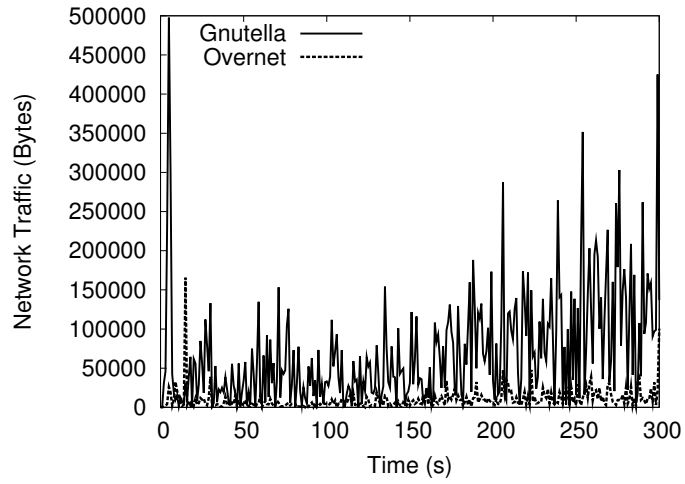


(c)

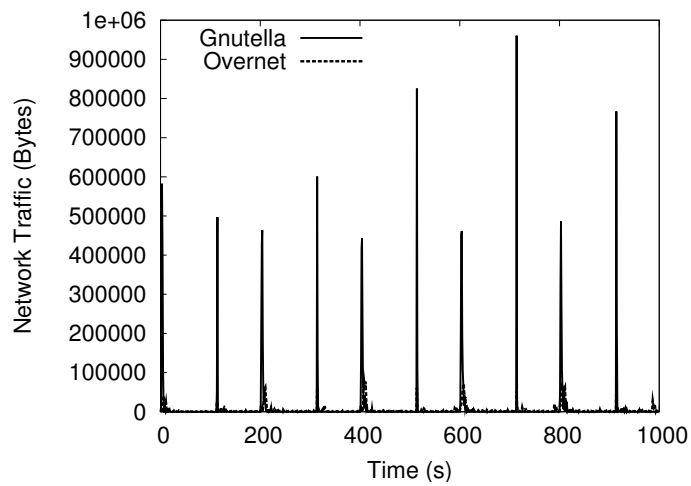
Figure 5.4: The Clustering coefficient in the a) small, b) large, and c) search traffic simulation.



(a)



(b)



(c)

Figure 5.5: The amount of network traffic in the a) small, b) large, and c) search traffic simulation.

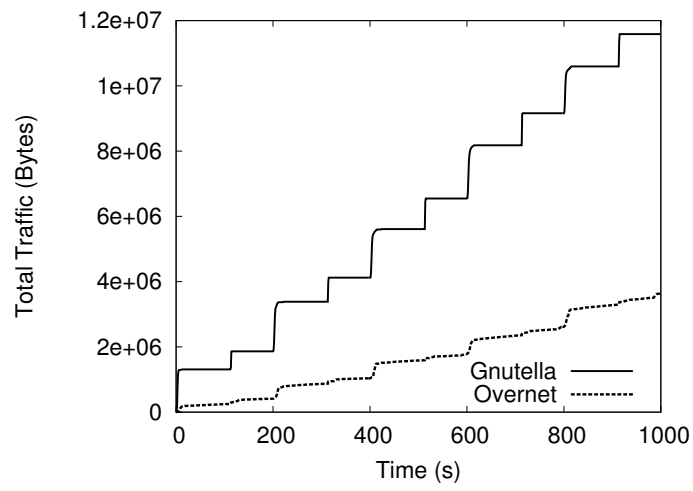


Figure 5.6: The cumulative network traffic during the search traffic simulation.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this thesis we have presented the design and implementation of the SYMPTOP simulation system for P2P networks. SYMPTOP allows us to simulate a large number of peers at the same time, using existing P2P clients to do the simulations instead of having to implement the P2P protocol into the simulation system ourselves. We have shown that performance metrics can be extracted from these simulations with existing clients by analysing network traffic that was logged during the simulations. We have presented results from our initial simulations of the Gnutella and Overnet network, which have shown that the more recent Overnet system makes more efficient use of network resources to do the same tasks.

The SYMPTOP simulations can yield very different performance measurements per simulation, even when the same scenario and client is used. The reason for this is that while the commands we give to peers in two simulations are the same, we have no control over the internal processes of the P2P clients. If a random element is used in routing or network construction by the simulated client, this can result in completely different networks every time a simulation with the same scenario is ran. Other factors like a different network delay per simulation can also be the cause of different network topologies. One way this could happen is that in one simulation a peer might receive a connection attempt from one peer first, and in the next simulation it receives a connection attempt from another peer first, causing it to drop the attempt from the first peer which arrived late due to network delay. In order to get more accurate idea of the performance of a simulated network, multiple simulations with both the same and different scenarios would have to be run.

The quality of the results of a simulation also depends on the network protocol a P2P system uses. While traffic is constantly measured in SYMPTOP, information on existing connections is only sampled at a fixed interval. This means that very short TCP/IP connections might not be logged. Use of the UDP network protocol also decreases the accuracy of some of the results, as we are unable to determine whether a connection between two peers using the UDP protocol still exists if no data is sent. Simulations of P2P systems that use TCP connections that last a longer period of time will therefore yield more accurate performance metrics than when a network with many short-lived connections, or with UDP traffic.

The metrics generated by SYMPTOP give a detailed view of the network properties of a P2P system. Special scenarios allow us to measure the message complexity of certain events in the P2P network. How the performance measurements obtained by running a SYMPTOP simulation relate to the performance of a P2P network deployed on the Internet depends heavily on the quality of the simulated scenarios. In order to get the best results, we advice to use data collected in studies of live P2P systems such as given in [20, 17] to generate accurate simulation parameters and scenarios.

6.2 Future Work

There are four aspects in which we would like to extend and improve the SYMPTOP simulation system. First, in order to test the download performance of P2P protocols, we would need to integrate downloading in our simulation system. Integrating downloads into our system would however require us to limit the bandwidth available to peers. Otherwise all peers would be able to use the fast interconnection network between simulation nodes, or even worse, transfers between peers running on the same simulation node would not even have network delay between them.

A second addition to SYMPTOP would be to make peers have more heterogeneous properties. Currently, heterogeneity is only based on connection time, and/or search and publish frequency. In real P2P networks, peers may differ in many more aspects, such as bandwidth, storage size, and processing power. A good portion of peers in real P2P networks is also incapable of accepting incoming connections, as these are either blocked by a firewall or a NAT setup where a peer connects to the Internet through another computer or router that does not forward incoming connections to the peer.

Third, a big problem SYMPTOP faces right now is that of data processing time. The results from simulations that can be performed within half an hour can take several hours to be processed by the Result Processing Tool. Most of this time is not spent in parsing the logged data, but rather in the metrics generation, which contains all-to-all pathlength calculation, a high-complexity algorithm that becomes very expensive to run as the network grows to larger sizes. Two possible solutions to this problem might be to convert the Result Processing Tool from Python to another, faster, language, and/or parallelising the Result Processing Tool, as these calculations can be done in an embarrassingly parallel way.

The last suggested improvement would be to include more detailed network traffic analysis in the Result Processing Tool. Analysing the contents of network packets could allow the SYMPTOP system to track messages being routed through the P2P network. This information could then be used to analyse the routing protocol that is used in the simulated networks, and to determine exactly which portion of network traffic is caused by which actions.

Bibliography

- [1] Distributed ASCI Supercomputer. <http://www.cs.vu.nl/das2/>, 2002.
- [2] A. Oram (ed). *Peer-to-Peer, Harnessing the Benefits of a Disruptive Technology*. O'Reilly, 2001.
- [3] R. Bhagwan et al. Understanding Availability. In *Peer-to-Peer Systems II: Second International Workshop*, volume 2735 of *LNCS*, pages 256–267. Springer-Verlag, 2003.
- [4] The Gnuplot Website. <http://www.gnuplot.info/>, 2004.
- [5] M. Harren, J.H. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In A. Rowstron P. Druschel, F. Kaashoek, editor, *Peer-to-Peer Systems: First International Workshop*, volume 2429 of *LNCS*, pages 242–250. Springer-Verlag, 2002.
- [6] Java Programming Language. <http://java.sun.com/>, 2005.
- [7] S.D. Kamvar, M.T. Schlosser, and H. Garcia-Molina. Incentives for Combatting Freeriding on P2P Networks. In H. Kosh, L. Böszörményi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference Klagenfurt, Austria*, volume 2790 of *LNCS*, pages 1273–1279. Springer-Verlag, 2004.
- [8] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the Dynamic Evolution of Peer-to-Peer Networks. In A. Rowstron P. Druschel, F. Kaashoek, editor, *Peer-to-Peer Systems: First International Workshop*, volume 2429 of *LNCS*, pages 22–33. Springer-Verlag, 2002.
- [9] The Lsof Project. <http://freshmeat.net/projects/lsof>, 2004.
- [10] Q. Lv, S. Ratnasamy, and S. Shenker. Can Heterogeneity Make Gnutella Scalable? In *Peer-to-Peer Systems: First International Workshop*, volume 2429 of *LNCS*, pages 94–103. Springer-Verlag, 2002.
- [11] D. Karger M. Kaashoek. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Peer-to-Peer Systems II: Second International Workshop*, volume 2735 of *LNCS*, pages 98–107. Springer-Verlag, 2003.
- [12] U. Wieder M. Naor. A Simple Fault Tolerant Distributed Hash Table. In *Peer-to-Peer Systems II: Second International Workshop*, volume 2735 of *LNCS*, pages 88–97. Springer-Verlag, 2003.

- [13] P. Maymounkov and D. Mazières. Kademia: A Peer-to-Peer Information System Based on the XOR Metric. In A. Rowstron P. Druschel, F. Kaashoek, editor, *Peer-to-Peer Systems: First International Workshop*, volume 2429 of *LNCS*, pages 53–65. Springer-Verslag, 2002.
- [14] A.T. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured Superpeers: Leveraging Heterogeneity to Provide Constant-Time Lookup. In *Internet Applications: Third IEEE Workshop (WIAPP 2003)*, pages 104–111. IEEE, 2003.
- [15] T. Ngan. Enforcing Fair Sharing of Peer-to-Peer Resources. In *Peer-to-Peer Systems II: Second International Workshop*, volume 2735 of *LNCS*, pages 149–159. Springer-Verslag, 2003.
- [16] J.O. Patterson. A Matter of Trust: Reputation Management in Peer-to-Peer Networks. 2005.
- [17] J.A. Pouwelse, P. Garbacki, D.H.J. Epema, and H.J. Sips. A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System. Technical report, Delft University of Technology, 2004. 4th International Workshop on Peer-to-Peer Systems (IPTPS'05).
- [18] Python Programming Language. <http://www.python.org/>, 2005.
- [19] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.
- [20] S. Saroiu, P.K. Gummadi, and S.D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical report, University of Washington, 2001. UW-CSE-01-06-02.
- [21] Tcpdump Website. <http://www.tcpdump.org/>, 2004.
- [22] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *Data Engineering: 19th International Conference (2003)*, pages 49–60. IEEE, 2003.
- [23] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-area Location and Routing. Technical report, University of California, Berkely, 2001.
- [24] BitTorrent Website. <http://bittorrent.com/>, 2004.
- [25] CHARVA: A Java Windowing Toolkit for Text Terminals. <http://www.pitman.co.za/projects/charva/>, 2004.
- [26] eDonkey Website. <http://www.edonkey2000.com/>, 2004.
- [27] EliteTorrents Website. <http://www.elitetorrents.org>, 2005.
- [28] Gnutella Website. <http://www.gnutella.com/>, 2004.
- [29] KaZaa Website. <http://www.kazaa.com/>, 2004.
- [30] Overnet Website. <http://www.overnet.com/>, 2004.
- [31] TorrentBytes Website. <http://www.torrentbytes.net>, 2005.

Appendix A

Simulation Scenario File Format

A Simulation Scenario File (SSF) consists of any number of lines. Each line contains one command for the Coordinator, and is composed as follows:

<timestamp> [peerID] <command> [argument]

1. **timestamp:** The time in seconds after the simulation started at which the command should be executed
2. **peerID:** If the command is intended for a single peer, this is its ID (optional)
3. **command:** One of the commands specified in Table A.1 or A.2
4. **argument:** Any required argument to the command (optional)

The possible commands are shown in Tables A.1 and A.2. Table A.1 deals with commands that affect a single predetermined peer. The commands specified in Table A.2 affect multiple random peers at the same time, or are global system commands that affect the simulation in general.

| Command | Argument | Description |
|----------------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------|
| create | none | Starts the P2P client |
| connect | none | Gives the peer the command to connect to the system |
| leave | none | Cause the peer to disconnect from the system (in the process possibly killing the client, if disconnect is not supported cleanly) |
| restart | none | Causes the peer to reconnect to the network, should be used after a leave command was used. |
| search | search string | The peer will search the network for the terms given in the search string |
| publish | filename | Publishes a file with the specified filename in the system |

Table A.1: Peer Commands in Simulation Scenario Files

| Command | Argument | Description |
|----------------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| startn | number of peers | Starts the specified number of peers |
| connectstarted | none | Connects the peers created in the most recent start statement |
| leaven | number of peers | Forces a number of peers to leave the system. These peers are chosen randomly, out of the pool of peers that are not individually addressed in the simulation. The peer client might be killed if the client does not support a clean disconnect |
| restartn | number of peers | The specified number of peers are restarted |
| quit | none | Stops the simulation |
| drawnetwork | number | If the client the peers run support output of all the peers a single peer is connected to, this command will create a picture of the layout of the network, and some additional statistical information about the system in its current state |

Table A.2: Global Commands in Simulation Scenario Files

Appendix B

SYMPTOP Installation and Usage

This appendix will discuss the installation and usage of the SYMPTOP system. SYMPTOP consists of three parts, all provided in the SYMPTOP distribution package: *symptop.tgz*, of which the installation and usage will be discussed: The Simulation Generator, the main simulation system, and the Result Processing Tool.

B.1 Installation

SYMPTOP has been designed with a Linux cluster environment in mind, which means that some of its features may not work on other systems. It is assumed that the user has basic knowledge of the Java programming language and of running Java programs.

After extraction of *symptop.tgz* the following directories will be created, containing all the software:

| Directory name | Contents |
|----------------|-------------------------------------|
| <i>mysrc</i> | SYMPTOP Java Packages |
| <i>bin</i> | shell scripts used by the simulator |
| <i>tcpp</i> | the Result Processing Tool |

B.1.1 The *mysrc* Directory

The *mysrc* directory contains all the Java packages required to run the Simulator Generator, and the main simulation system. The *mysrc* should be included in the user's classpath. If it is not already, go to the directory in which the *mysrc* directory is located, and give the following command:

```
export CLASSPATH=${CLASSPATH}:'pwd'/mysrc/
```

This should be done every time you want to use the Simulator Generator or the main simulation system.

The *mysrc* directory contains three Java Packages: *simgen*, *symptop*, and *util*. The contents and configuration of these packages will be discussed below.

The *simgen* Package

The *simgen* package contains the simulator generator. It comes pre-compiled with the sources included. No additional configuration is required.

The *symptop* Package

The *symptop* package contains the main simulation package, as well as the Gnutella and Overnet clients we used in our initial simulations. It is supplied pre-compiled with the sources included.

The main configuration settings are supplied in the *simconfig* file, located in the main package directory. It contains the following settings:

| Setting name | Description |
|-------------------|--------------------------------------------------|
| <i>password</i> | the user's password |
| <i>rmiaddress</i> | the IP:port combination of the Java RMI Registry |

The *password* is required for executing the TCPDump program as root using the *sudo* command. More details on the *sudo* command will be given in Section B.2.

The *rmiaddress* line should contain the IP and port number of the Java RMI Registry. The default setting provided in the *simconfig* file is the fs3 file-server of the DAS-2 cluster, with port number 8666. When running simulations on the DAS-2 cluster located in Delft, there is no need to change this.

The *symptop* packages assumes it is running on a Linux system. If this is not the case, some references to the */tmp/* directory located in the following files will need changing: *SimNode.java*, *GnutellaPeer.java*, and *OvernetPeer.java*. If *symptop* is running on a Linux system, modification of these files is not required.

The *util* Package

The *util* package contains some utilities used by the *simgen* and *symptop* packages. The utilities consist of a random number generator, implementations of various distributions, and some file utilities. No modification of these files is necessary, the package is supplied pre-compiled with sources included.

The Result Processing Tool is a collection of Python scripts, which can be placed anywhere. It can be instantly used, without configuration or additional action required from the user.

Some additional scripts are supplied which are used by the simulation package to transfer files from the SNs after a simulation is finished, these are extracted to the *bin* directory.

B.1.2 The *bin* Directory

The *bin* directory contains shell scripts that are used by the main simulation system when running on the DAS-2 cluster. If the simulations will not be running on the DAS-2 system, this directory can be safely deleted.

The *nodelinux* file in this directory contains a single-line list of DAS-2 node numbers. This list should include all the computation node numbers of the DAS-2 cluster

that can be used during a simulation. The only time this file should be edited when running simulations on the DAS-2 cluster in Delft, is when a node is down, in which case it should then be removed from this file, because otherwise some scripts might have long execution delays.

The *bin* directory is assumed to reside in the user's home directory. If this is not the case some scripts in the scripts supplied with the main simulation program should be edited to reflect the location that the *bin* directory is in.

B.1.3 The *tcpp* Directory

The *tcpp* directory contains the Result Processing Tool. This tool is written in the Python programming language, so no compilation is required. There are also no configuration options for this software.

B.2 Required *sudo* Usage Rights

The *sudo* tool provided with Linux allows a regular user of a system to execute specified commands as though that user had super-user permissions on the system. The SYMPTOP requires that the user has *sudo* rights to the following executables:

/usr/sbin/tcpdump
/bin/kltcpdump
/bin/mvlog

The *kltcpdump* and *mvlog* scripts are located in the *bin* directory supplied with SYMPTOP, and are only required when running SYMPTOP on the DAS-2.

B.3 Usage

In this section the use of SYMPTOP is explained, from Simulation Scenario File generation to results parsing.

When a P2P Client ID is referred to in the text, the following values should be supplied for the two supplied clients:

| P2P Client name | P2P Client ID |
|-----------------|---------------|
| Gnutella | 0 |
| Overnet | 1 |

B.3.1 Simulation Scenario File Generation

Simulation Scenario Files can either be generated by the Simulation Generator, or specified manually by the user. This section will deal with the generation of a scenario using the Simulation Generator. Manual specification can be done with the help of Appendix A.

The Simulation Generator is started from any directory with the following command:

java simgen.Simgen

This will show an interface consisting of a *File* and an *Edit* menu, and some general information about the simulation parameters.

The *edit* menu allows the user to enter a *Simulation Duration* in seconds, a *Random Seed* used when generating the simulation. The user is also able to select the *Peer Types* option, which will show a list of peer types, which will be empty initially. From here the user can add, edit, and delete peer types.

When adding or editing a peer type, the following parameters can be set per peer type:

Initial number
Join rate
Leave rate
Restart rate
Search rate
Publish rate
Search Hit Fraction

The *initial number* defines the number of peers of this type that are connected to the network before the simulation starts. The *Join rate* defines at what rate new peers connect to the system. The *Leave rate* defines how long a single peer stays connected to the system. The *Restart rate* defines how long peers that disconnected from the network stay disconnected from the network. The *Search rate* and *Publish rate* the intervals between consecutive search and publish events of one peer, respectively. The *Search Hit Fraction* is a number between 0 and 1 that determines what fraction of search queries issued by peers contain a keyword that matches the filename of a previously published file.

The various rate have two input fields: An average in seconds, and a distribution. The possible distributions are: *Exponential*, *Gamma*, *Normal*, *Poisson*, *Weibull*, and *Flat Rate*. The specified average determines the average of the specified distribution, or when *Flat Rate* was chosen, the constant value used.

After all peer types have been specified, the user can save the simulation parameters to a file for later use using the *Save* or *Save As* options from the *File* menu. In order to generate a Simulation Scenario File the *Export* option in the *File* menu should be used, which prompts the user for a filename, after which the Simulation Scenario File is written to that file.

B.3.2 Starting the Simulation

Starting the simulation on the DAS-2 system differs from starting the system on other systems, as for the DAS-2 a script was written to do most of the work. First starting simulations on the DAS-2 will be explained, after which starting simulations on other systems is explained.

Starting the Simulation on the DAS-2

In order to start the simulation on the DAS-2, the user should go to the *mysrc/symptop/* in which the main simulation system sources and compiled classes are. The simulation is then started with the following command:

```
simulate <number of SNs> <P2P Client ID> <Simulation Scenario File>
```

The *number of SNs* argument specifies how many DAS-2 processors should be used for this simulation. The *P2P Client ID* should be one of the previously mentioned values. The *Simulation Scenario File* argument should be the filename of a previously generated Simulation Scenario File.

After giving this command the simulation will be automatically queued on the DAS-2 system and started when the specified number of processors are available.

B.3.3 Starting the Simulation on Other Systems

If the simulation is not run on the DAS-2, the following steps should be taken.

Starting the RMI Registry

The Java RMI registry should be started on the machine specified in the *simconfig* file, with the specified port. The Java RMI Registry can be started with the following command:

```
rmiregistry [portnumber]
```

If no port number is specified, the default value of 1099 is used.

Starting the Coordinator

The Coordinator is started with the following command:

```
java symptop.Coordinator <number of SNs> <SSF>
```

Both arguments are mandatory, specifying the number of SNs that will be connecting to the system, and the simulation scenario that will be used. After the specified number of SNs have connected to the Coordinator, the simulation will start automatically.

Starting the Simulation Nodes

The SNs are started with the following command:

```
java symptop <SN identification number> <P2P Client ID>
```

The SN identification number should be 0 for the first node, 1 for the second, and so on. The P2P Client ID should be one of the values mentioned before.

After the Simulation

If the default scripts are not used, the user will have to copy the network traffic logs himself to the machine which will take care of result processing. The files that needs to be copied are the following:

```
/tmp/nodeX.log  
mysrc/symptop/output/tcpd/nodeX.netlog
```

Both files will need to be moved to the *mysrc/symptop/output/tcpd/* directory of the machine the Result Processing Tool will be running on.

B.3.4 Result Processing

Simulation output is parsed by giving the following command:

```
tcp.py <P2P Client ID> <simulation output directory>
```

Both arguments are mandatory. P2P Client ID should be one of the values mentioned before. The *simulation output directory* specified is by default *mysrc/symptop/output/*, but the data in this directory can be moved anywhere before processing. The command can be run from any directory, but the output files from the parsing will be written in the directory the command was given from.

Parsing for other clients which the user implements into SYMPTOP himself require the user to edit the Result Processing software, though limited statistics (number of peers, network traffic) can be obtained without modification by using the Gnutella parsing rules.

The output of the Result Processing tool consists a number of files with the *.dat* extension. The *dateinfo.dat* file contains a line for every second the simulation lasted, with the following data per second: the time in simulated seconds since the start of the simulation, the number of peers in the system, the amount of traffic, the number of search events, the number of publish events, the average pathlength, the clustering coefficient, and the average number of connections. For every peer a separate *.dat* file is generated, containing the following data for every second: the time in simulated seconds since the start of the simulation, the number of bytes sent by the peer, the number of bytes received by the peer.

