HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Olli Railio

EFFECT OF SERVICE TIME DISTRIBUTION ON BITTORRENT-TYPE PEER-TO-PEER FILE TRANSFER

Thesis submitted for examination for the degree of Master of Science in Technology

Espoo September 29, 2009

Thesis supervisor:

Prof. (pro tem) Samuli Aalto

Thesis instructor:

Prof. (pro tem) Samuli Aalto

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Olli Railio

Työn nimi: Palveluaikajakauman vaikutus BitTorrent-tyyppisessä peer-to-peer tiedonsiirrossa

Päivämäärä: September 29, 2009 Kieli: Englanti Sivumäärä: 9+98

Tiedekunta: Elektroniikan, tietoliikenteen ja automaation tiedekunta

Professuuri: Tietoverkkotekniikka Koodi: S-38

Valvoja: Prof. (ma) Samuli Aalto

Ohjaaja: Prof. (ma) Samuli Aalto

Vertaisverkot ovat viime aikoina kasvaneet suureksi osaksi Internetin kokonaiskuvaa. Ne perustuvat pyrkimykseen korvata perinteinen palvelin-asiakas malli, jossa asiakkaat saavat haluamansa datan siihen keskittyneeltä palvelimelta, ja tekevät joka käyttäjästä sekä tiedon vastaanottajan että lähettäjän. Vertaisverkkojen suosion kasvaessa ja niiden käytön laajentuessa uusiin tarkoituksiin, niiden tutkimisesta ja ymmärtämisestä tulee entistäkin tärkeämpää.

Työssä tutkitaan pienimuotoisen tapahtumavetoisen simulaattorin avulla yksinkertaistetun tiedoston siirtoon käytetyn monipalaisen vertaisverkon verkon käyttäytymistä eri parametreja vaihdellessa. Keskeisimpänä tarkoituksena on tutkia verkon käyttäjien palveluaikajakauman vaikutusta koko verkon tehokkuuteen, mitä mitataan lähinnä simuloitujen tiedostojen siirtoaikojen keskiarvojen kautta. Muita työssä varioituja parametreja ovat käyttäjien verkkoonsaapumisnopeus, lähteiden poistumisnopeus, jaettavan tiedoston palojen määrä sekä palojen valintapolitiikka, eli se, millä perusteella kukin lataaja valitsee seuraavan ladattavan palan ja käyttäjän, jolta se ladataan.

Tuloksena työssä havaitaan, että käytetty jakauma voi vaikuttaa paljonkin tiedostojen simuloituihin keskimääräisiin siirtonopeuksiin. Tulokset ovat monessa tapauksessa ennakoitavissa, mutta jotkut niistä voivat rippua myös itse käytetyn simulaattorin rakenteesta. Tärkeimpänä tuloksena ehkä onkin, että yksittäisen simulaattorin tuloksiin ei tule sokeasti luottaa, vaan perustellunkin mallin tuottamat tulokset olisi hyvä varmistaa myös käytännössä.

Avainsanat: Vertaisverkko, tapahtumavetoinen simulaatio, palveluaikajakauma

Author: Olli Railio

Title: Effect of service time distribution on BitTorrent-type peer-to-peer file transfer

Date: September 29, 2009 Language: English Number of pages: 9+98

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Networking technology Code: S-38

Supervisor: Prof. (pro tem) Samuli Aalto

Instructor: Prof. (pro tem) Samuli Aalto

Peer-to-peer networks have recently become a significant part of the Internet's landscape. They are based on the attempts to move away from the traditional client-server model of the Internet by making each user both a sender and receiver of data. As peer-to-peer networks grow in number and expand to new uses, their research and understanding becomes increasingly important.

The primary goal of this thesis is to study the effect of the peers' service time distribution on the efficiency of the whole network, which is mostly measured through the mean times that the peers spend downloading a file. This is accomplished by creating a simplified model of a peer-to-peer network, building a simple event-driven simulator and studying how it behaves when certain parameters in the simulations are varied. These parameters include the arrival rate of peers, the departure rate of seeds, the number of chunks in a file and the peer selection policy, ie. the method through which downloaders select the next piece to download, and the peer from which they will download it.

In the course of the work it is discovered that the service time distribution can have a significant effect on the mean transfer times of a network. While the results are usually predictable, some results could be surprising and dependent on the type of the simulator used. The most important insight might be that one should not blindly trust a single simulator, and even a well thought-out model could need practical validation.

Keywords: Peer-to-peer network, event-driven simulation, service time distribution

Preface

I would like to thank my family, friends, and everyone else who has offered encouragement and support for the completion of not only this thesis, but also the rest of my studies. There have been good times and bad times, and always having someone around to share them both has been invaluable. Thank you to William Martin for helping me finalize the language of the thesis. I would also like to express my deepest gratitudes to the thesis instructor and supervisor, Samuli Aalto, for his time and expert guidance. Every time we talked, you had good ideas on how to move forward with this thesis, and without those, it probably wouldn't be half as good. Thank you.

Otaniemi, September 29, 2009

Olli Railio

Contents

Terminology

BitTorrent	A peer-to-peer technology for transferring files in chunks
Chunk	A piece of a file being distributed in a peer-to-peer network
Churn rate	The rate of peers entering and exiting the network
Downloader	A peer who has not yet completed the download of a single chunk, and is only downloading
Leecher	A peer who has completed the download of at least one chunk of a file and is both uploading and downloading; there is no particular distinction between leechers and downloaders in this work
P2P	Peer-to-peer
Peer	A participant in a peer-to-peer network
Seed	A peer who has a complete file in a peer-to-peer system and is only uploading data
Swarm	A group of peers sharing a file or files described in a single .torrent-file
Service time	A time calculated for each peer upon their entrance to the system, the fastest time it can take for that peer to complete a transfer
Download time	The time during a transfer when a downloading peer is receiving data from another peer, calculated as the maximum of the service times of a downloader and an uploader
Transfer time	The time elapsed between the arrival of a peer into the system and the completion of that peer's download
Sojourn time	The full time that a peer spends within a peer-to-peer system

Symbols

Symbols used in defining the model described in this document.

λ	The arrival rate of peers
γ	The departure rate of seeds
α	A rate parameter used for many of the probability distributions
μ	The upload bandwidth of a peer
c	The download bandwidth of a peer
n	The number of chunks in a file
t	The elapsed time in the simulation
t_{max}	The maximum simulation time, after which the simulation is ended
$x(t)$	The number of peers downloading the file at time t
$y(t)$	The number of seeds at time t
X	The service time of a peer
Y	The download time of a peer, ie. the maximum of the service times of an uploader and a downloader
T	The total transfer time of a peer

Common symbols used for denoting additional variables in related research.

θ	Abort rate of peers, ie. the rate at which leechers leave the system without completing their download
η	The effectiveness of file sharing, which determines how efficiently a leecher uploads a file in the network as opposed to a seed who has already completed the entire download. A value of 0 means that leechers are not uploading at all, while a value of 1 means that the leechers upload just as much as seeds

1 Introduction

The recent popularization of peer-to-peer (P2P) technology has caused a noticeable paradigm shift in the transfer of files over the Internet. Starting with Napster in 1999, and continuing with eg. Kazaa, Gnutella, eDonkey, and BitTorrent, P2P provides an alternative to the traditional client-server model of the Internet. More recently, peer-to-peer technology has begun to spread into other applications, such as streaming video (eg. SopCast, TVUPlayer) and IP telephony (mainly through Skype). While most analysts agree that a significant portion of total Internet traffic is currently taken by P2P protocols, the exact amount has proven difficult to measure or even estimate. The percentages given have ranged anywhere from 25% and 90% in the past few years.[2][1]

The actual technologies used for P2P file transfer can differ in many ways. While older technologies—such as Napster or Kazaa—relied to some extent on centralized servers to keep track of peers and files, BitTorrent allows users to set up trackers and indexes for similar purposes. With certain technologies, peers can share data about other peers inside the network. The common element in all peer-to-peer networks is the ability for peers to share the data they have, utilizing their upload bandwidth—which had previously been mostly idle during file transfers—to a much larger degree.

While the client-server model defines the roles of clients and servers clearly, the distinction is more fluid in peer-to-peer systems. In BitTorrent, any new peer entering the network will first act as a downloader until the completion of the first chunk, or a piece of the file being downloaded. Thereafter, the peer will keep downloading until it has the entire file, but will also offer its upload capacity to the network by sharing the chunks it already has. A peer that has a portion of a file is sometimes called a leecher. After the download is completed, the peer can still stay on the network as a seed.

This thesis will concentrate on analyzing the performance of a BitTorrent-like network, and an architecture that does not rely on a centralized server. The analysis is done using an event-driven simulator of a peer-to-peer network, where the service rates are drawn from different probability distributions and the effect of varying several parameters such as number of chunks, arrival rate of peers and departure rate of seeds on the performance of the system is studied. The thesis specifically builds on the work done by Susitaival and Aalto in [26] and the special assigment by Bhusal in [7]. The former raised the question about the effects of service time distributions on the performance of peer-to-peer systems while the latter concentrated on analyzing the effects of the service time distribution on the performance of a single-chunk system. This thesis will extend some of the latter results to multi-chunk models.

This document is structured as follows. Section 2 will briefly outline the history and development of peer-to-peer systems, and proceed to summarize the design considerations of the BitTorrent-protocol in particular. Section 3 focuses on some previous efforts at analyzing BitTorrent-like networks and briefly discusses their

relationship with this thesis. Section 4 gives a detailed outline of the methods used in analyzing the problem, including both the simulator and the model it is based on. Section 5 describes the results obtained through those methods and will provide some analysis of these results. Section 6 gives some conclusions that one can draw by using the research methods defined here, and suggests some ways one can build on this research.

2 Peer-to-peer systems

This section will first define what peer-to-peer systems are and where they have traditionally been used. It will then describe the evolution of peer-to-peer technologies and discuss what has led to their recent popularity. Finally, it will concentrate on the BitTorrent protocol in particular, and attempt to give some insight into its defining characteristics and general functionality.

2.1 Historical perspective on peer-to-peer systems

Peer-to-peer systems have existed in some form since the early days of the Internet. The Internet's newsgroup system, Usenet, is built around a decentralized model where news servers containing the posts can be added or removed from the network without any data being lost from the system as a whole. The domain name system (DNS) also makes use of a decentralized approach, where the hierarchical structure is distributed among name servers across the Internet, with each lower-level server being responsible for only a small subset of hosts.[21] However, the client-server model eventually became the standard for most Internet applications, until the popularization of a new generation of peer-to-peer systems at the turn of the millennium.

2.2 Evolution of peer-to-peer technologies

The recent rise to prominence of peer-to-peer protocols started with the introduction of Napster in mid-1999. Napster was limited to sharing music files and relied on a central server for peer and file discovery.[4] This eventually led to its demise as the central server was shut down because of claims of copyright violations, and the new, pay version of Napster has not reached its previous popularity.

After the initial success of Napster, several other peer-to-peer protocols sprung into existence. Some of these were—in terms of their structure—imitations that relied on a similar architecture with a centralized server, but could expand the system to for example more file types. Others, such as Freenet and the first versions of Gnutella, introduced a so-called decentralized or pure p2p architecture, where each peer only needed to know the location of one other peer, and the addresses of other peers and locations of available files within the network would then be shared among the peers.[4][12] While the decentralized approach was closer to a true peer-to-peer network, in which the lines between clients and servers would blur completely, the actual implementations proved inefficient in some cases. There was simply too much overhead since requests would have to be sent throughout the entire network to get comprehensive results on file availability.[12]

To battle the issues introduced by a fully decentralized structure, a hybrid approach between the centralized and decentralized network structures was implemented for example by later versions of Gnutella, from version 0.6. onwards. In a hybrid peer-

to-peer network some nodes are assigned as so-called superpeers or supernodes, which take on some responsibilities of a centralized server, acting as indexes for a small subset of regular nodes, or leafnodes.[12] Still, any single supernode is not crucial for the functioning of the network as a whole: upon the departure of one such supernode, each remaining leafnode will simply be assigned to another. The assignment as a supernode is not generally set in stone; a leafnode can be promoted if it has the requisite capabilities and it is deemed necessary by the network. Similarly, a supernode can be 'demoted', if its performance has degraded.

Figure 1 provides an illustration of these general network structures that have been in common use in peer-to-peer networks.

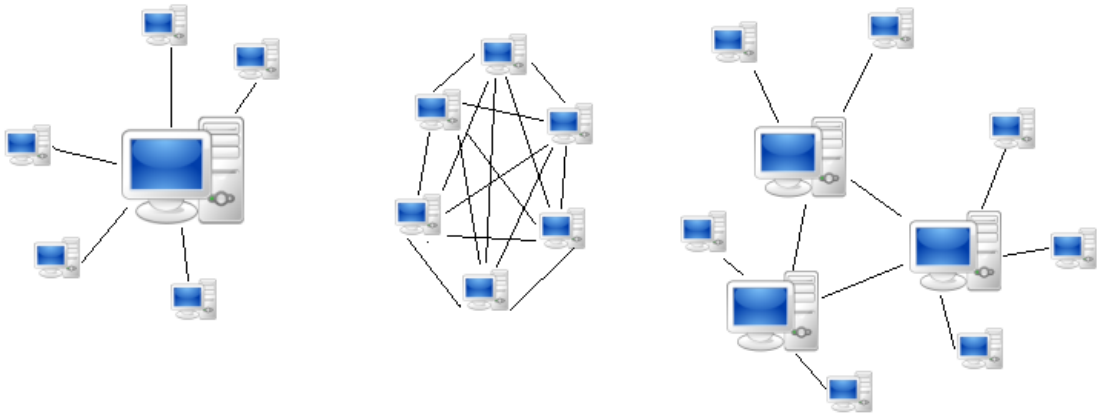

Figure 1: *Different structure types of peer to peer networks. From left to right: centralized server structure, decentralized structure and a hybrid structure with three supernodes. Note that the pictured structures are only applicable for eg. file discovery, the actual transfer of data will happen directly between the peers in each case.*

2.3 BitTorrent technology

The BitTorrent protocol was designed by Bram Cohen, who also introduced the first client and tracker implementations in July 2001. The protocol uses a scaled-down version of a centralized server architecture, meaning that there is no official central server. Rather, it is possible for any user to set up a server of their own. The server in question, called a tracker, is only responsible for keeping track of the peers on the files that are linked with that particular tracker. The tracker software itself is included with some versions of the official BitTorrent-client.

In addition to the trackers, some centralized method of file discovery is also needed. This is usually accomplished through indexes that provide a list of available files. Some BitTorrent client applications also use distributed hash tables (DHTs) for keeping track of files that the peers within the network are sharing and making those files available to others. The fundamentals of the BitTorrent protocol are defined in several papers, the following basics are given in [10] written by Cohen.

A BitTorrent swarm—meaning a group of peers sharing a single file or a group of files using the BitTorrent protocol—is started when an initial seed builds a .torrent-file based on the files that the seed wishes to share. This .torrent-file is linked to one or more trackers, which can also be set up for the single purpose of sharing this grouping of files. A peer wishing to download the files contained within the torrent will then download the .torrent-file, open it with a BitTorrent-client and connect to the tracker, which gives the peer information on the other peers already in the swarm. The sharing of the file happens when a peer sends requests for chunks of the file to other peers in the swarm who have already completed downloading that particular chunk.

Most implementations of the BitTorrent-protocol generally use a rarest first (RF) policy in selecting a piece to download. When using this policy, a peer will try to request chunks of the shared file in order of their rarity within the system. This is done in order to facilitate a more even distribution of the pieces: with rarest first, peers will be more likely to have pieces that other peers will be interested in downloading in the future. This selection policy also helps increase the lifetime of a torrent swarm by making it more likely that at least one instance of each chunk is contained within the swarm if or when a seed leaves. An exception to the rarest first policy is done when a peer first starts his download, as the first piece to be downloaded is selected randomly. This is done in order to make it more likely that a peer gets at least one complete chunk of the file as quickly as possible, so that they will have something to share with the rest of the swarm.

BitTorrent also utilizes a so-called choking algorithm—implemented in different ways in different clients—which decides to which other peer another peer will upload. This is done in order to give preference to peers who share well: a client will generally be more likely to unchoke a peer that it is downloading from at a higher rate. The end result is that peers who share a larger amount of their own upload bandwidth, will in turn get faster download rates. According to Cohen in [10], this gives the protocol a built-in incentive mechanism, which in turn lead to the robustness of the protocol. The choking algorithm also leads to an increased efficiency of the entire system, since the result of the higher-bandwidth peers continually trying to upload to other high-bandwidth peers is an eventual clustering of peers with similar bandwidths.[17]

A thorough explanation of the choking algorithm used in the official BitTorrent client version 4.0.2 is given by Legout et al. in [17]. The algorithm lists each peer that has uploaded to the current peer within the previous ten second timeframe. Those peers are then ranked in order of their upload rate to the current peer, and the current peer will then decide to upload to the peers that have transmitted data the fastest. By default, a peer will simultaneously unchoke four other peers, but this amount can be changed in most BitTorrent clients. Additionally, a so-called optimistic unchoke mechanism is run every thirty seconds, where an additional peer is chosen to be unchoked at random from among all peers that are interested in the current peer's data. This allows for new peers—who at first will not have any pieces to share—to enter the swarm without the precondition of having uploaded data first. Furthermore, the optimistic unchoke algorithm helps break up existing

clusters of peers, in order for the peers to find better sharing partners.

Some BitTorrent client applications have introduced a so-called super-seeding mode to distribute at least one copy of a shared file more rapidly. When this mode is used, the initial seed will try to prioritize its upload capacity to peers that share data well in the network, by initially sending out only one copy of each chunk, and looking at which chunks have been well-shared within the network when deciding which peers to upload to next.

3 Related research

During the years that peer-to-peer networks have been in the forefront of file sharing technologies, a significant amount of research has gone into understanding and analyzing their behavior. The different networks have been analyzed through mathematical models, simulations, emulations and measurement of actual data gathered from P2P networks. Since the amount of source material is so extensive, this section will by necessity only concentrate on a small subset of this research. It will first examine different types of models used in analyzing BitTorrent-like networks before looking at the results gleaned from the analysis. Finally, a brief look is given at some sources offering concrete improvements to the BitTorrent protocol and its client software.

3.1 Models used for analyzing BitTorrent-like networks

The performance of BitTorrent-like networks has been evaluated in several scientific papers. Many of these have used theoretical models to simulate the network. There are certain parameters that are common among many such models. These include the arrival rate of peers (λ), the departure rate of seeds after their download has completed (γ) and the upload and download rates of peers (μ and c, respectively). The current time in the system is given as t, and the number of peers who are currently downloading is given as $x(t)$ and the number of seeds in the system is $y(t)$. Some papers also use a parameter signifying the rate of peers aborting their download before its completion (θ) and the so-called effectiveness of file sharing (η). The latter is a parameter with values between 0 and 1 that determines how efficiently a leecher uploads data compared to a seed with similar bandwidth limitations (with 0 meaning a leecher will not upload anything and 1 meaning it uploads as efficiently as a seeder).

3.1.1 Deterministic fluid model

Qiu and Srikant present a widely emulated, simple deterministic fluid model for analyzing a BitTorrent-like network in [25]. Since the model is deterministic, all analysis is done on sample averages. Most of the following description is simply paraphrasing Qiu's and Srikant's paper.

The model presented does not distinguish between separate chunks of a file. It is simply assumed that each leecher will be contributing a set amount to the total upload bandwidth, along with each of the seeders. The upload and download bandwidths of the peers are kept constant. The total departure rate of downloaders (which includes downloaders becoming seeds and abortions of downloads) is given as

$$\min\{cx(t), \mu(\eta x(t) + y(t))\} + \theta x(t), \tag{1}$$

which, along with the arrival rate of downloaders λ and the departure rate of seeds γ, represents all of the transitions happening inside the modeled system.

The deterministic fluid model for the evolution of the number of downloaders and seeds within the system can then be given as

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \lambda - \theta x(t) - \min\{cx(t), \mu(\eta x(t) + y(t))\} \tag{2}$$

and

$$\frac{\mathrm{d}y(t)}{\mathrm{d}t} = \min\{cx(t), \mu(\eta x(t) + y(t))\} - \gamma y(t). \tag{3}$$

In a steady-state, the amount of leechers and seeds in the system will be constant, ie.

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = 0. \tag{4}$$

Combining Equations (2), (3) and (4), Qiu and Srikant obtain the steady state values for the downloaders and seeds:

$$\begin{aligned} 0 &= \lambda - \theta\bar{x} - \min\{c\bar{x}, \mu(\eta\bar{x} + \bar{y})\}, \\ 0 &= \min\{c\bar{x}, \mu(\eta\bar{x} + \bar{y})\} - \gamma y(t), \end{aligned} \tag{5}$$

where $\bar{x}$ and $\bar{y}$ represent the equilibrium values of the amounts of leechers and seeds, respectively.

When the upload rates of peers are assumed to always be higher than download rates, the downloading bandwidth of peers will be the constraint on the performance of the network, ie. $c\bar{x} \leq \mu(\eta\bar{x} + \bar{y})$. Solving Equation (5) results in

$$\begin{aligned} \bar{x} &= \frac{\lambda}{c(1 + \frac{\theta}{c})} \\ \bar{y} &= \frac{\lambda}{\gamma(1 + \frac{\theta}{c})}. \end{aligned} \tag{6}$$

On the other hand, when the uploading bandwidth of peers is assumed to be the constraint—ie. $c\bar{x} \geq \mu(\eta\bar{x} + \bar{y})$—Equation (5) results in

$$\begin{aligned} \bar{x} &= \frac{\lambda}{\nu(1 + \frac{\theta}{\nu})} \\ \bar{y} &= \frac{\lambda}{\gamma(1 + \frac{\theta}{\nu})} \end{aligned} \tag{7}$$

for the peer amounts in an equilibrium state. In the equation,

$$\frac{1}{\nu} = \frac{1}{\eta}(\frac{1}{\mu} - \frac{1}{\gamma}). \tag{8}$$

Combining Equations (6) and (7) results in

$$\begin{aligned} \bar{x} &= \frac{\lambda}{\beta(1+\frac{\theta}{\beta})} \\ \bar{y} &= \frac{\lambda}{\gamma(1+\frac{\theta}{\beta})}, \end{aligned} \tag{9}$$

where

$$\frac{1}{\beta} = \max\{\frac{1}{c}, \frac{1}{\eta}(\frac{1}{\mu} - \frac{1}{\gamma}\}. \tag{10}$$

The average time a peer spends in the system T can then be calculated using Equation (9) and Little's law in the form

$$\frac{\lambda - \theta\bar{x}}{\lambda}\bar{x} = (\lambda - \theta\bar{x})T. \tag{11}$$

T can then be seen to be

$$T = \frac{1}{\theta + \beta}. \tag{12}$$

Qiu and Srikant also present some interesting findings based on these formulas: the mean download time T is not related to the arrival rate of peers λ, and the system will always scale (regardless of the other parameters) as long as $\eta > 0$. When comparing the results given by their fluid model to actual data captured from a BitTorrent network, Qiu and Srikant found that it could simulate a network fairly well.

Many researchers have used a similar model as the one presented by Qiu and Srikant to represent a BitTorrent network. Guo et al. used a deterministic fluid model with a decreasing arrival rate in [14]. The purpose of their paper was to study the behavior of the BitTorrent protocol during the initial flash crowd phase—when many peers join the swarm soon after the torrent file is published—as well as later on, when interest in the torrent will wane. The actual function for the arrival rate was measured from real data.

Yue et al. also present a deterministic fluid model of a BitTorrent-like system in [28]. Instead of uniform peer types as in [25], the peers are distributed into groups according to parameters that simulate their connection types. It is argued that since there is only a limited number of connection types available to the global population

of Internet users, this should be enough to give a more representative look into a realistic network. However, it is still assumed that the peers in a network would use all of their bandwidth for the purpose of the single network that is being analyzed. In reality, it stands to reason that the peers would also use their Internet connection for sharing and leeching other files and other tasks such as browsing the web, therefore creating completely discrete sets of users might not be the best alternative.

While a deterministic fluid model is relatively simple to build, it has faced some criticism as a methodology. Since it only works with sample averages, such a model will not be useful in showing all of the variations in a network.[26] Since there is no stochasticity, it could only show how a network behaves in an average case, and will not be able to show how different types of peers would perceive the network. In order to meaningfully extend the results to a model with multiple chunks, the simple fluid model would have to be altered to incorporate multiple 'queues' that each represent one chunk of the system and a meaningful way to separate the rates of each queue. In the given form, it is not possible to examine file availability or other similar parameters in a BitTorrent network, since when using the formulas as presented, even a lone peer would always finish its download as long as the transfer abort rate was low enough.

3.1.2 Markov chain model

Another widely used model type utilized in describing a BitTorrent-like network is the Markov chain. De Veciana and Yang used a Markov chain model to represent a BitTorrent-like network in [11]. In his special assignment described in [7], Bhusal uses both a deterministic fluid model and a Markov chain to simulate a BitTorrent-like network where the files consist of a single chunk. In [26], Susitaival and Aalto use a Markov chain to represent a BitTorrent network for a file with two chunks.

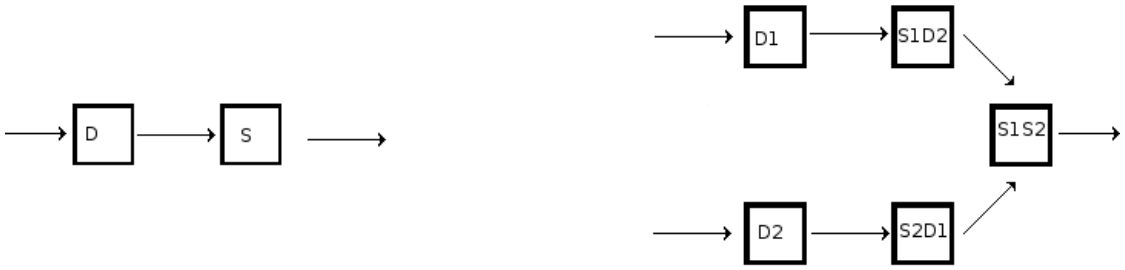

Figure 2: *The different states of a single peer in a single-chunk and two-chunk Markov chain model. D represents the download of a chunk while S represents seeding. In the single-chunk model, peers will always enter the system as downloaders, then become seeds before leaving the system. In the two-chunk model, peers can start downloading either of the two chunks, before becoming an uploader for the downloaded chunk and a downloader for the other chunk. They then become seeders who share both of the chunks before departing the system.*

In the Markov chain model, a single peer is modeled by the combination of chunks in a file that the peer has already downloaded, as shown in Figure 2. The state of the entire system is represented by the number of peers that have downloaded a particular combination of the chunks. Transitions can happen from each state to one representing the previous state with the addition of a single chunk for one downloader. The transition rates between these states are determined by the upload and download speeds of the peers in the system. Furthermore, downloaders will enter the system at a rate λ and each seeder will depart at a rate γ. De Veciana and Yang use the notation (x, y) in [11] to represent the state of the system, with x being the number of downloaders and y the number of seeds. The full set of transition rates is given as

$$\begin{aligned} q((x,y),(x+1,y)) &= \lambda \\ q((x,y),(x-1,y+1)) &= \mu(\eta x + y) \\ q((x,y),(x,y-1)) &= \gamma y, \end{aligned} \tag{13}$$

where the first equation represents the arrival of a new peer, the second the completion of a download and the last the departure of a seed. The transfer rates on the system are assumed to be limited only by the upload rates of the peers and the effectiveness of sharing η.

The Markov chain approach is limited to exponentially distributed arrival, departure and transition rates and does not scale well, since the number of states needed to model each peer increases rapidly when the number of chunks increases. A single-chunk model requires only two states to represent each peer, a two-chunk model uses five states for each peer, and the number of states grows to 13, 33 and 81 for three, four and five chunk models respectively. As files shared in a BitTorrent network can often be divided into thousands of chunks, a complete model of such a system would not be possible (for 1000 chunks, the number of states needed to represent each peer would be around $5 * 10^{303}$). However, it is argued in [26] that many of the relevant characteristics of a BitTorrent-like peer-to-peer system can be analyzed even when the maximum number of chunks is limited to only two.

3.1.3 Event-based simulation

To experiment with systems with more chunks, Susitaival and Aalto also use an event-based simulation, where pairs of peers exchanging chunks are formed.[26] The simulator works by assigning each leecher to download from a seed or another leecher that has already downloaded chunks that the first leecher requires. A very similar simulator is used as the main method of analysis in this thesis; a more complete description of the simulator is given in Section 4.

A more complete survey of different existing peer-to-peer network simulators is given by Naicken at al. in [20].

3.1.4 Other methods

In [5], Barbera et al. concentrate on defining a complex Markov chain model of a single BitTorrent peer, taking into account the number of connections of different connection types that the peer has to other peers in the network.

3.2 Evaluation of the BitTorrent protocol and its variants

In the course of their work in [25], Qiu and Srikant discovered that the effectiveness of file sharing in a BitTorrent network (η) is very close to 1 (a more complete definition of the parameter was given in 3.1). This signifies that the upload bandwidth of peers who are still downloading the file is utilized very efficiently. Legout et al. demonstrate the same property experimentally in [17], showing that the built-in sharing incentives of BitTorrent—namely the choking algorithm—works in maximizing the effectiveness of peers' upload bandwidth. In the same paper, Legout at al. discuss the clustering of peers that have similar bandwidth, also enabled by the choking algorithm.

Legout et al. argue in [16] that the rarest first chunk selection policy and the choking algorithm are enough to make BitTorrent a robust and efficient protocol for file sharing.

Pouwelse et al. studied some issues of the BitTorrent networks in [23]. Their research indicates a high level of integrity in BitTorrent networks, with few fakes and little noise in the networks. They also commented on the transitional nature of the protocol—when the last seeder leaves a swarm and the remaining leechers do not have a complete copy of the file among them, the torrent will die, reducing availability of the content. According to Pouwelse, giving incentives to seed after a peer has finished the download would aid in this problem. Some private trackers already have a built-in incentive mechanism by requiring relatively high upload/download ratios from their members.

The superseeding or initial seeding mode used by some BitTorrent clients is analyzed by Chen et al. in [9]. While using the superseeding mode, the initial seeder in the torrent swarm tries to maximize the effectiveness of its upload bandwidth by seeding a full copy of the file to useful peers as quickly as possible. This is done by keeping track of the circulation of the chunks in the swarm and prioritizing the upload to those peers that have shared the chunks previously uploaded to them. By measuring real data transfers in an experimental setting, Chen et al. discovered that the superseeding mode reduced the upload bandwidth need of the initial seeder by around 20% before the completion of the first download.

3.3 Improving the BitTorrent protocol and clients

While the general consensus seems to be that the BitTorrent protocol works reasonably well, some improvements are still being suggested. In practical terms, the most

substantial effort has gone towards developing 'better' clients within the confines of the BitTorrent protocol. These include BitTyrant, developed by the University of Washington and the University of Massachusetts Amherst and outlined by Piatek et al. in [22], and Tribler, whose initial development was made in the Delft University of Technology and the Vrije Universiteit. Tribler is described by Pouwelse at al. in [24]. BitTyrant is sometimes called a selfish client, since it is only concerned in getting a file as fast as possible instead of the well-being of the entire network. However, [22] also argues that if every peer in the network would use the client, even the collective performance would improve.

Another type of BitTorrent-client is introduced by Locher et al. in [18]. This so-called proof-of-concept client—BitThief—is built so that it never actually uploads data within a network, but still manages to download files at a rate that is usually competitive among other clients by flooding the network with requests far more often. The selfish behavior is rewarded by fast download rates even in what they call sharing communities, or private BitTorrent trackers. This reveals an intrinsic vulnerability in most BitTorrent networks, in that while the actual protocol might be robust, it still relies on some honesty in its implementations to be effective. Combating the exploit is possible, since BitTorrent trackers can be configured to filter out client softwares that are known to be selfish or malicious.

In [27], Tian et al. suggest a different unchoking algorithm to be used with the BitTorrent protocol, which would help increase the stability of a BitTorrent network and thus the availability of files at the cost of longer download times for some peers. Their suggestion is to alter the unchoking algorithm to account for not only current upload rate, but consideration for future availability. This would be done by uploading faster to peers that have downloaded less of the file. However, the availability of files and the number of seeds usually decreases as the interest in the file wanes, so while keeping a torrent alive a little longer after the last seeder has left might help a bit, it probably would not have much of an effect in total download amounts. As with many other models, they also do not consider the return rate of peers who have already completed the download, which might have a larger effect on the network as a whole.

4 Research methods

This section will first define the type of mathematical model on which the work is based, and then proceed to discuss the actual simulation model used and the simulator built on the basis of that model. The peer selection policies and probability distributions that can be used by the simulator are discussed in detail in the latter parts of this section. A more detailed look is given to the exponential distribution, as an example of how the service times will behave in a multi-chunk model.

4.1 Terminology

The terms and symbols defined in this section are adhered to throughout this thesis. Most of these will match with the ones described in Section 3.1. λ denotes the arrival rate of peers while γ denotes the departure rate of seeds. n is the number of chunks in a file and t is the time elapsed during a simulation.

A distinction is made between the terms sojourn time, transfer time, download time and service time. In this work, sojourn time is used to denote the full time that a peer spends within a peer-to-peer system, beginning from that peer's entry into the system and ending with its departure. Each individual peer will have a service time X, which will indicate how fast that peer could at best conclude the transfer of a chunk. The actual download time—ie. the time it takes to download a chunk—will be denoted by Y and determined by the maximum of the uploader's and downloader's service times. Transfer time will be denoted by T, meaning the time elapsed between the arrival of a peer into the system and the completion of that peer's entire download, including the time spent queuing for available uploaders. The parameter α will be used to define the rate parameter given to each probability distribution, where applicable.

4.2 Reasons for model selection

The primary purpose of this thesis is to examine the effects of service time distributions on the performance of a peer-to-peer system where the files being shared are divided into multiple chunks. The other main parameters to be altered are the chunk selection policies and the number of chunks used to model the file.

The analysis of the peer-to-peer network is done through an event based simulation, whose basic principle is very similar to the one used by Susitaival and Aalto in [26]. As is seen from the work done by Bhusal in [7] and Susitaival and Aalto in [26], a comprehensive mathematical analysis of a peer-to-peer network through a Markov chain model is possible for single-chunk and two-chunk models. However, the number of states in the system grows exponentially when the file is split into more chunks and a mathematical model becomes too complex to be a feasible tool for analysis. Similarly, a fluid model would not be useful in analyzing the effects of changing the service time distributions of peers.

No readily-available simulator was found that could easily be used to change the desired parameters (mainly service time distribution and chunk selection policies). Consequently a simplified model of the BitTorrent-like network was developed and a simulator was created based on that model.

4.3 Simplified model of a BitTorrent network

In the simplified model of a BitTorrent-like network used for this work, each peer can simultaneously download only one chunk of the file, which can consist of one or more chunks. After the peer has downloaded at least one chunk, it can begin uploading to a maximum of one other peer at any given time. If a peer cannot find another peer to upload to, its upload capacity will remain idle until at least one download is completed or a new peer enters the system. If a peer enters the system and cannot find another peer to download from, its download capacity will remain idle until the download of at least one chunk is completed within the system. Flow charts showing the download and upload behavior of peers in the system are included in Figures 3 and 4.

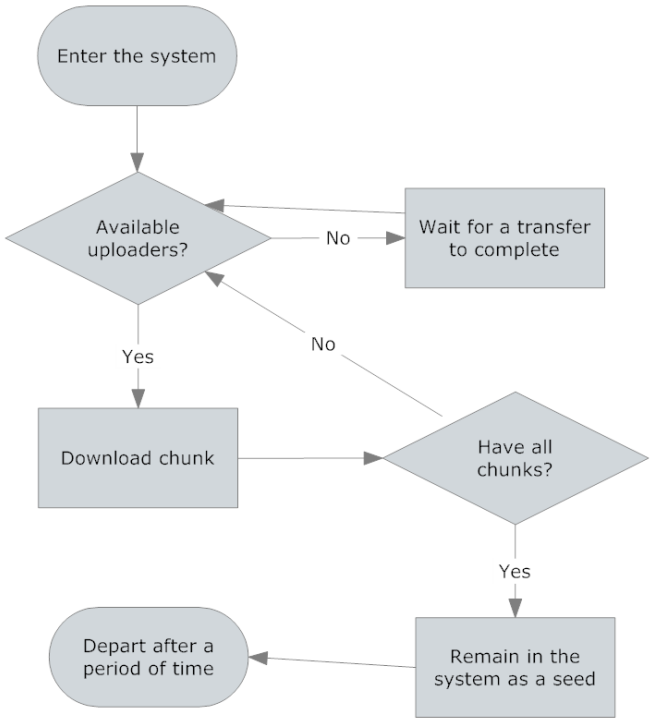

Figure 3: *A flow chart describing the download behavior of a peer. A new peer entering the system will try to find a peer it can download a chunk from. If none can be found, the peer remains idle until another transfer is completed. Upon the completion of a transfer, the peer will try to find another peer to download from, or—if it has collected all of the chunks of the file—remain in the system as a seed. The upload behavior of the peers remains separate from this chart, and is shown in Figure 4.*

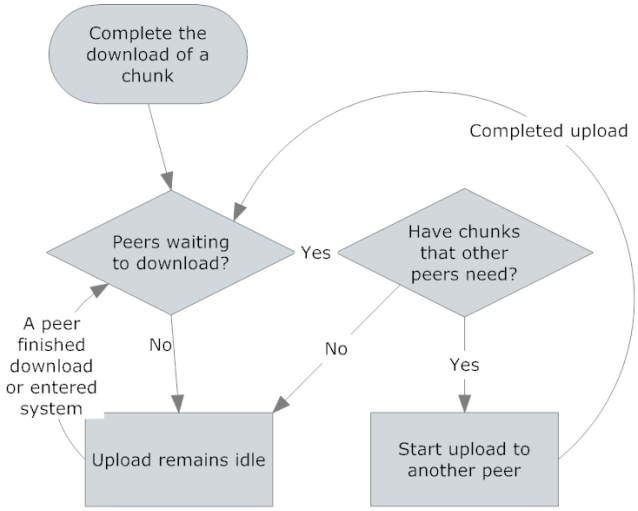

Figure 4: *A flow chart describing the upload behavior of a peer. Any peer that has completed a download will attempt to find another peer it can upload to. If no downloader is found, the peer remains idle. After the completion of an upload, the cycle is repeated.*

The limit on the simultaneous uploads and downloads is used to simplify the modeling and to enable quicker and longer simulations. A more complex simulation model would require significant processing power to correctly calculate the transfer rates of the peers, and all of the transfer rates would need to be recalculated every time a peer was assigned to download from another. Since the actual transfer rates would in reality fluctuate beyond those parameters due to events unrelated to the transfer of the current file, completely realistic modeling would still not be possible. If more accurate calculations of real transfer rates are needed, they could be obtained through analysis of actual data gleaned from BitTorrent networks, as is done by eg. Izal et al. in [15].

The simulator uses a stochastic model, introducing random elements in the peer arrival and departure times, and the service times of each peer. Peers arrive into the network at a rate λ and depart at a rate γ after they have completed their download. Both the arrival and departure times are drawn from an exponential distribution.

The upload and download times of a peer are equal and remain constant throughout a simulation run; in this thesis the service time of peers will simply be referred to with X. Having equal rates is a design choice made to simplify the system and the behavior of the peers. In reality, the peers' download rates would very likely be faster than their upload rates. The service times are drawn from some probability distribution—which can be changed for each simulation run—upon the peer's arrival in the network. The duration of each transfer—the download time—is the maximum of the service time of the uploader and the service time of the downloader.

The service times of each peer are kept constant throughout their stay in the system because it is believed that this gives a better representation of a file-sharing net-

work; it would seem likely that the speed of a client's connection will not see huge fluctuations during the course of a single transfer. While the actual transfer rates that users experience do vary within a system, the major factor affecting these rates is the connections that the user has to other peers, which will vary over time. In this thesis, this is modeled on a smaller scale, since each peer can only download from one other peer at the same time.

When different parameters are altered and compared, the mean ideal time it takes to transfer each file will be kept constant. This means that, for example, when the number of chunks in a file is increased, the service time of each individual chunk is decreased so that the sum of the expected service times of all of the chunks will remain constant. Similarly, when different probability distributions are compared, the mean service time will be kept constant in order to enable comparisons between the characteristics of the distributions.

4.4 Description of the simulator

The simulation is done in Java and its full source code is included in Appendix B. Java is used because its object-oriented structure simplifies the modeling of different parts of the system, when compared with a more mathematically-oriented programming language. The simulator was only meant to be used as a tool for this particular thesis, so it will not be particularly well-suited for other similar tasks without some modifications. All of the parameters are varied within the source code, and the program does not understand command line arguments. Therefore simply compiling and running the simulator would only give very limited results. While writing this thesis, the simulator was run within an IDE while the relevant parameters were changed and slight tweaks in other parts of the source code were made between the runs.

In the simulator, each peer is an object and has an instance of a file object. The peers are generated by a factory-type object, and the transfer times are generated by a separate generator based on the probability distribution used for the simulation. The events occurring during the simulation are modeled as separate objects, which are stored in a simple priority queue based on their time of occurrence. The simulator utilizes the strategy design pattern for the peer selection policies, which means that all of the selection policies are modeled as separate objects with certain common methods inherited from an abstract selection policy, and the policy to be used during a simulation run can be loaded during runtime. A tracker object is used to keep track of the number of peers in the system at all times. The SimulatorFramework class is a helper class that is only used to simplify running the simulations. It can iterate over all of the used distribution classes and selection policies, while also varying the arrival rate of downloaders, the departure rate of seeds and the number of chunks.

At the start of a simulation, an initial seeder—who has all of the chunks of the file—is first generated and the arrival time of the first peer is calculated. Further arrivals are generated using a Poisson process and peers which have completed their

download will depart from the system after some time according to the departure rate determined for that simulation run. In some simulations, the initial peer will be set to never leave the system. However, the simulator will sometimes swap a departing peer with the peer that was previously set to not leave the system (the probability that this swap is done is $\frac{1}{y}$, where y is the number of seeds in the system at the time of this departure). This is done in order not to skew the results obtained from long simulations based on one peer's transfer rate.

The simulation runs by drawing the next event (in chronological order) from a queue and changing the simulation time to match that of the event. The event can be either an arrival or departure of a peer, or the completion of the download of a chunk. When a peer arrives, it is assigned to download an available chunk from any other peer; if no chunks are available, the peer becomes idle. A new event depicting the arrival of the next peer to the system is calculated at this point and added to the event queue. If a seed leaves the network while it is uploading, the transfer will halt and the peer that was downloading will look for another peer to complete the transfer. If none is found, that peer will remain idle until another transfer is completed. The portion of the chunk that the peer has already downloaded is not lost, and it will take less time to finish the transfer when another uploader is found. When the peers are determining which peers to upload to next, priority is given to peers that have one or more partially completed chunks.

When the download of a chunk is completed, the downloader will either start to download a new chunk or—if it has completed the download of all of the chunks—calculate its own departure time and add it to the event queue. The peer that completed the upload of a chunk will be assigned a new downloader from the idle peers at this point, if one exists. The peer that completed the download will also look for a new peer to upload to, if it was not previously uploading. The simulation ends after a certain amount of time has passed. Statistics of the number of seeds and leechers within the system are collected at exponentially distributed intervals.

The chunk selection policies—which are used when the leechers look for a peer they can download from—are varied in the simulation; these include the random peer, rarest first, most common first and random chunk policies, which are outlined below in Section 4.5.

4.5 Chunk selection policies

The chunk selection policies outlined in this section are used during the simulation and their performance is also analyzed in this thesis. While the rarest first policy can intuitively be felt to be superior to the other policies, the inclusion of the others can give some insight when altered in association with the service time distributions. The chunk selection schemes are modeled after the similar policies used by Susitaival and Aalto in [26] and Bhusal in [7], in order to give some comparable results.

4.5.1 Rarest first

The rarest first (RF) policy means that whenever a downloader or a leecher has to decide which chunk to download next, priority is given to the chunks that are least abundant within the swarm. In theory, this will lead to an even distribution of chunks among the system and will help in increasing the lifetime of a torrent. As is outlined in Section 2.3, the BitTorrent protocol uses rarest first as its primary chunk selection policy. In this thesis, when the rarest first policy is used, it is followed for every chunk that the peers download. Even the first chunk a peer downloads is selected through this method, while with the actual BitTorrent protocol, it would be selected at random.

4.5.2 Most common first

As its name indicates, the most common first (MCF) selection policy is the opposite of the rarest first policy. Whichever chunk is the most numerous among the peers in the swarm will get priority when a downloader is choosing which of the available chunks to download next.

4.5.3 Random chunk

When starting a download, the random chunk policy means that each peer will first select a chunk to download at random from the set of all currently available chunks within the system, before selecting the peer to download from. When starting an upload, a peer will select the chunk to be uploaded at random from among all the chunks it has that are needed by other peers. This policy should in theory lead to a more even chunk distribution than the MCF policy, while not being quite as robust as the RF policy.

4.5.4 Random peer

When the random peer (RP) policy is used, each peer will select the next peer it will upload to or download from at random from the set of all available peers which either needs chunks from the current peer or has chunks that the current peer does not yet have. The chunk to be downloaded is then selected at random from among all of the desired chunks. Similarly to the random chunk policy, the performance of this policy is expected to fall somewhere between the rarest first and most common first policies.

In the simulator, a downloader will normally prioritize the download of chunks that it has already partially acquired. When the random peer policy is used, the uploading peer is first selected from the complete set of all available uploaders, before selecting a partial chunk if one is available. This was done to maximize the randomness of the selection policy, but giving a priority to completing each partial transfer might also have been a defensible design decision.

4.6 Service time distributions

This section outlines the probability distributions from which the service times can be drawn when using the simulator. It also describes how the variables are generated when using those distributions, and discusses the actual download times seen by each peer in the system. The exponential, Erlang-k and hyper-exponential distribution are used in a similar way as by Bhusal in [7]. The latter two distributions are mostly used to examine how well the results match with the earlier work and the effect of the variance of a distribution on the download and transfer times. The defining features of these two distribution classes (their relation to the exponential distribution) are not particularly relevant to the type of simulator being used.

Note that the parameter α is used to represent the rate parameter in most of the distributions, instead of λ as is used in most literature. This is done to keep the parameters constant within this thesis, as λ will represent the arrival rate of peers within the simulated system. Similarly, Bhusal used μ as the transfer rate parameter in his paper, but since the service times seen in the system will not exactly follow any simple probability distribution, the parameter α was selected for simplicity while μ will be used to denote the transfer rates seen in the system.

4.6.1 Service time distributions within the simulator

In most simulation runs the simulator is set to choose the download time of a chunk by using the maximum of the service times of the uploader and the downloader between which the transfer is taking place, ie.

$$Y = \max(X_d, X_u), \tag{14}$$

where X_d is the service time determined for the downloader and X_u is the service time of the uploader. When the actual download times are not calculated in such a way, it is always explicitly mentioned when discussing the results.

For certain reasons, the total transfer times will not follow the formulas given here exactly. During the simulation, if a seed that is uploading a file leaves, the transfer of the file halts and the downloader will look for another peer it can download the chunk from. An uploader will be more likely to leave during any single transfer if its own service rate is low because the service time will be longer, meaning it is more likely that the peer's departure event happens within that time frame. For that reason the real transfer time will also be dependent of the departure rate of the peers (which leads to an interesting situation, since peers leaving the system at a higher rate could actually decrease the mean transfer times). These formulas also do not account for waiting times in the system, which would be difficult to determine exactly when the amount of chunks in the system is high.

When two variables are drawn from a distribution, the maximum of those two variables does not follow the same distribution. In the simulation, each peer only has one service time which describes both the upload and the download capability of

that peer. These times are always drawn from the same distribution, so the actual download time of a chunk between two peers will be the maximum of two identically distributed and independent random variables. The cumulative distribution function of the maximum of two such random variables would simply be

$$F(y) = P[(X_1 \leq y) \cap (X_2 \leq y)] = F(x)^2, \tag{15}$$

where X_i is a random variable depicting the service time of a peer and $F(x)$ is the cumulative distribution function of those service times.[3] In practice, this means there will be a smaller proportion of low-end values and a larger proportion of high-end values of the underlying distribution. This result describes the distribution of the duration of any service time in the system when each file consists of only one chunk. The probability density function in the same case would be

$$f(y) = 2F(x)f(x). \tag{16}$$

When a file consists of multiple chunks, the total transfer time of a file will not simply be an independent sum of these variables. This is because the service time of each peer is kept constant throughout that peer's lifetime, while only the upload rate of the peer sending the data varies. As a result, the service times of the chunks a peer downloads will not be independent of each other.

Let $X_i, 0 \leq i \leq n$ be the service time of each peer, and $Y_i, 1 \leq i \leq n$ be the maximum of the service times of the first peer and each other peer ie.

$$Y_i = \max(X_0, X_i). \tag{17}$$

$Z = \sum_{i=1}^{n} Y_i$ is then the sum of those maximums. The mean of Z will simply be the sum of the means of Y_i, ie.

$$\bar{Z} = \sum_{i=1}^{n} \bar{Y_i}. \tag{18}$$

The variables Y_i are not independent, and therefore the variance of Z will be affected not only by the variances of each Y_i, but also their covariances. Since $\mathrm{Cov}(Y_i, Y_i) = \mathrm{Var}(Y_i)$,

$$\mathrm{Var}(Z) = \sum_{i=1}^{n} \sum_{j=1}^{n} \mathrm{Cov}(Y_i, Y_j). \tag{19}$$

This can be written in terms of the expected values as

$$\mathrm{Var}(Z) = \sum_{i=1}^{n} \sum_{j=1}^{n} (\mathrm{E}(Y_i Y_j) - \mathrm{E}(Y_i)\mathrm{E}(Y_j)). \tag{20}$$

Since the expected values of all Y_i are the same, and the covariance between each Y_i, Y_j are the same, when $i \neq j$, the result can also be written as

$$\begin{aligned} \mathrm{Var}(Z) &= \sum_{i=1}^{n}\sum_{j=1}^{n}(\mathrm{E}(Y_iY_j)) - (n\mathrm{E}(Y))^2 \\ &= n(n-1)\mathrm{E}(Y_1Y_2) + n\mathrm{E}(Y^2) - (n\mathrm{E}(Y))^2. \end{aligned} \tag{21}$$

As an example, the values of these parameters are calculated for the exponential distribution.

4.6.2 Exponential distribution

The exponential distributions are a widely-used class of probability distributions that can be used to depict memoryless events, where the time that has already elapsed has no bearing on future observations. The probability density function and the cumulative distribution function of an exponential distribution are

$$f(x) = \alpha e^{-\alpha x} \tag{22}$$

and

$$F(x) = 1 - e^{-\alpha x}. \tag{23}$$

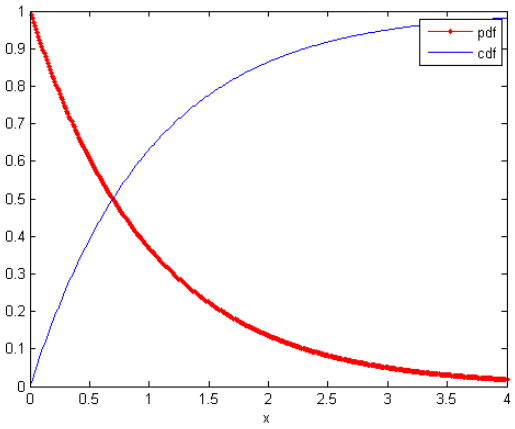

Figure 5: *The probability density function and the cumulative distribution function of an exponential distribution with the rate parameter* $\alpha = 1$.

The mean and variance of an exponentially distributed variable are, respectively,

$$\mathrm{E}(X) = \frac{1}{\alpha} \tag{24}$$

and

$$\mathrm{Var}(X) = \frac{1}{\alpha^2}. \tag{25}$$

In order to generate exponentially distributed random variables in the simulator, inverse transform sampling is used. The inverse of the cumulative distribution function of the exponential distribution is

$$F^{-1}(p) = \frac{-\ln(1-p)}{\alpha}. \tag{26}$$

Therefore random exponentially distributed variables can be generated with the formula

$$X = \frac{-\ln(U)}{\alpha}, \tag{27}$$

where U is a random variable that is uniformly distributed on $(0; 1)$ (which standard Java libraries can generate).

Based on Equation (15), the cumulative distribution function of the maximum of two exponentially distributed variables, ie. the download time of any randomly sampled chunk, is

$$F(y) = F(x)^2 = (1 - e^{-\alpha x})^2 = 1 - 2e^{-\alpha x} + e^{-2\alpha x}. \tag{28}$$

The probability density function can then be easily calculated from the cumulative distribution function (it could also be verified by using Equation (16)):

$$f(y) = \frac{d}{dt}F(y) = 2\alpha(e^{-\alpha x} - e^{-2\alpha x}). \tag{29}$$

The expected value of such a distribution is

$$\begin{aligned} \mathrm{E}(Y) &= \int_0^\infty 1 - F(y)dy = \int_0^\infty 2e^{-\alpha x} - e^{-2\alpha x}dx \\ &= \int_0^\infty 2e^{-\alpha x}dx - \int_0^\infty e^{-2\alpha x}dx = \frac{2}{\alpha} - \frac{1}{2\alpha} = \frac{3}{2\alpha} \end{aligned} \tag{30}$$

ie. $\frac{3}{2}$ times the expected value of an exponential distribution.

The expected value of the square of such a distribution, which is needed in calculating

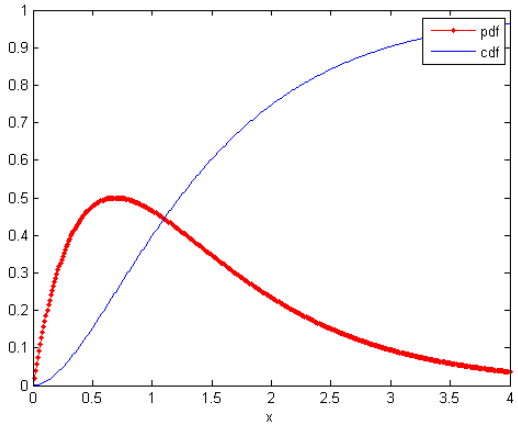

Figure 6: *The probability density function and the cumulative distribution function for a distribution depicting the maximum of two independent exponentially distributed variables with rate parameters* $\alpha = 1$.

the variance, is

$$\begin{aligned} \mathrm{E}(Y^2) &= 2\int_0^\infty t(1-F(y)) = 2\int_0^\infty x2e^{-\alpha x} - xe^{-2\alpha x}dx \\ &= 2\int_0^\infty x2e^{-\alpha x}dx - 2\int_0^\infty xe^{-2\alpha x} \\ &= 4\left|_{x=0}^{\infty}(\frac{-1}{\alpha^2} - \frac{x}{\alpha})e^{-\alpha x} - 2\right|_{x=0}^{\infty}(\frac{-1}{(2\alpha)^2} - \frac{x}{2\alpha})e^{-2\alpha x} \\ &= \frac{4}{\alpha^2} - \frac{2}{(2\alpha)^2} = \frac{7}{2\alpha^2}. \end{aligned} \tag{31}$$

Based on the above two results, the variance of the maximum of two exponentially distributed variables can be seen to be

$$\mathrm{Var}(Y) = \mathrm{E}(Y^2) - \mathrm{E}(Y)^2 = \frac{7}{2\alpha^2} - (\frac{3}{2\alpha})^2 = \frac{5}{4\alpha^2} \tag{32}$$

The expected value of the product of two variables drawn from two such distributions (which are not independent)—which is needed in determining the covariance—can be calculated to be

$$\mathrm{E}(Y_1Y_2) = \frac{2E(Y^2) + E(Y)E(X)}{3} = \frac{\frac{7}{\alpha^2} + \frac{3}{2\alpha^2}}{3} = \frac{17}{6\alpha^2}. \tag{33}$$

Using these results along with Equation (21), the variance for the distribution describing the total transfer time of a peer downloading a file of n chunks can be shown

to be

$$\begin{aligned} \mathrm{Var}(Z) &= n(n-1)\mathrm{E}(Y_1Y_2) + n\mathrm{E}(Y^2) - (n\mathrm{E}(Y))^2 \\ &= n(n-1)(\frac{17}{6\alpha^2}) + n(\frac{7}{2\alpha^2}) - (n\frac{3}{2\alpha})^2 \\ &= \frac{n}{\alpha^2}(\frac{7n}{12} + \frac{2}{3}). \end{aligned} \tag{34}$$

The mean transfer time for a file of n chunks using transfer times drawn from such a distribution would be

$$\mathrm{E}(Z) = \frac{3n}{2\alpha}. \tag{35}$$

As can be seen, this is simply the mean download time of one chunk multiplied by the number of chunks. In the actual simulator, the rate parameter will be altered so that the predicted mean transfer times will remain constant even when the number of chunks changes.

4.6.3 Erlang-k distribution

The Erlang-k distribution denotes a class of distributions that consists of the sum of k independent, identically distributed exponentially distributed variables. Its probability density function and cumulative distribution function are

$$f(x) = \frac{\alpha^k x^{k-1} e^{-\alpha x}}{(k-1)!} \tag{36}$$

and

$$F(x) = \frac{\gamma(k, \alpha x)}{(k-1)!}, \tag{37}$$

where γ denotes the lower incomplete gamma function $\gamma(s,t) = \int_0^t x^{s-1}\, e^{-x}\, dx$.

The mean and variance of the distribution are

$$\mathrm{E}(X) = \frac{k}{\alpha} \tag{38}$$

and

$$\mathrm{Var}(X) = \frac{k}{\alpha^2}. \tag{39}$$

Erlang-k distributed random variables can be generated by using Equation (27) to generate k exponentially distributed random variables and simply adding them together.

4.6.4 Hyper-exponential distribution

The hyper-exponential distribution is a probability distribution that gets its value from any one of k exponential distributions with a probability p_i, where $i = 1, ..., k$. The probability distribution function of the hyper-exponential distribution is

$$f(x) = \sum_{i=1}^{k} f_{X_i}(x) p_i, \tag{40}$$

where X_i is a random variable from one of the k exponential distributions. This can also be written as

$$f(x) = \sum_{i=1}^{k} \alpha_i e^{-\alpha_i x} p_i, \tag{41}$$

where α_i is the rate parameter of a particular exponential distribution.

The mean and variance of the distribution are

$$\mathrm{E}(X) = \sum_{i=1}^{k} \frac{p_i}{\alpha_i} \tag{42}$$

and

$$\mathrm{Var}(X) = 2 \sum_{i=1}^{k} \frac{p_i}{\alpha_i^2} - \mathrm{E}(X)^2 \tag{43}$$

Hyperexponentially distributed random variables can be generated by first determining the value of p from a uniform distribution and determining the correct α_i for that p. Equation (27) can then be used to calculate an exponentially distributed random variable with the correct value of α_i.

4.6.5 Discrete service times

The service times can also be set to be drawn from a discrete set of possible values. With the most simple case, each peer can be set to have the same transfer rate. In this work, when discrete service times are used, the transfer rate of each peer will be equal. This means that there will also be no variance in the service times. Furthermore, the maximum of two service times will always be equal to the mean service time, ie. the download time and service time will be equal.

5 Results

This section will show the results obtained by running the simulator. First, a sample of the simulator's output is given. This is followed by results that compare the simulator's results using a single chunk model to those calculated by Bhusal in [7], in order to determine how well the results of the two simulators match. The analysis is then extended to models that consist of multiple chunks, while varying the amount of chunks, the peer selection policy and the transfer time distribution. This is followed by an analysis of the mean transfer times encountered in the system when the arrival rate of peers and the departure rate of seeds is varied. Finally, these results are compared with some results from earlier research.

5.1 Example results from the simulator

Appendix A contains results that are presented as examples of a part of the simulator's output. The first run includes event information; it is mostly only used for debugging to see if the program is functioning correctly. The second run only includes peer counts, in a way that could easily be transferred and plotted using some other tool (ie. Matlab or Excel). Many different statistics about actual transfer time distributions (variance, skew) etc. could also be easily added. In both cases, $\frac{1}{\lambda} = 5$, $\frac{1}{\alpha} = 10$, $\frac{1}{\gamma} = 100$, $n = 5$, the peer selection strategy was rarest first, and the service time distribution was exponential.

5.2 Single-chunk models

To test the simulator and see how its results compare with related research (namely the results given by Bhusal in [7] using a fluid model and a Markov chain model), the service times are drawn from an exponential distribution, while the number of chunks is kept at 1. The arrival rate of peers $\lambda = 10$, and the departure rate of seeds $\gamma = 0.5$. The underlying exponential distribution was given the rate parameter $\alpha = 1$. The actual rate for the completion of transfers is not known exactly, since it will also depend on waiting times within the system. With a single-chunk model it is expected to be close to α when the mean number of seeds in the system is significantly larger than the mean number of downloaders (which will happen when $\gamma \ll \alpha$).

The results are sample averages calculated over 10,000 runs of the simulation. In order to get more similar results between the two methods and unlike in the other simulations described in this thesis, the total download times are determined only by the transfer rate of the downloader, and the upload rate of the seeder is not considered in the calculations. The original seed is set to never leave the system, so that early death of the file-sharing process will not skew the results. This will, however, lead to the number of seeds being one too high during most of the simulation.

While there is variation in each individual run of the simulation, the results will

even out over the sample size and the mean number of seeds will eventually reach an equilibrium state which is not related to the service time distributions. It will simply be the value $\frac{\lambda}{\gamma} + 1$, ie. the arrival rate of peers divided by the departure rate of seeds, plus the initial seed. The number of downloaders does not follow such a straightforward formula, and will be analyzed in further subsections. In this particular simulation, it is seen to be close to $\frac{\lambda}{\alpha}$, since the waiting times within the system are low and the only thing affecting actual download speeds is the transfer rate of each particular downloader.

Results from the comparison between the simulator used in this work and that of Bhusal's can be seen in Figure 7. The shapes of the plots for the number of downloaders and seeds seem to match quite closely with the numbers determined by the fluid model employed by Bhusal. The different methodologies probably account for the variation of the maximum peer numbers and the time it takes to reach an equilibrium state. When using the fluid model, it is assumed that the transfer rates are restricted only by the total bandwidth with each downloader getting its share. With the simulation, each peer may additionally have to wait for the completion of another transfer before starting its download. Straight comparisons with the Markov chain model used by Bhusal are not feasible by looking at the plots alone, since his plotted results are calculated over just one simulation run, and it is impossible to completely separate random fluctuation from the underlying function on the number of peers.

In another simulation—the results of which are shown in Figure 8—all of the other variables are kept the same as in the previous simulation but the simulation is altered to account for the transfer times of both the seed and the downloader. The download time when transferring a chunk is calculated as the maximum of these two times. In this case, it takes much longer for the simulation to reach an equilibrium state, and the number of downloaders reaches a higher peak before the number of seeders starts catching up. This is not surprising, since the expected value of the transfer times are known to be around 1.5 times higher when using an exponential distribution, as shown in Equation (30).

When the expected value of the service time is reduced to two thirds of its original value to account for the increase experienced in the previous simulation, the actual values for the peer numbers seen in the simulation drop a bit under the values seen in Figure 7. This is believed to demonstrate the effect of seeds whose mean upload times are higher being more likely to leave the system during a transfer, as predicted in Section 4.6. The results of this simulation can be seen in Figure 9.

5.3 Progress of peer amounts in multiple-chunk models

In this section, the effect of having the file split into multiple chunks is studied. The effects of peer selection policies and service time distributions on multi-chunk models are also examined. This analysis first concentrates on the initial progression of the number of downloaders and seeds, similar to those shown in Figure 7. The

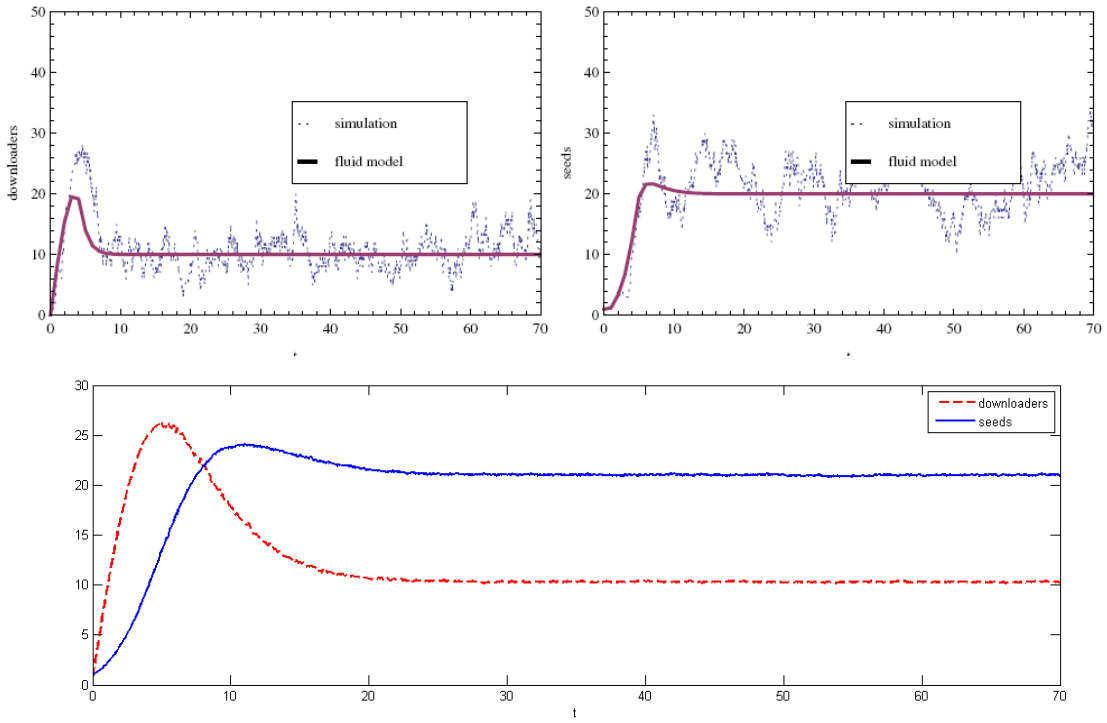

Figure 7: *The top figures show the number of downloaders and seeds as calculated by the fluid model and a simulation in [7]. The bottom figure shows the number of downloaders and seeds calculated by the simulator used in this work, described in Section 4, when the simulator only considers the transfer rate of the downloader in determining the total transfer rate.*

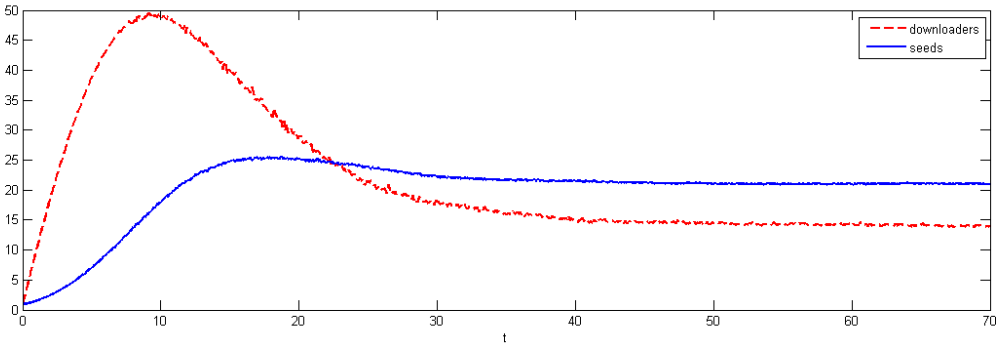

Figure 8: *The number of peers calculated by the simulator when the files consist of one chunk. Note the different scale for the peer numbers when comparing with Figure 7.*

effects on the mean service times is studied further on in Section 5.4.

The expected service time of the file is kept constant throughout the simulations, which means that the rate parameter of each distribution α is always multiplied by the number of chunks n.

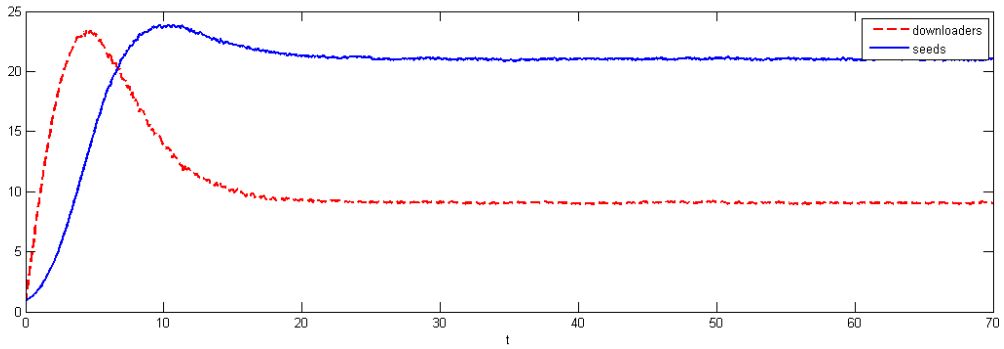

Figure 9: *The number of peers calculated by the simulator when the files consist of one chunk and the mean service time is reduced to two thirds of its previous value.*

5.3.1 Effect of the number of chunks

Figure 10 shows the changes to the number of seeders and downloaders in the system when the number of chunks is altered. The transfer rate parameter is changed as mentioned above while all of the other parameters are kept the same as they were for Figure 8 ($\lambda = 10$, $\gamma = 0.5$, exponentially distributed service times and rarest first peer selection policy). The greatest change to the numbers is seen when the model is changed from one chunk to two chunks. This is not unexpected: a similar result was given by Susitaival and Aalto in [26].

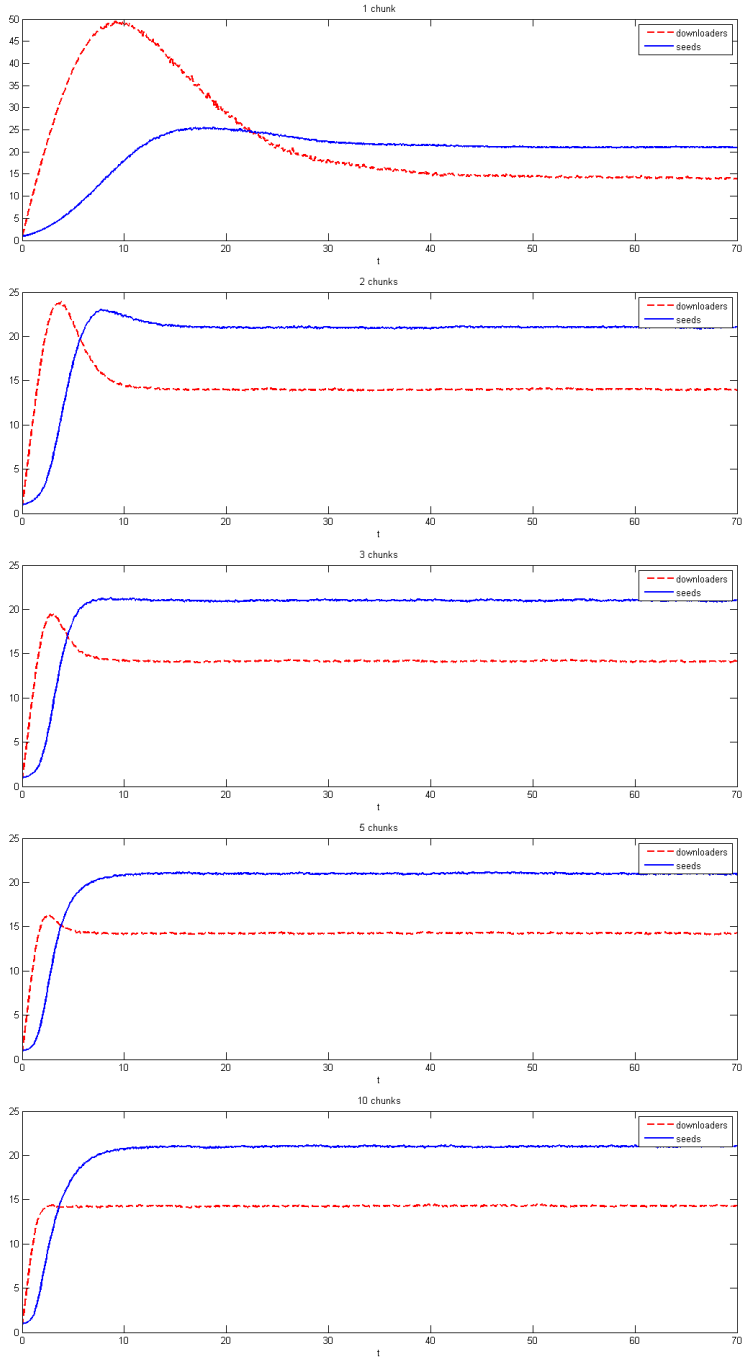

Figure 10: *The figures show the number of seeds and downloaders in a system when the service times of the peers are exponentially distributed and the numbers of chunks are 1, 2, 3, 5 and 10. When the number of chunks in the system is increased, the number of downloaders in the system can be seen to reach its equilibrium state faster. The largest difference is seen when the system is changed from a single-chunk model to a two-chunk model. The other parameters are kept at* $\lambda = 10$, $\alpha = 1$, *and* $\gamma = 0.5$, *the same values as were used for Figure 8. The peer selection policy is rarest first.*

5.3.2 Effect of the peer selection policy

The effect of the peer selection policy is illustrated in Figure 11. The rate parameters are again kept the same as in the previous section ($\lambda = 10$, $\alpha = 1$, $\gamma = 0.5$, with exponentially distributed service times). The number of chunks is kept at 5 while the peer selection policy is varied. The random peer and random chunk selection policies can be seen to perform slightly worse than the rarest first policy, while the most common first policy causes a larger jump in the numbers of downloaders and seeds before the graphs reach an equilibrium state.

5.3.3 Effect of the service time distribution

Figure 12 shows the effect of changing the service time distribution between the exponential, Erlang-k and hyperexponential distributions, as well as a constant service time, on the number of downloaders and seeds in a simulated peer-to-peer network. In order to compare the behavior of the distributions, the mean time of each distribution will be the same, $\mathrm{E}[X] = 1$, meaning that the service time given to each peer would average to the same value regardless of the distribution used. The actual download times of the chunks will consist of the maximum of two peers' service times; the transfer time will be the sum of all of a peer's download times plus the waiting times experienced by that peer.

The parameters of the exponential distribution are again kept unchanged from the previous sections, while the number of chunks is kept at 5. For the Erlang-k distribution, $k = 10$ and the rate parameter of each underlying exponential distribution is increased by the same factor, ie. $\alpha = 10$ in these cases. The hyperexponential distribution consists of two exponential distributions with mean values $\mathrm{E}[X_1] = 0.1$ and $\mathrm{E}[X_2] = 9.1$ and weights $p_1 = 0.9$ and $p_2 = 0.1$.

The variances of the different distributions can be calculated by using Equations 25, 39 and 43 given in Section 4.6. For the exponential distribution, $\mathrm{Var}[X]_{exp} = 1.00$, for the Erlang-10 distribution, $\mathrm{Var}[X]_{erl} = 0.10$, for the hyperexponential distribution, $\mathrm{Var}[X]_{hyper} = 15.58$. When the service time is kept constant, the variance will naturally be at 0. Since $\mathrm{E}[X] = 1$ for each distribution, the coefficient of variation will be the square root of the variance for each distribution.

The evolution of peer numbers seems to be smoother when the distribution's variance inceases. When the variance is smaller, there will be a larger spike before the number of downloaders evens out, but the balance state that is eventually reached will have fewer downloaders. It is not proven here that these effects result solely from the variance of the distribution; other properties of the distributions might also play some part.

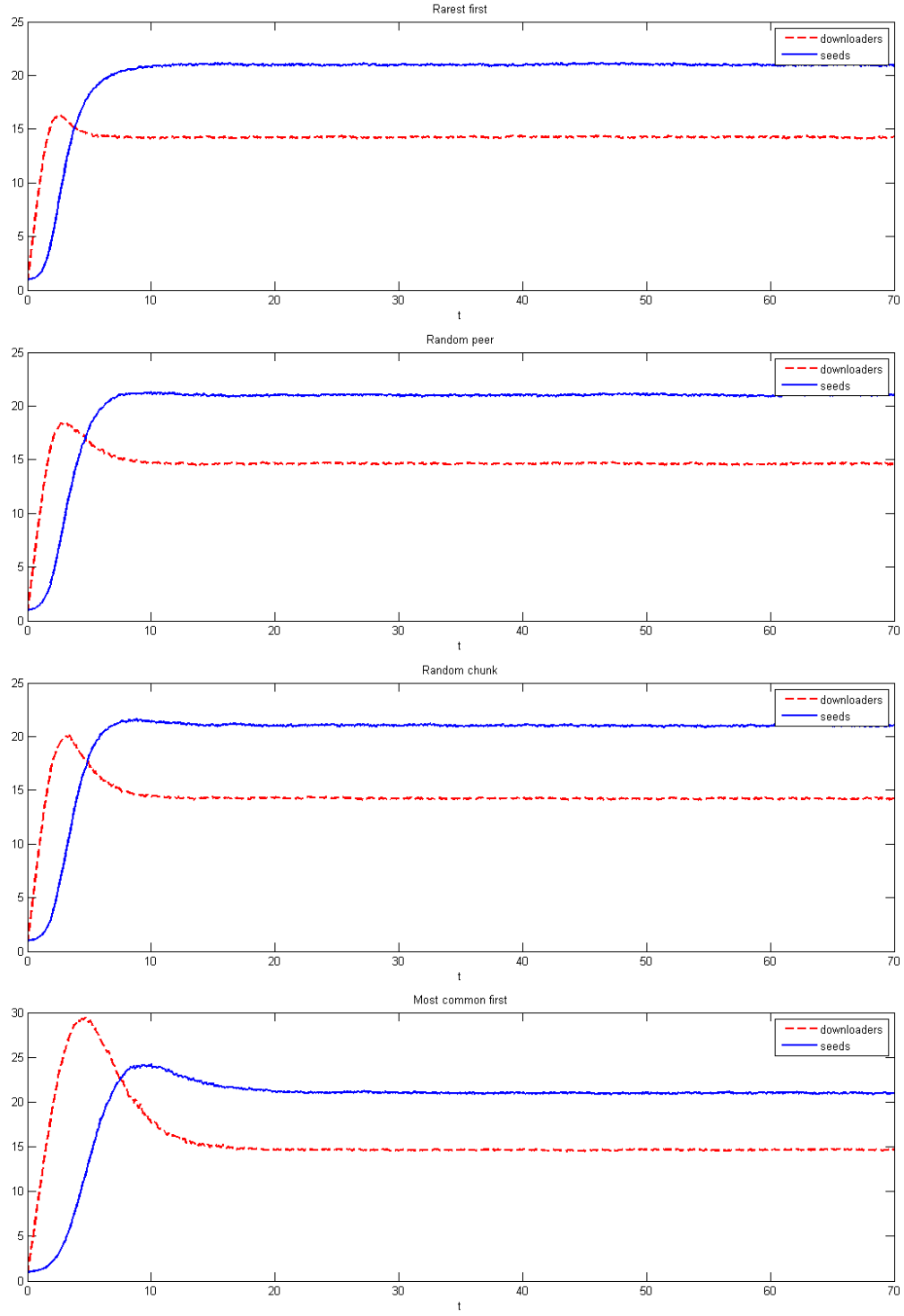

Figure 11: *These figures show the effect of the peer selection policies on the development on the number of downloaders and seeds. All of the simulations used a five-chunk model and exponentially distributed service times.*

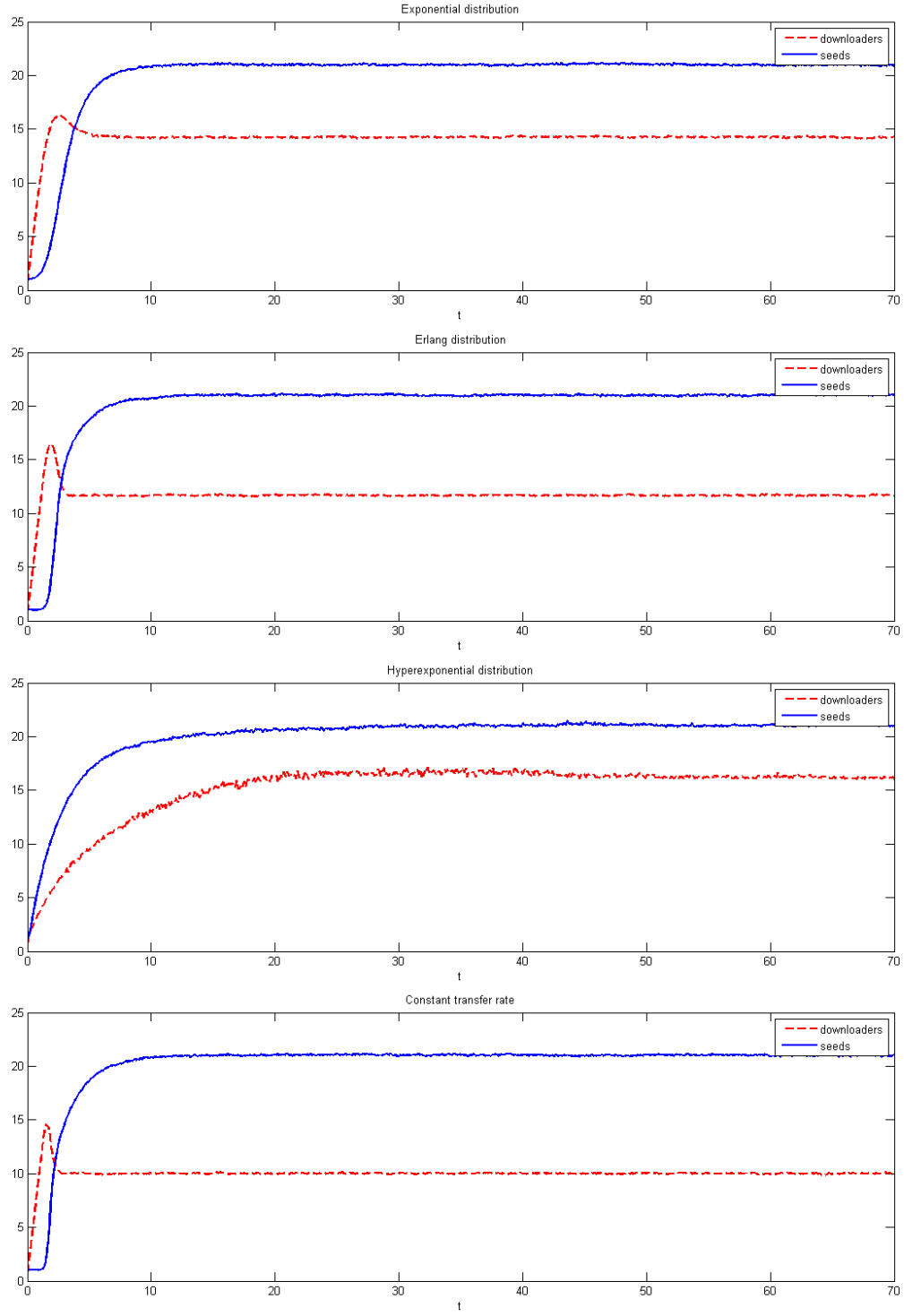

Figure 12: *The figures show the effect of the service time distribution on the evolution of the number of downloaders and seeds in a system. All of the simulations used a five-chunk model while the peer selection policy was always rarest first.*

5.3.4 Altering a combination of variables

As is demonstrated in Figures 10, 11 and 12, increasing the number of chunks will cause the system to stabilize faster, as will having a service time distribution with a lower variance. The rarest first peer selection policy is the most efficient, the most common first policy is the least efficient, and the random peer and random chunk policies fall in the middle of the two extremes.

When combinations of different variables—number of chunks, peer selection policy and service time distribution—are considered, the system will behave 'predictably'. This means that for example a service time distribution with a lower variance will result in a sharper peak in the number of downloaders before smoothing out, while having a peer selection policy of most common first will result in a higher peak in the number of downloaders (when compared with a rarest first selection policy), regardless of the other parameters that were changed. There were no anomalous results that would have been unexpected in light of the previous results.

5.4 Steady-state service times in multi-chunk models

This section contains results about the effect of varying the service time distribution, the number of chunks, the arrival rate of peers, and the departure rate of seeds on mean transfer times experienced by peers in the system. The analysis concentrates on systems that have reached an equilibrium state: the initial part of the simulation is ignored and the results are only gathered after a period of time has passed—after $t = 1000$ in a simulation where $t_{max} = 100000$. This length of time is sufficient to reach a steady state in most, but not all cases. However, since the analysis will also look at some cases where the simulation will never reach an equilibrium and the number of peers will keep growing indefinitely, it was necessary to establish some semi-arbitrary cutoff point.

Some results about the mean transfer times can already be seen in the previous section; the mean transfer time $\mathrm{E}[T]$ of a downloader is roughly $\frac{x}{\lambda}$, where x is the number of downloaders when a stable phase has been reached in the plotted results. The mean download times seem to increase as the variance of the used service time distribution increases, since there will be more disparity between the service times of two peers and more bandwidth wasted.

5.4.1 Effect of the number of chunks

The effect of the number of chunks on the mean transfer times is studied in this section. In these simulations, $\lambda = 10$, $\alpha = 1$ and $\gamma = 0.5$ while the peer selection policy is rarest first. The parameters of the different distributions were kept the same as in Section 5.3.3.

With an exponential distribution, the mean transfer times were 1.419, 1.387 and 1.419 for 1, 2 and 10 chunks. With a hyper-exponential distribution, the mean

transfer times were 3.732, 1.481 and 1.586 for 1, 2 and 10 chunks. With an Erlang distribution, the mean transfer times were 1.172, 1.162 and 1.165 for 1, 2 and 10 chunks. When a constant service time is used, the mean transfer times were 1.006, 1.000 and 1.000 for 1, 2 and 10 chunks. A graph of the results can be seen in Figure 13. There is some random variation in the numbers, but the general trend remains unchanged. After increasing the number of chunks from one to two, the mean transfer times will decrease somewhat. Further increases to the number of chunks have little effect on the mean transfer times, but they trend slightly upwards, ie. towards longer transfer times. With the constant service times, this increase does not occur.

The explanation for the decrease in transfer times after the number of chunks is increased from 1 to 2 is simple, the system simply becomes slightly more efficient with smaller waiting times. With these parameters, the system will be efficient with most distributions even when the number of chunks is at 1; the lone exception are the hyper-exponentially distributed service times where the improvement is more obvious.

The values of λ and γ can have an effect on the number of chunks needed to reach peak performance in the system. As can be seen further on in this thesis, increasing the number of chunks to higher than 2 can improve the performance depending on the values of the other parameters. However, at some stage the system will always reach a point where further increases in the number of chunks will lead to either stable or slightly increasing mean transfer times. The explanation offered to these increases is that it is just a feature in the simulation: with more chunks, the peers with larger service times are less likely to leave during an upload, while with only two chunks, these peers might be unlikely to finish even a single complete upload, which will then be finished with another peer with likely a higher upload rate. With the constant service times, there are no peers with longer service times than others, which would explain the even trend in the mean transfer times.

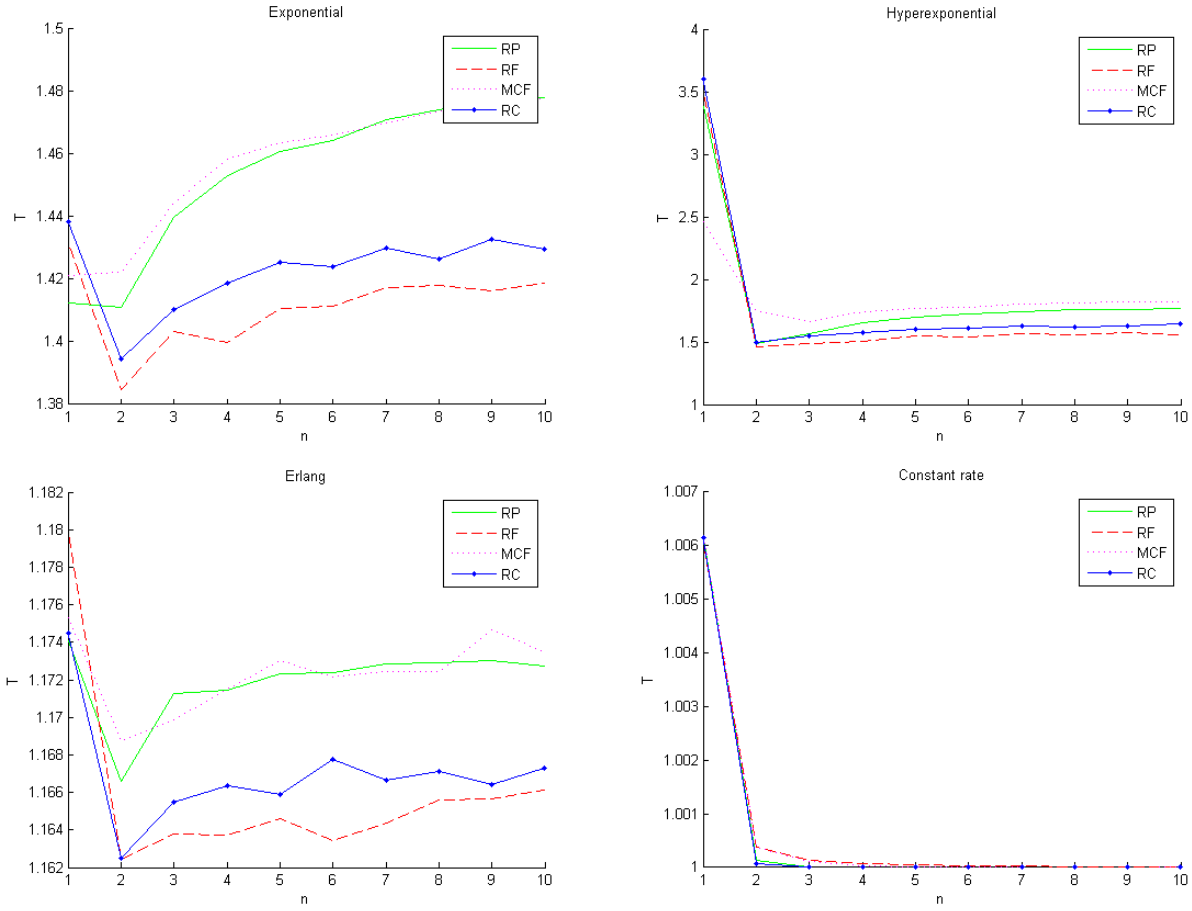

Figure 13: *The effect of varying the number of chunks (n) on the transfer times of a peer-to-peer network. Note the different scales in the figures.*

5.4.2 Effect of the arrival rate of peers

When the arrival rate of peers is varied, the effect on the transfer rates is very different from the model used by Susitaival and Aalto in [26]. The values of the departure rate of seeds and the mean completion time are kept constant at $\gamma = \frac{1}{2}$ and $\alpha = 1$ while the arrival rate λ is varied. With all of the different distributions and peer selection policies, the mean transfer times become slower when the arrival rate of peers begins to increase, and then speeds up again when the arrival rate increases even further. The peak is seen to be when the value of λ is around 1. This is illustrated in Figures 14 and 15 for two-chunk and 10-chunk models.

It stands to reason that the mean transfer times would increase when the arrival rate is initially increased from 0, since it becomes less likely that the initial seeder will be able to service peers that enter the system nearer to each other, while there are not enough other seeders to satisfy this demand. The decrease that is later seen in the figures probably happens since with these values of the parameters the seeds depart at a lower rate than the peers complete their downloads, ie. $\frac{1}{\gamma} > \mathrm{E}[Y]$, where $\mathrm{E}[Y]$ is the time it takes to transfer one chunk. This leads to an increased likelihood of having more seeders (and leechers that have already completed the tranfer of one chunk) than there are downloaders with no chunks when the total peer numbers grow.

The service time distribution of each peer plays a large role in the actual experienced download times within the system. However, the shapes of the plots are very similar, and the effect of the distributions seems to be close to a predictable scaling factor. There is less random variation in the plots for the distribution with least variance (constant service times), but the other plots are hard to put into order in this sense. It is likely that the plots for the constant service times give a good indication of how the system will behave when random variation is accounted for, but this is not certain based on these result alone.

In the two-chunk model, the most common first peer selection policy can be seen to have the worst performance of all tested selection policies. The other three policies result in more closely-matched graphs, with the rarest first policy usually performing best. When the ten-chunk model is used, the plots show a clearer distinction between the performances of the policies. This seems reasonable, since the higher number of chunks will allow the peer selection policy to have a larger effect on the performance of the system. The rarest first policy again performs the best, followed by the random peer policy, then the random chunk policy, and finally the most common first policy.

The plots in 16 and 18 contain a graphical representation of the transfer times of two-chunk and ten-chunk models when the value of γ is increased to ∞, ie. when the peers will leave the system as soon as they have completed their download. Figure 17 shows the two-chunk results plotted on a logarithmic scale to demonstrate a fuller range of some of the plots.

In the cases with infinite departure rates, the mean transfer times will increase

monotonically. This means that, on average, each peer will only burden the system further, since they will download more than they upload. The plots seem to again behave predictably when the service time is changed; the plots seem to be simply scaled versions of each other. However, this scaling can be more readily seen to extend also to the x-axis (denoting λ) of the plots. When the value of λ surpasses a certain value, the system will balloon out of control. This results from the inability of the initial peer and the downloaders that are sharing some of the chunks to service the entire system rapidly enough. It is expected that this value of λ will be close to $\frac{n}{E(Y)}$.

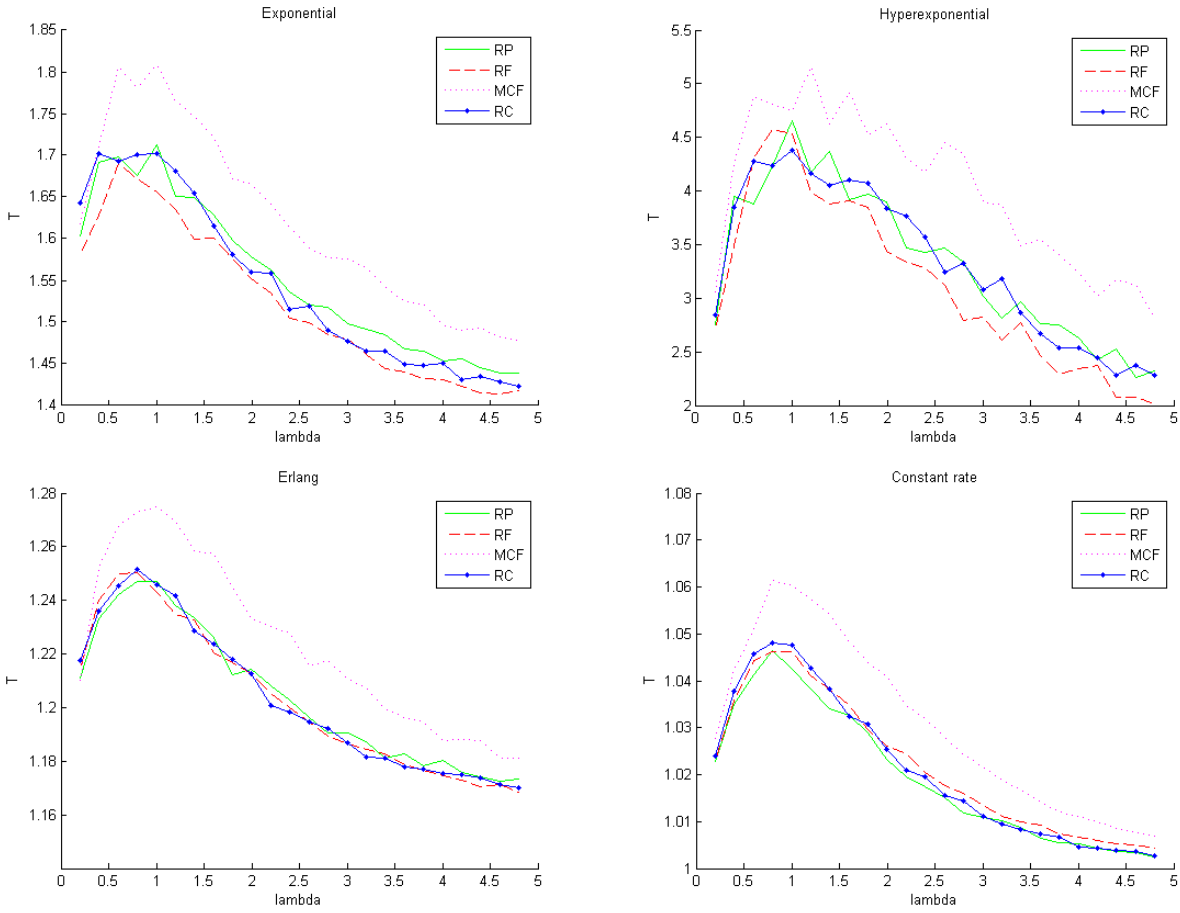

Figure 14: *The effect of varying the arrival rate of peers (λ) on the transfer times of a two-chunk model of a peer-to-peer network. The departure rate of seeds $\gamma = \frac{1}{2}$. Note the different scales in the figures.*

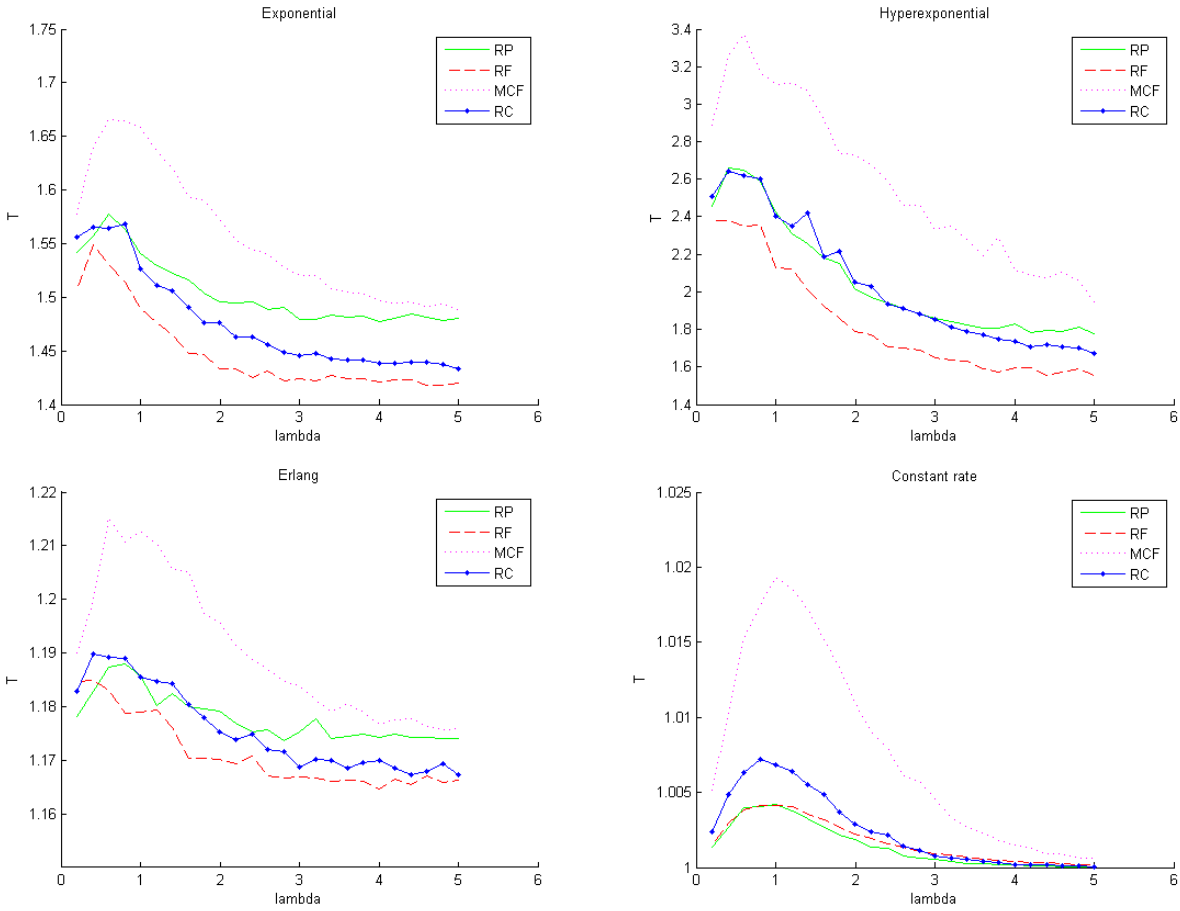

Figure 15: *When the number of chunks is increased to 10, the mean transfer times will be smaller, but the shapes of the plots remain similar.*

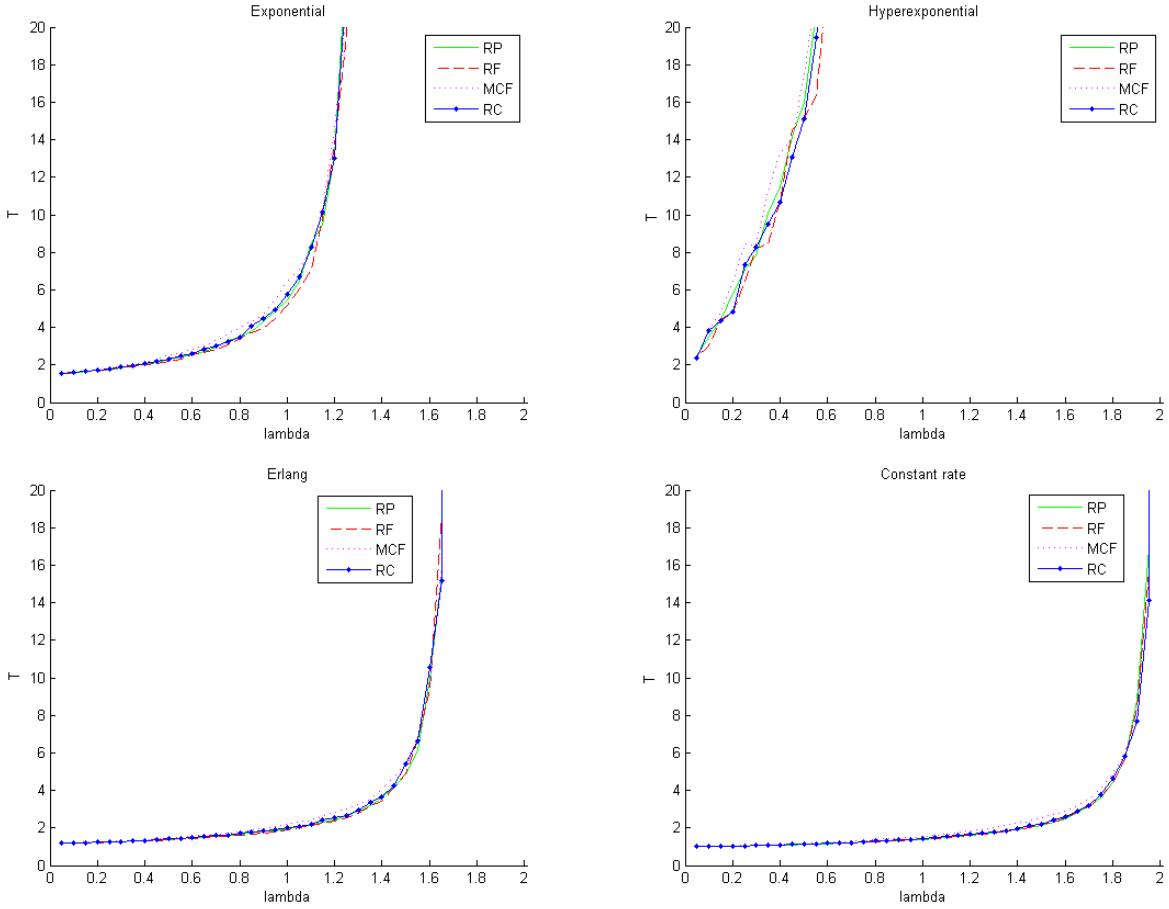

Figure 16: *In these plots, the value of γ is increased to infinity, ie. the peers will leave the system as soon as they have completed their download. The number of chunks is 2.*

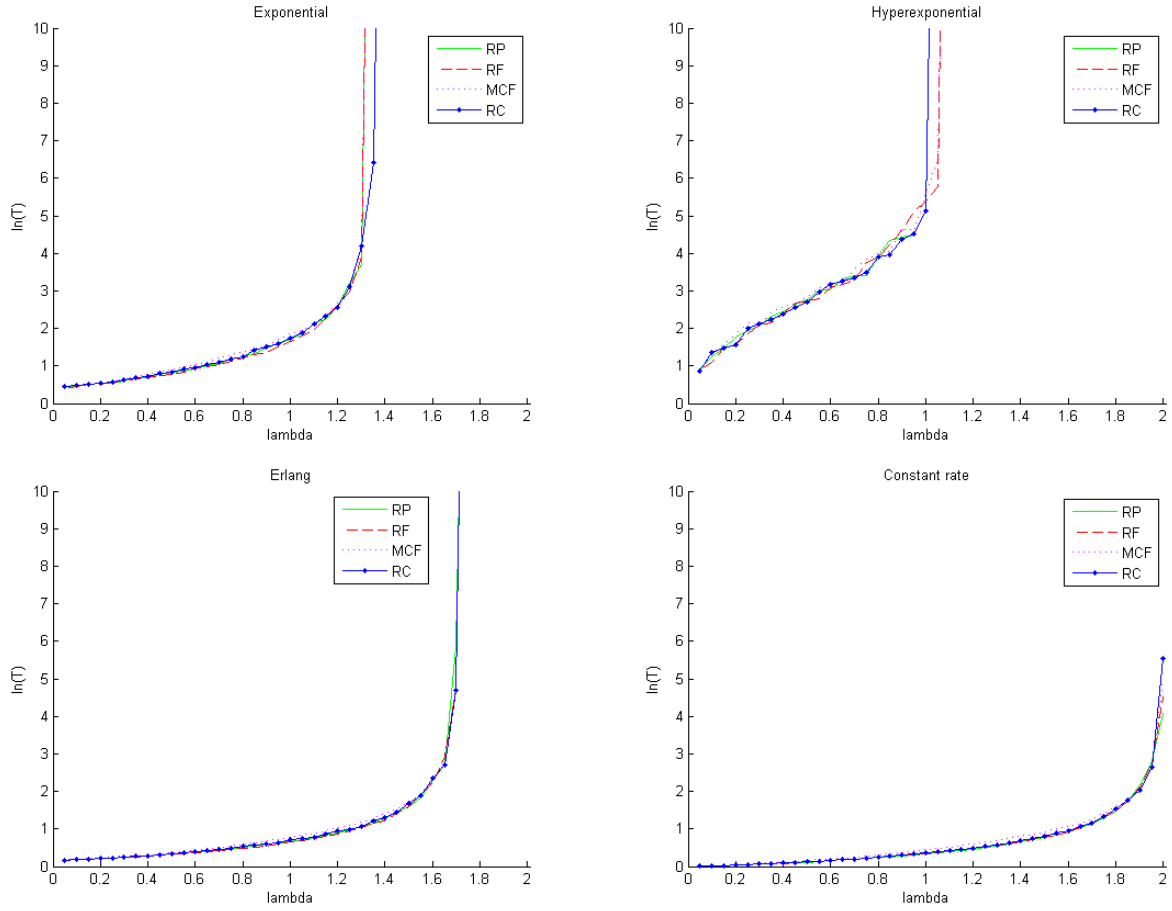

Figure 17: *These plots are equivalent to those in 16 but are changed to a logarithmic scale. This is done to demonstrate the full range of results from the simulations where hyperexponential distribution is used.*

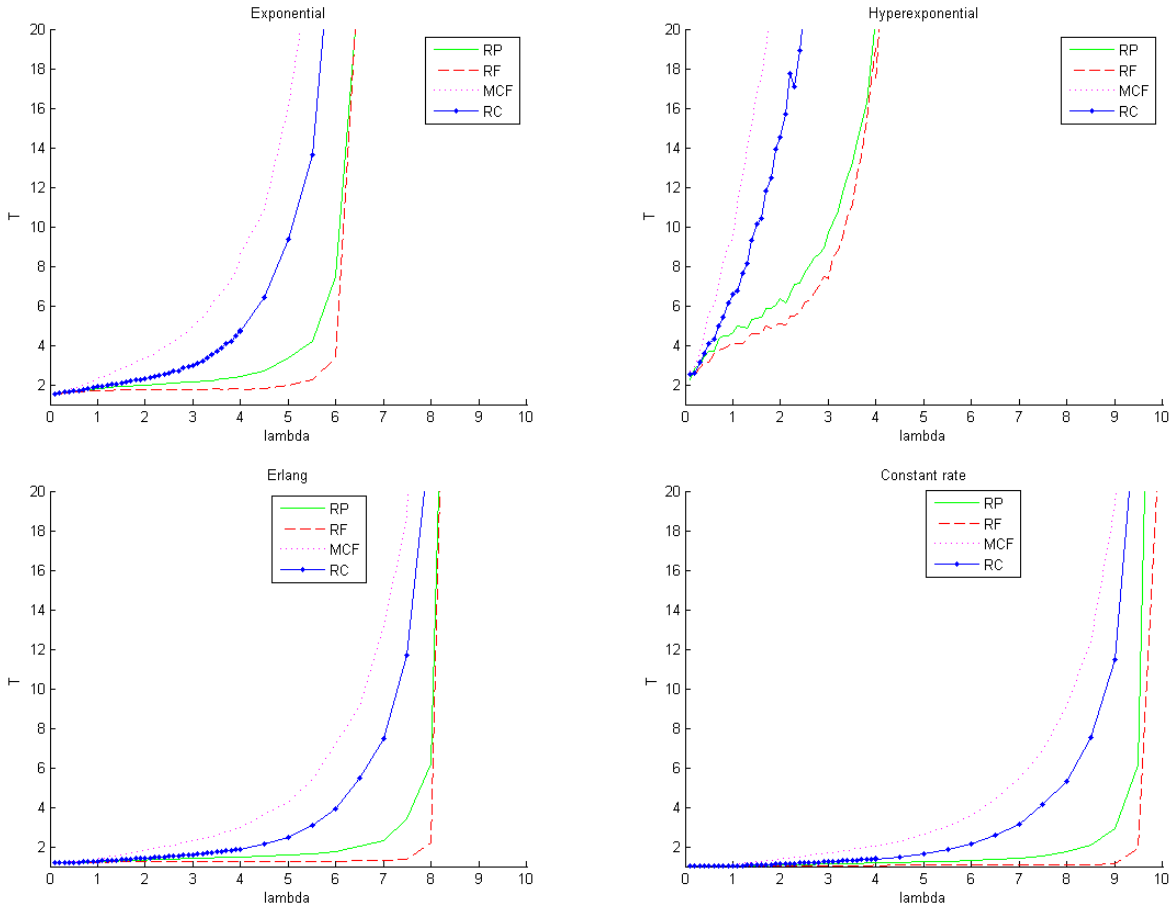

Figure 18: *The value of gamma is kept at infinity, and the number of chunks is increased to 10. Note the different scale of λ when compared with Figure 16. The maximum arrival rate before the system becomes unstable will be close to five times as large as before, resulting from the five-fold increase in the number of chunks. There is a clearer distinction between the different policies resulting from this increase, but otherwise the graphs closely resemble those shown in Figure 16.*

5.4.3 Effect of the departure rate of seeds

Next, the effect of varying the departure rate of seeds (γ) is studied. The transfer times of two-chunk models with different peer selection policies and service time distributions are plotted in Figure 19. The value of the peer arrival rate $\lambda = 1$ and the rate parameter of the distributions is $\alpha = 1$. As can be seen from the plots, the system remains stable with nearly every distribution class and value of $\frac{1}{\gamma}$. The lone expection being the hyperexponentially distributed service times when peers leave the system as soon as their downloads are finished ($\gamma = \infty$). As the actual mean transfer time is known to be highest when this distribution class is used, the results is not unexpected.

In Figure 20, the arrival rate of peers is increased to $\lambda = 5$. As was shown in Section 5.4.2 and illustrated in Figure 16, the system will buckle under the pressure of new arrivals and balloon out of control when the value of γ is high (and $\frac{1}{\gamma}$ is low).

In Figure 21, the arrival rate is kept at $\lambda = 5$ but the number of chunks in the models is increased to 10. The increase in the number of chunks will now cause the downloaders to contribute more of their upload bandwidth to the system, and the system will remain stable for all values of γ in most cases. The hyperexponentially distributed service times again provide the exception at $\gamma = \infty$.

Similarly to the cases where the arrival rate of peers was varied, the effect of the service time distributions seems predictable in the light of previous results. The shape of the plots is very similar, and a simple scaling factor that is related to the mean of the maximum of two peers' service times (ie. the download time) could be used to explain most of the variation in the plots.

Of the peer selection policies, most common first will again perform the worst while rarest first sees the best performance.

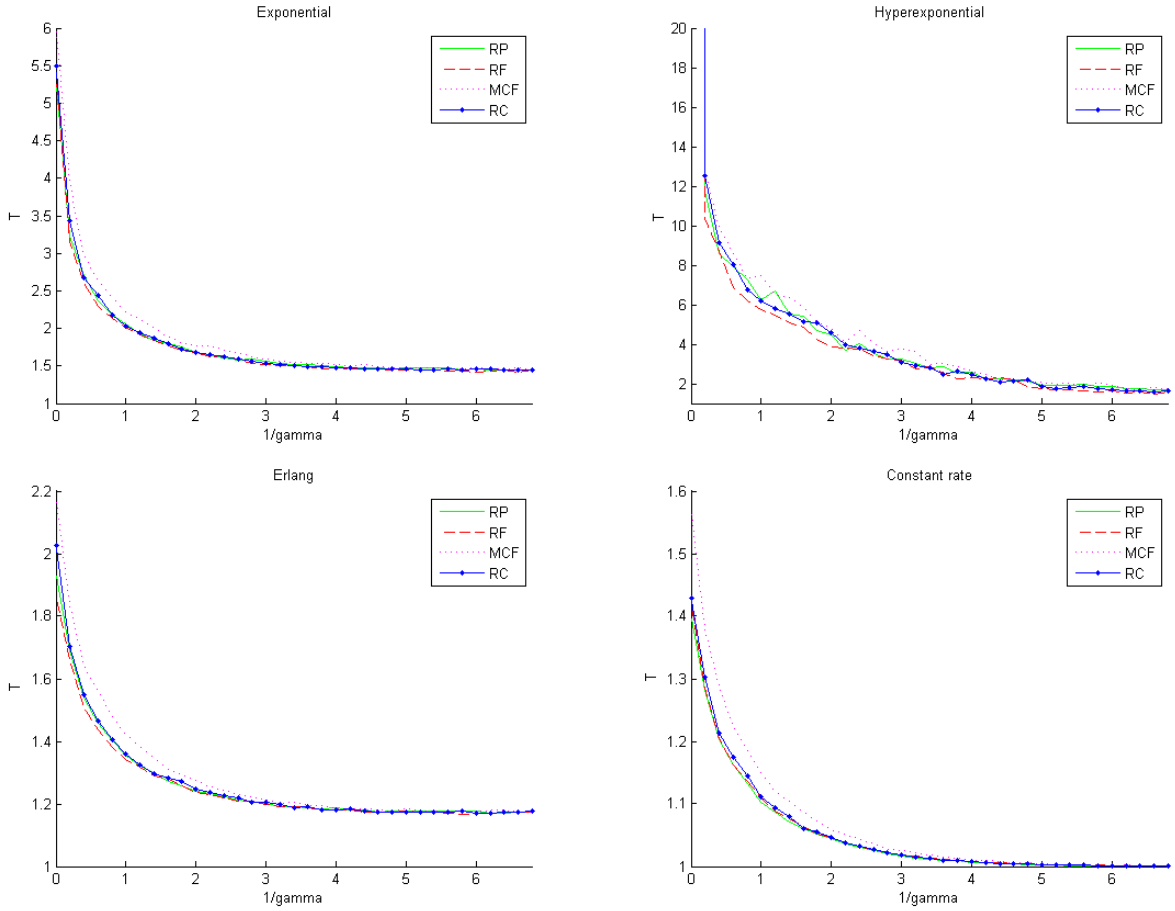

Figure 19: *The effect of varying the mean departure time of seeds ($\frac{1}{\gamma}$) on the transfer times of a two-chunk model of a peer-to-peer network. The value of λ is 1. Note the different scales in the images.*

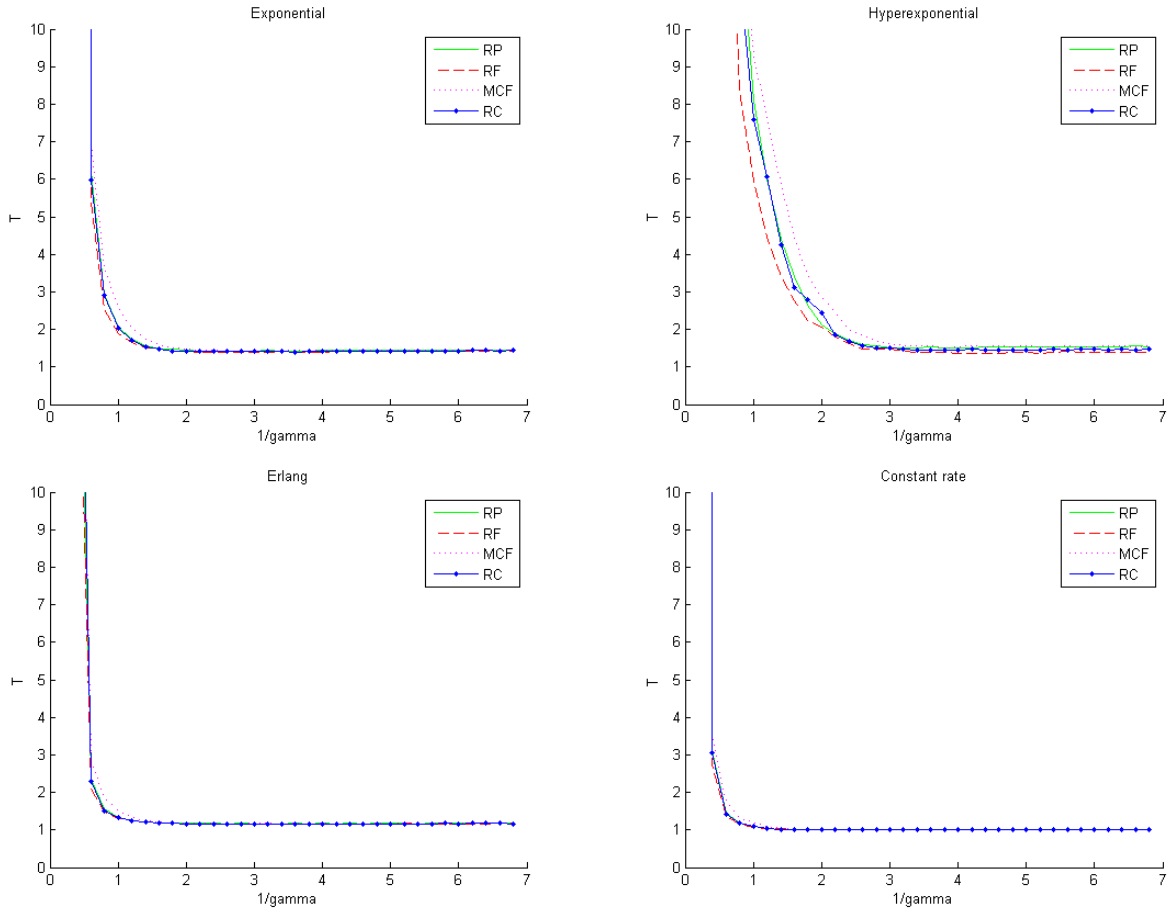

Figure 20: *The effect of varying the mean departure time of seeds ($\frac{1}{\gamma}$) on the transfer times of a two-chunk model of a peer-to-peer network. The value of λ is increased to 5.*

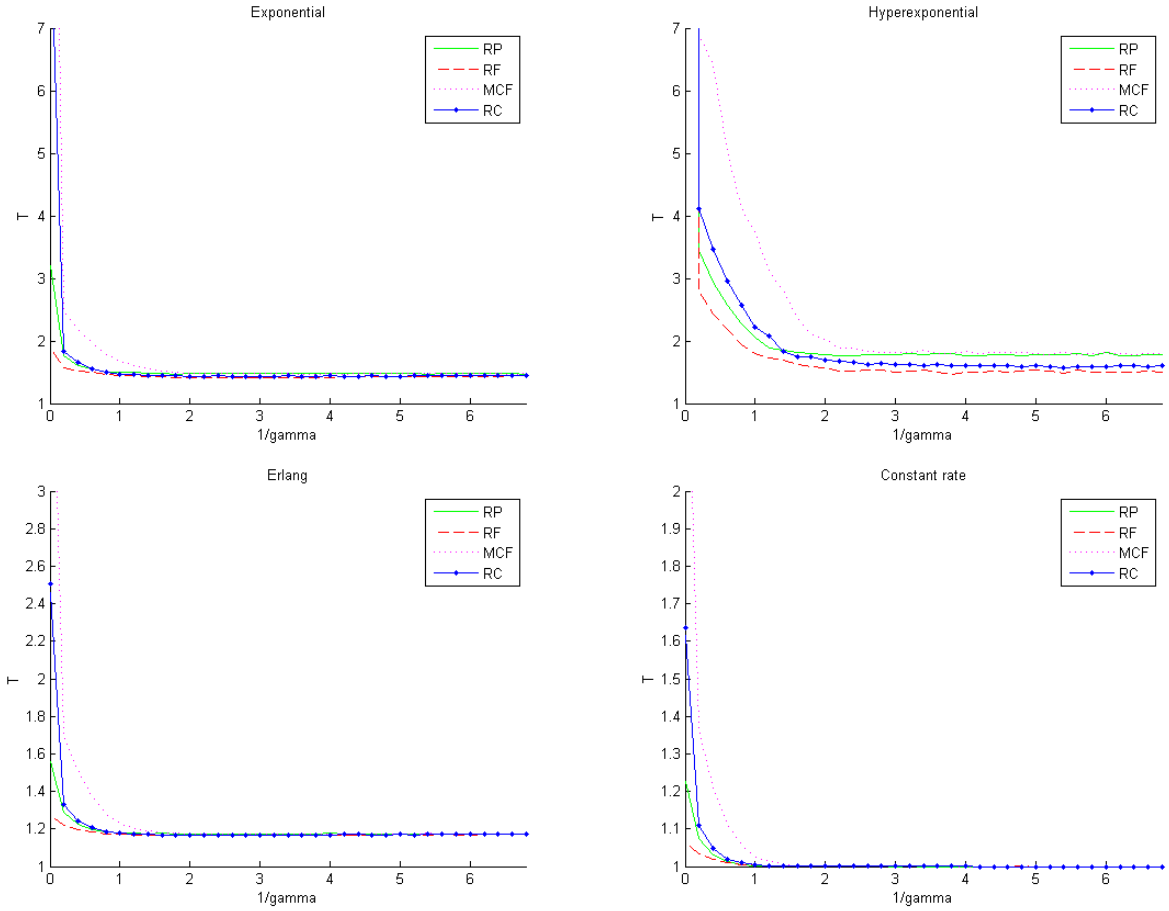

Figure 21: *The effect of varying the mean departure time of seeds ($\frac{1}{\gamma}$) on the transfer times of a ten-chunk model of a peer-to-peer network. The value of λ is 5. Again, note the different scales in the images.*

5.5 Comparisons with earlier research

When the results are compared to those given by the event-based simulator used by Susitaival and Aalto in [26], certain discrepancies can be seen. Most of these can be explained by the differing design philosophies behind the simulators: while each seed in Susitaival and Aalto's simulator can service an infinite amount of peers, in this work, they can service only one at a time. Neither of these two extremes exactly models the actual BitTorrent protocol, where peers will (at default settings) unchoke and upload to a maximum of four other peers at a time. The real performance could therefore be expected to fall somewhere in between the results given by these two simulators.

Having a higher arrival rate of peers will generally have a positive effect on the performance of the whole system, as long as the net effect of each peer on the system is positive. This will happen when a peer will remain in the system for a length of time after that peer has completed its download. That length of time should on average be longer than it takes that peer to download one chunk of the file. This somewhat contradicts the earlier results, given in Figure 6 of [26]. The effect can be explained by the different design of the simulators. In this work, the time it takes to download a chunk will only be dependent on the transfer rates of the two peers that are involved in the transfer, no matter how many or few available uploaders there are. The only effect that increases the mean total transfer times will therefore come from added queuing times, which will be less likely to be a problem when there are more total peers in the system.

When the departure rate of seeds is increased to ∞, there is a significant deviation in the results. In the similar study by Susitaival and Aalto, the peer amounts would oscillate between the two download states when the rarest first policy was used along with discrete service times. Each peer would start downloading the same chunk, and when the download of that chunk would finally complete, the peers that had already downloaded the other chunk would leave, and leave an ever-growing backlog on the remaining chunk with only one seed remaining to upload. On the other hand, when the service times were exponentially distributed, a similar effect was not seen and the system seemed to scale much better. In the results obtained in this thesis, the differences between the peer selection policies and service time distributions were much smaller. The effect of the peer selection policy is minimal, and the effect of the service time distribution seems predictable, since the rate at which new peers can be serviced is inversely proportional to the maximum of the uploader's and downloader's service times.

Another result that differs from earlier literature—namely [25] by Qiu and Srikant—is that the modeled system is not always stable (in Qiu and Srikant's paper this is the case when the sharing efficiency parameter $\eta > 0$, which simply means that the downloaders are sometimes uploading data). This can be seen most clearly in Section 5.4.2 when the departure rate of seeds is set to $\gamma = \infty$. The differences in the results have not been thoroughly examined, but many possible explanations exist; only one of them is offered here. In Qiu and Srikant's paper the downloaders

leave the system at a certain rate θ, which will lead to an eventual equilibrium point in their number, whether any of the peers actually complete their download or not. This parameter was not included in this work, but could have worked as a quick fix for the disparity. However, when the departure parameter was left out of the formula, the system would still scale. An explanation could be that the actual file sharing efficiency of peers seen in this thesis would approach zero as the number of peers in the system grows. One of the chunks could become a bottleneck in the system, since only one peer would have it at a time and any other peer would leave as soon as they downloaded it. This has not been verified.

6 Conclusions

When simulating and eventually designing a peer-to-peer system, a multitude of parameters needs to be taken into account, and the effects of all of these might not be immediately obvious. The results shown in this thesis demonstrate the importance of taking the service time distribution of peers into consideration when simulating a peer-to-peer system. The lower the variance in the peers' service times is, the faster the actual transfers will occur. Resulting from this, a simulated system might become unstable faster than expected if the actual transfer time distribution is not taken into account.

When the results are compared to similar research, the importance of the design decisions made when creating the simulator become obvious. While in this thesis the effects of the service time distribution do not seem unexpected, and the changes that result from altering the distribution behave in predictable fashion, the simulator introduced in [26] experienced more aberrant behavior without necessary being more 'right' or 'wrong'. Understanding the limitations of the simulational model being used is crucial, and testing a system with several different simulators might be necessary. One needs to remember that a simulator will not be an exact representation of a system, but can be useful in finding out some of its features faster than a full implementation and testing of a new system would allow. Implementing different service time distributions to a simulator should not be a difficult task in most cases, and could be helpful in discovering features of the system.

The largest effect that the peers' service time distribution has is the wasted bandwidth that results from disparities in the peers' own bandwidths. In reality, this shows a strength in the actual BitTorrent protocol, where peers will try to upload to those other peers from which they download the fastest. This eventually results in clustering of similar-bandwidth peers, which seems like it could lead to an increase in efficiency of the entire system.

As was to be expected, the rarest first policy produces the best results in every simulated case, while the most common first policy performs the worst. This gives further justification to the policy's use as the main peer selection policy in a BitTorrent network. However, the differences between the performances of the different policies are not overly large, and simply selecting a random peer or chunk when choosing what to download might be a justifiable design decision in some cases where it is computationally simpler. The peer selection policy will play a larger role in systems where the number of chunks is high.

In the future, a model which allows each peer to be involved in simultaneous transfers of multiple chunks could be used to better model an effective peer-to-peer system. Using a threaded programming architecture, which could also be implemented in Java, might enable a workable model, and the results given would be likely to correspond more closely to reality. However, the performance of the simulator itself would be much slower. Comparing the results of the two extremes of simulator types as seen in this thesis (where each peer could only upload and download from one

other user) and the paper by Susitaival and Aalto in [26] (where each peer could upload to every user in the system that was downloading a particular chunk) could give some indication on which parameters need to be evaluated more closely. The results could be expected to fall somewhere between these two simulator types.

This work concentrated on peers with equal upload and download rates, which is not usually the case in real-world peer-to-peer networks. Most personal Internet connections will have larger download bandwidth than upload bandwidth, and this could also be taken into account in future simulations, depending on the goals of the analysis. Setting the peers' upload rates to a percentage of their download rate should not be hard to accomplish. Similarly, some peers in these types of networks will simply be so-called freeloaders, who will contribute minimally to the upload rates of the system, and these users could also be taken into account.

A mathematical problem that arose during this work could later be examined more closely: is the maximum of two service times only dependent on the mean and variance of the underlying service time distributions, or is some other attribute of the distributions necessary to explain the result? This was briefly looked at during the work, but finding a conclusive result remained outside its scope.

References

[1] Nate Anderson, P2P traffic drops as streaming video grows in popularity. Ars Technica, September 02, 2008. http://arstechnica.com/news.ars/post/20080902-p2p-traffic-drops-as-streaming-video-grows-in-popularity.html

[2] Eric Bangeman, P2P responsible for as much as 90 percent of all 'Net traffic. Ars Technica, September 03, 2007. http://arstechnica.com/news.ars/post/20070903-p2p-responsible-for-as-much-as-90-percent-of-all-net-traffic.html

[3] MIT OpenCourseWare, Distribution of the Maximum of Independent Identically-distributed Variables. http://ocw.mit.edu/NR/rdonlyres/Civil-and-Environmental-Engineering/1-151Spring-2005/3547C56A-C1E6-4FF9-9AED-BDEF6B97330E/0/app11_max.pdf

[4] K. Aberer, M. Hauswirth, An overview on peer-to-peer information systems. Proceedings of WDAS-2002, 2002.

[5] Mario Barbera, Alfio Lombardo, Giovanni Schembra, Mirco Tribastone, An Analytical Model of a BitTorrent Peer. 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, pp. 482–489, 2007.

[6] Ernst W. Biersack, Pablo Rodriguez, Pascal Felber, Performance Analysis of Peer-to-Peer Networks for File Distribution. Lecture Notes in Computer Science, No. 3266, pp. 1–10, 2004.

[7] Manoj Bhusal, Effect of Service Time Distribution on the Performance of P2P File Sharing. Special Assignment in Networking Technology, Helsinki University of Technology, 2008.

[8] Thomas Bonald, Laurent Massouli, Fabien Mathieu, Diego Perino, Andrew Twigg, Epidemic Live Streaming: Optimal Performance Trade-offs Proceedings of 2008 ACM SIGMETRICS International, pp. 325–336, 2008.

[9] Z. Chen, Y. Chen, C. Lin, V. Nivargi, P. Cao, Experimental Analysis of Super-Seeding in BitTorrent. IEEE International Conference on Communications, pp. 65–69, 2008.

[10] Bram Cohen, Incentives Build Robustness in BitTorrent, Workshop on Economics of Peer-to-Peer Systems, 2003.

[11] G. de Veciana, X. Yang, Fairness, Incentives and Performance in Peer-to-Peer Networks. Proceedings of the annual Allerton conference on communication control and computing, Vol. 41, Part 2, pp. 749–758.

[12] Jörg Eberspächer, Rüdiger Schollmeier, First and Second Generation of Peer-to-Peer Systems. In Peer-to-Peer Systems and Applications. Lecture Notes in Computer Science 3485, Springer, 2005.

[13] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. Proceedings of ACM SOSP, October 2003.

[14] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, X. Zhang, A Performance Study of BitTorrent-like Peer-to-Peer Systems. IEEE Journal on Selected Areas in Communications, Vol. 25, No. 1, pp. 155–169, 2007.

[15] M. Izal, G. Urvoy-Keller, E.W. Biersack, P.A. Felber, A. Al Hamra, L. Garces-Erice, Dissecting BitTorrent: Five Months in a Torrent's Lifetime. Proceedings of PAM 2004.

[16] A. Legout, G. Urvoy-Keller, P. Michiardi, Rarest First and Choke Algorithms Are Enough. Proceedings of IMC, 2006.

[17] A. Legout, N. Liogkas, E. Kohler, L. Zhang, Clustering and Sharing Incentives in BitTorrent. Performance Evaluation Review, Vol. 35, No. 1, pp. 301–312, 2007.

[18] T. Locher, P. Moor, S. Schmid, R. Wattenhofer, Free Riding in BitTorrent is Cheap. Proceedings of HotNets V, 2006.

[19] Bivas Mitra, Sujoy Ghose, Niloy Ganguly, Fernando Peruani, Stability analysis of peer-to-peer networks against churn. Paramana – Journal of Physics, Vol. 71, No. 2, pp. 263–273, 2008.

[20] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai, A Survey of Peer-to-Peer Network Simulators. Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK, 2006.

[21] Andy Oram (ed.), Peer-to-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly & Associates, 2001.

[22] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, A. Venkataramani, Do Incentives Build Robustness in BitTorrent? Proceedings of NSDI 2007.

[23] J.A. Pouwelsem, P. Garbacki, D.H.J. Epema, H.J. Sips, The BitTorrent P2P File-sharing system: Measurements and analysis. Lecture Notes in Computer Science, No. 3640, pp. 205–216, 2005.

[24] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. Epema, M. Reinders, M. R. van Steen, H. J. Sips, TRIBLER: a social-based peer-to-peer system. Concurrency and Computation, Vol. 20, No. 2, pp. 127–138, 2008.

[25] Dongyu Qiu, R. Srikant, Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks. Computer Communication Review, Vol. 34, No. 4, pp. 367–378, 2004.

[26] Riikka Susitaival, Samuli Aalto, Analyzing the file availability and download time in a P2P file sharing system. Proceedings of NGI 2007, pp. 88–95, 2007.

[27] Ye Tian, Di Wu, Kam-Wing Ng, Performance Analysis and Improvement for BitTorrent-like File Sharing Systems. Concurrency and Computation: Practice and Experience, Vol. 19, No. 13, pp. 1811–1835, 2007.

[28] Yao Yue, Chuang Lin, Zhangxi Tan, Analyzing the performance and fairness of BitTorrent-like networks using a general fluid model. Computer Communications, Vol. 29, No. 18, pp. 3946–3956, 2006.

Appendix A

Output given by the program in example runs of the simulator P2PSimulator with event information useful for debugging:

```
4998.326431598374 Assigned peer 10081 to download chunk 3 from peer 10078
4998.329012595653 Peer 10080 completed downloading chunk 3
4998.329012595653 Peer 10080 is idle
4998.329012595653 Assigned peer 10080 to download chunk 4 from peer 10079
4998.34670665701 Peer 10081 completed downloading chunk 3
4998.34670665701 Peer 10081 is idle
4998.34670665701 Assigned peer 10081 to download chunk 4 from peer 10078
4998.35351052179 Peer 10080 completed download in 0.12248963068486773
4998.35351052179 Peer 10080 is idle
4998.35351052179 Peer 10079 is idle
4998.366981715646 Peer 10081 completed download in 0.10137529317944427
4998.366981715646 Peer 10081 is idle
4998.366981715646 Peer 10078 is idle
4998.658262162024 Peer 10078 left.
4998.666776000954 Peer 10036 completed download in 15.227988123438081
4998.666776000954 Peer 10071 is idle
4998.907812092034 Peer 10042 completed downloading chunk 3
4998.907812092034 Assigned peer 10042 to download chunk 4 from peer 10036
4999.187989081445 Peer 10066 completed downloading chunk 2
4999.187989081445 Assigned peer 10066 to download chunk 3 from peer 10042
4999.479772813067 Peer 10082 arrived
4999.479772813067 Assigned peer 10082 to download chunk 0 from peer 10080
4999.496628862492 Peer 10075 completed downloading chunk 1
4999.496628862492 Peer 10075 is idle
4999.496628862492 Assigned peer 10075 to download chunk 2 from peer 10066
4999.504270739204 Peer 10082 completed downloading chunk 0
4999.504270739204 Peer 10082 is idle
4999.504270739204 Assigned peer 10082 to download chunk 1 from peer 10080
4999.528768665341 Peer 10082 completed downloading chunk 1
4999.528768665341 Peer 10082 is idle
4999.528768665341 Assigned peer 10082 to download chunk 2 from peer 10080
4999.553266591478 Peer 10082 completed downloading chunk 2
4999.553266591478 Peer 10082 is idle
4999.553266591478 Assigned peer 10082 to download chunk 3 from peer 10080
4999.555368011695 Peer 10074 left, adding 0.29770847809540707 of the file
being uploaded.
4999.555368011695 Assigned peer 10028 to download chunk 4 from peer 10081
4999.577764517615 Peer 10082 completed downloading chunk 3
4999.577764517615 Peer 10082 is idle
4999.577764517615 Assigned peer 10082 to download chunk 4 from peer 10080
```

```
4999.602262443752 Peer 10082 completed download in 0.12248963068486773
4999.602262443752 Peer 10082 is idle
4999.602262443752 Peer 10080 is idle
4999.765692016568 Peer 10083 arrived
4999.765692016568 Assigned peer 10083 to download chunk 0 from peer 10071
4999.80294467893 Peer 10040 completed downloading chunk 2
4999.80294467893 Peer 10040 is idle
4999.80294467893 Assigned peer 10040 to download chunk 3 from peer 10028
4999.80642024253 Peer 10083 completed downloading chunk 0
4999.80642024253 Peer 10083 is idle
4999.80642024253 Assigned peer 10083 to download chunk 1 from peer 10071
4999.847148468492 Peer 10083 completed downloading chunk 1
4999.847148468492 Peer 10083 is idle
4999.847148468492 Assigned peer 10083 to download chunk 2 from peer 10071
4999.8878766944545 Peer 10083 completed downloading chunk 2
4999.8878766944545 Peer 10083 is idle
4999.8878766944545 Assigned peer 10083 to download chunk 3 from peer 10071
4999.898332483127 Peer 10081 left, adding 0.075567028810258 of the file
being uploaded.
4999.898332483127 Assigned peer 10028 to download chunk 4 from peer 10079
4999.928604920417 Peer 10083 completed downloading chunk 3
4999.928604920417 Peer 10083 is idle
4999.928604920417 Assigned peer 10083 to download chunk 4 from peer 10071
4999.969333146379 Peer 10083 completed download in 0.20364112981042126
4999.969333146379 Peer 10083 is idle
4999.969333146379 Peer 10071 is idle
5000.017315625586 Peer 10084 arrived
```

Example results (with line breaks added) from the SimulatorFramework-class, which iterates through all of the distributions and selection policies, with desired values for λ, $\frac{1}{\gamma}$ and n:

```
Distribution: Exponential, Selection policy: Random_chunk, Chunks: 10,
Mean_time_between_arrivals: 0.5 (lambda=2.0), Mean_time_between_departures:
0.5, Average_download_time 1.7381858952428735 variance 1.3014907333204104
with 8079 completions in 5000.092536351473.

Distribution: Exponential, Selection policy: Random_chunk, Chunks: 5,
Mean_time_between_arrivals: 1.0 (lambda=1.0), Mean_time_between_departures:
1.0, Average_download_time 1.6319687929649993 variance 1.299337545492834
with 4054 completions in 5000.067164684866.

Distribution: Exponential, Selection policy: Random_chunk, Chunks: 10,
Mean_time_between_arrivals: 1.0 (lambda=1.0), Mean_time_between_departures:
1.0, Average_download_time 1.6281158252114996 variance 1.2928907856762337
with 3929 completions in 5000.041328573342.
```

Distribution: Exponential, Selection policy: Random_chunk, Chunks: 5, Mean_time_between_arrivals: 0.5 (lambda=2.0), Mean_time_between_departures: 1.0, Average_download_time 1.672543361726924 variance 1.3657836182168026 with 8056 completions in 5000.029978772758.

Distribution: Exponential, Selection policy: Random_chunk, Chunks: 10, Mean_time_between_arrivals: 0.5 (lambda=2.0), Mean_time_between_departures: 1.0, Average_download_time 1.6037439926288144 variance 1.247973433809371 with 7924 completions in 5000.000469471728.

Distribution: HyperExponential, Selection policy: Random_peer, Chunks: 5, Mean_time_between_arrivals: 1.0 (lambda=1.0), Mean_time_between_departures: 0.5, Average_download_time 4.221493365820459 variance 40.20971819577172 with 4000 completions in 5000.006677358926.

Distribution: HyperExponential, Selection policy: Random_peer, Chunks: 10, Mean_time_between_arrivals: 1.0 (lambda=1.0), Mean_time_between_departures: 0.5, Average_download_time 3.6986588176523587 variance 38.49169186351741 with 4037 completions in 5000.058791081908.

Distribution: HyperExponential, Selection policy: Random_peer, Chunks: 5, Mean_time_between_arrivals: 0.5 (lambda=2.0), Mean_time_between_departures: 0.5, Average_download_time 4.412434605528398 variance 39.24828348653216 with 8051 completions in 5000.240163276354.

Appendix B

The Java source code of the simulator.

P2PSimulator.java

```
package sim;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.PriorityQueue;

import model.*;
import strategy.*;

public class P2PSimulator {

   // Used for debugging, to keep track of what the peers are doing
   public static final boolean printEvents = false;
   // If true, the initial peer will never leave the system
   public static final boolean PERMANENTPEER = true;
   // Determines whether the permanent peer can change during the
   // simulation. If true, a departing seed can swap places with the
   // previously defined permanent peer, so that its transfer rate
   // will vary throughout a simulation.
   public static final boolean SWAPPERMANENTPEER = true;
   // Defines whether partially downloaded chunks will add to the
   // completion total. If set to false, any non-complete download
   // will result in the data being discarded (similarly to the
   // simulator used by Susitaival and Aalto).
   public static final boolean PARTIALDOWNLOADS = true;
   // Defines whether the initial seeder's transfer rate will simply
   // be the mean rate of the current simulation. This eliminates much
   // of the randomness when looking at the stability of the system
   public static final boolean USEMEANFORINITIALSEED = false;
   // Defines whether a peer will complete its current transfer
   // before leaving the network.
   public static final boolean COMPLETEUPLOAD = false;
   // Defines whether the completion time is the maximum of the
   // two peers' completion times (true) or defined only by the
   // downloader's transfer time (false).
   public static final boolean MAXCOMPLETIONTIME = true;
   // Defines whether the system will halt if the amount of seeds
   // grows beyond a certain point
   private static final boolean CHECKFORBALLOONING = true;
   // Will the system give priority to partially completed downloads
   // when peers are deciding who to upload/download from?
```

```
public static final boolean PRIORITIZEPARTIALDOWNLOADS = true;
public static final boolean MATLABRESULTS = true;
// Constants that define the mean time of events in the
// simulation, the length of the simulation and the amount of
// chunks
// in the file
public static final double MAXTIME = 40000; // t_max
public static final double COMPLETIONMEAN = 1; // 1/mu
// Determines the initial interval before which statistics will
// not be tracked in the system, used for giving time for the
// peer amounts to reach a balanced state
public static final double STATSTARTTIME = 1000;

public static final double STATMEAN = 0.1;

public static double arrivalMean = 0.1; // 1/lambda
public static double departureMean = 2; // 1/gamma
public static int fileSize = 1; // N

private int distributionNumber = 0;
private int policyNumber = 0;

public static enum EventType {
   Arrival, Completion, Departure, Stats
};

public static enum DistributionClass {
   Exponential, HyperExponential, Constant_rate, Erlang
};

public static enum SelectionPolicy {
   Random_peer, Rarest_first, Most_common_first, Random_chunk
};

public static DistributionClass distribution = DistributionClass.Exponential;
public static SelectionPolicy policy = SelectionPolicy.Random_chunk;
private static SelectionStrategy selectionStrategy;

// Variables for calculating certain statistics in the simulation
private double totalCompletionTime = 0;
private double totalCompletionTimeSquared = 0;
private int completions = 0;

// Data structures
private PeerList peers = new PeerList();
private PeerFactory peerFactory = new PeerFactory();
private PriorityQueue<Event> eventQueue = new PriorityQueue<Event>();
public static int[] systemChunks = new int[fileSize];
```

```
public static int[] downloadingChunks = new int[fileSize];
private Peer permanentPeer;

private Tracker tracker = new Tracker();
public static double simTime = 0;
private boolean stop = false;

public static DistributionClass getDistribution() {
   return distribution;
}

public static void setDistribution(DistributionClass distribution) {
   P2PSimulator.distribution = distribution;
}

private void setDistributionNumber(DistributionClass dist) {
   switch (dist) {
   case Exponential:
      distributionNumber = 1;
      break;
   case HyperExponential:
      distributionNumber = 2;
      break;
   case Constant_rate:
      distributionNumber = 3;
      break;
   case Erlang:
      distributionNumber = 4;
      break;
   }
}

public static SelectionPolicy getPolicy() {
   return policy;
}

public static void setPolicy(SelectionPolicy policy) {
   P2PSimulator.policy = policy;
   switch (policy) {
   case Random_peer:
      selectionStrategy = new RandomPeer();
      break;
   case Rarest_first:
      selectionStrategy = new RarestFirst();
      break;
   case Most_common_first:
      selectionStrategy = new MostCommonFirst();
      break;
```

```
   case Random_chunk:
      selectionStrategy = new RandomChunk();
      break;
   // default:
   // selectionStrategy = new RandomPeer();
   }
}

private void setPolicyNumber() {
   switch (policy) {
   case Random_peer:
      policyNumber = 1;
      break;
   case Rarest_first:
      policyNumber = 2;
      break;
   case Most_common_first:
      policyNumber = 3;
      break;
   case Random_chunk:
      policyNumber = 4;
      break;
   // default:
   // selectionStrategy = new RandomPeer();
   }
}

public static double getArrivalMean() {
   return arrivalMean;
}

public static void setArrivalMean(double meanTime) {
   arrivalMean = meanTime;
}

public static double getDepartureMean() {
   return departureMean;
}

public static void setDepartureMean(double departureMean) {
   P2PSimulator.departureMean = departureMean;
}

public static int getFileSize() {
   return fileSize;
}

public static void setFileSize(int size) {
```

```
   fileSize = size;
}

private Peer swapPermanentPeer(Peer newPermanentPeer) {
   Peer oldPermanentPeer = permanentPeer;
   permanentPeer = newPermanentPeer;
   return oldPermanentPeer;
}

// The addChunk and removeChunk functions are used to keep track of
// the amount of chunks in the system. They are used by the chunk
// selection policies.
private void removeChunk(int index) {
   systemChunks[index]--;
}

private void addChunk(int index) {
   systemChunks[index]++;
}

private void addChunkSet() {
   for (int i = 0; i < fileSize; i++) {
      addChunk(i);
   }
}

private void removeChunkSet() {
   for (int i = 0; i < fileSize; i++) {
      removeChunk(i);
      // Death occurs if there isn't a complete set of chunks
      // remaining in the system, and the simulation will halt
      if (systemChunks[i] == 0) {
         // System.out.println("Death in "+simTime);
         stop = true;
         return;
      }
   }

}

// The addDownloadingChunk and removeDownloadingChunk functions are
// used to keep track of the amount of chunks currently being
// downloaded by peers in the system. They are also used by the
// chunk selection policies (so that everyone won't start
// downloading the same chunk).
   private void addDownloadingChunk(int index) {
   downloadingChunks[index]++;
}
```

```
private void removeDownloadingChunk(int index) {
   downloadingChunks[index]--;
}

// Generates an arrival event for the next peer to enter the
// system.
private void generateArrival() {
   Peer newPeer = peerFactory.generatePeer();
   eventQueue.add(new Event(newPeer.getArrivalTime(), newPeer,
         EventType.Arrival));
}

// Called upon the arrival of a peer to the system.
private void doArrival(Event currentEvent) {
   generateArrival();
   tracker.addLeecher();
   Peer peer = currentEvent.getPeer();
   peers.add(peer);
   if (printEvents) {
      System.out.println(simTime + " Peer " + peer.getId()
            + " arrived");
   }
   assignPeer(peer);

   if (CHECKFORBALLOONING == true && tracker.getLeechers() > 1500) {
      stop = true;
   }
}

// Called upon the departure of a peer from the system.
private void doDeparture(Event currentEvent) {
   Peer departingPeer = currentEvent.getPeer();
   if (SWAPPERMANENTPEER) {
      double seederAmount = tracker.getSeeds();
      if (Math.random() < (1 / seederAmount)) {
         departingPeer = swapPermanentPeer(departingPeer);
      }
   }
   // If the departing peer was uploading, the peer that was
   // downloading needs to be told to stop (or the departing
   // peer needs to be told to wait, if the system is set to
   // not allow seeds to depart during a transfer).
   if (departingPeer.isUploading()) {
      Peer downloadPeer = departingPeer.getDownloadPeer();
      Event event = downloadPeer.getEvent();
      assert (event.getType().equals(EventType.Completion));
      if (COMPLETEUPLOAD) {
```

```
            eventQueue.remove(currentEvent);
            Event newDeparture = new Event(event.getTime(),
                  departingPeer, EventType.Departure);
            eventQueue.add(newDeparture);
         } else {
            // eventQueue.remove(currentEvent);
            eventQueue.remove(event);
            downloadPeer.setEvent(null);
            downloadPeer.setDownloading(false);
            removeDownloadingChunk(downloadPeer.getCurrentChunk());
            double completionAmount = 0;
            if (MAXCOMPLETIONTIME) {
               completionAmount = (simTime - downloadPeer
                     .getDownloadStartTime())
                     / (Math.max(departingPeer.getTransferTime(),
                           downloadPeer.getTransferTime()));
            } else {
               completionAmount = (simTime - downloadPeer
                     .getDownloadStartTime())
                     / downloadPeer.getTransferTime();
            }
            if (printEvents) {
               System.out.println(simTime + " Peer "
                     + departingPeer.getId() + " left, adding "
                     + completionAmount
                     + " of the file being uploaded.");
            }
            if (PARTIALDOWNLOADS) {
               downloadPeer.addChunk(downloadPeer.getCurrentChunk(),
                     completionAmount);
            }
            if (!PARTIALDOWNLOADS) {
               assert (downloadPeer.getHighestCompletion() == 0);
            }
            assignPeer(downloadPeer);
            tracker.removeSeed();
            peers.remove(departingPeer);
            removeChunkSet();
         }
      } else {
         if (printEvents) {
            System.out.println(simTime + " Peer "
                  + departingPeer.getId() + " left.");
         }

         tracker.removeSeed();
         peers.remove(departingPeer);
         removeChunkSet();
```

```
   }

}

// Called upon the successful completion of a chunk's download.
private void doCompletion(Event currentEvent) {

   Peer completedPeer = currentEvent.getPeer();
   Peer uploadPeer = completedPeer.getUploadPeer();
   double completionAmount = 0;
   if (!MAXCOMPLETIONTIME) {
      completionAmount = (simTime - completedPeer
            .getDownloadStartTime())
            / completedPeer.getTransferTime();
   } else {
      completionAmount = (simTime - completedPeer
            .getDownloadStartTime())
            / Math.max(completedPeer.getTransferTime(), uploadPeer
                  .getTransferTime());
   }

   completedPeer.setDownloading(false);
   uploadPeer.setUploading(false);
   addChunk(completedPeer.getCurrentChunk());
   removeDownloadingChunk(completedPeer.getCurrentChunk());
   // Test if the downloader has finished the entire file
   boolean completedDownload = completedPeer.addChunk();
   if (completedDownload) {
      tracker.completedLeech();
      if (printEvents) {
         System.out.println(simTime + " Peer "
               + completedPeer.getId() + " completed download in "
               + completedPeer.getDownloadTime());
      }
      eventQueue.add(new Event(simTime
            + peerFactory.generateDepartureTime(), completedPeer,
            EventType.Departure));
      if (simTime > STATSTARTTIME) {
         completions++;
         totalCompletionTime += completedPeer.getDownloadTime();
         totalCompletionTimeSquared += Math.pow(completedPeer
               .getDownloadTime(), 2);
      }
   } else if (printEvents) {
      System.out.println(simTime + " Peer "
            + completedPeer.getId()
            + " completed downloading chunk "
            + completedPeer.getCurrentChunk());
```

```
   }
   completedPeer.setEvent(null);
   if (!completedPeer.isUploading()) {
      requestLeecher(completedPeer);
   }
   requestLeecher(uploadPeer);
   if (!completedPeer.isDownloading() && !completedDownload) {
      assignPeer(completedPeer);
   }
}

// Get the time of the next statistic-collecting event
private double getStatInterval() {
   return -Math.log(Math.random()) * STATMEAN;
}

// List the amount of peers for the statistics collector
private void doStats(Event currentEvent) {
   if (simTime > STATSTARTTIME) {
      tracker.addEvent();
   }
   eventQueue.add(new Event(simTime + getStatInterval(), null,
         EventType.Stats));
}

// Assign an uploader for a peer, if available
private void assignPeer(Peer peer) {
   Event event = selectionStrategy.assignPeer(peer, peers);

   if (!(event == null)) {
      eventQueue.add(event);
      addDownloadingChunk(peer.getCurrentChunk());
   }
}

// Assign a downloader for a peer, if available
private void requestLeecher(Peer peer) {
   Event event = selectionStrategy.offerChunks(peer, peers);
   if (!(event == null)) {
      eventQueue.add(event);
      addDownloadingChunk(peer.getDownloadPeer().getCurrentChunk());
   }
}

public ArrayList<LogEvent> run() {
   systemChunks = new int[fileSize];
   downloadingChunks = new int[fileSize];
```

```
simTime = 0;
eventQueue.add(new Event(getStatInterval() / 2, null,
      EventType.Stats));
Peer initialSeed = peerFactory.generateInitialSeed();
peers.add(initialSeed);
permanentPeer = initialSeed;
if (!PERMANENTPEER) {
   eventQueue.add(new Event(simTime
         + peerFactory.generateDepartureTime() + 5
         * COMPLETIONMEAN, initialSeed, EventType.Departure));
}
setPolicy(policy);
setDistributionNumber(distribution);
setPolicyNumber();
addChunkSet();
generateArrival();
while (simTime < MAXTIME && !stop) {

   Iterator<Event> it = eventQueue.iterator();
   Event currentEvent = it.next();
   eventQueue.remove(currentEvent);
   simTime = currentEvent.getTime();
   switch (currentEvent.getType()) {
   case Arrival:
      doArrival(currentEvent);
      break;
   case Departure:
      doDeparture(currentEvent);
      break;
   case Completion:
      doCompletion(currentEvent);
      break;
   case Stats:
      doStats(currentEvent);
      break;
   }
}

// tracker.listResults();
double meanTransferTime = totalCompletionTime / completions;
double meanTransferTimeSquared = totalCompletionTimeSquared
      / completions;
double variance = meanTransferTimeSquared
      - Math.pow(meanTransferTime, 2);
if (!stop) {

   /*System.out.println(MATLABRESULTS ? distributionNumber + ", "
         + policyNumber + ", " + 1 / arrivalMean + ","
```

```
            + meanTransferTime + ";" : "Distribution: "
            + distribution + ", Selection policy: " + policy
            + ", Chunks: " + fileSize
            + ", Mean_time_between_arrivals: " + arrivalMean
            + " (lambda=" + 1 / arrivalMean
            + "), Mean_time_between_departures: " + departureMean
            + ", Average_download_time " + meanTransferTime
            + " variance " + variance + " with " + completions
            + " completions in " + simTime + ".");*/
      System.out.println(MATLABRESULTS ? distributionNumber + ", "
            + policyNumber + ", " + departureMean + ", "
            + meanTransferTime + ";" : "Distribution: "
            + distribution + ", Selection policy: " + policy
            + ", Chunks: " + fileSize
            + ", Mean_time_between_arrivals: " + arrivalMean
            + " (lambda=" + 1 / arrivalMean
            + "), Mean_time_between_departures: " + departureMean
            + ", Average_download_time " + meanTransferTime
            + " variance " + variance + " with " + completions
            + " completions in " + simTime + ".");

      } else {
         /*System.out.println(MATLABRESULTS ? distributionNumber + ", "
               + policyNumber + ", " + 1 / arrivalMean + ", 1000000;"
               : "Distribution: " + distribution
                     + ", Selection policy: " + policy + ", Chunks: "
                     + fileSize + ", Mean_time_between_arrivals: "
                     + arrivalMean + " (lambda=" + 1 / arrivalMean
                     + "), Mean_time_between_departures: "
                     + departureMean + ", Average_download_time "
                     + meanTransferTime + " variance " + variance
                     + " with " + completions
                     + " completions, STOPPED in " + simTime + ".");*/
         System.out.println(MATLABRESULTS ? distributionNumber + ", "
               + policyNumber + ", " + departureMean + ", 1000000;"
               : "Distribution: " + distribution
                     + ", Selection policy: " + policy + ", Chunks: "
                     + fileSize + ", Mean_time_between_arrivals: "
                     + arrivalMean + " (lambda=" + 1 / arrivalMean
                     + "), Mean_time_between_departures: "
                     + departureMean + ", Average_download_time "
                     + meanTransferTime + " variance " + variance
                     + " with " + completions + " completions in "
                     + simTime + ".");
      }
      return tracker.getResults();

   }
```

```
   public static void main(String[] args) {
      P2PSimulator runner = new P2PSimulator();
      runner.run();
   }
}
```

SimulatorFramework.java

```
package sim;

import java.util.ArrayList;
import java.util.Collections;

import model.*;

public class SimulatorFramework {
   private static final int RUNS = 1000;
   private static final int MAXCHUNKS = 10;
   private static final int MINCHUNKS = 10;
   private static final int CHUNKINTERVAL = 2;

   private static final double MAXARRIVAL = 0.1;
   private static final double MINARRIVAL = 0.1;
   private static final double ARRIVALINTERVAL = 0.1;

   private static final double MAXDEPARTURE = 7;
   private static final double MINDEPARTURE = 0;
   private static final double DEPARTUREINTERVAL = 0.2;

   private static final double MAXLAMBDA = 5;
   private static final double MINLAMBDA = 5;
   private static final double LAMBDAINTERVAL = 0.2;

   // INITIAL, AVERAGE, ARRIVAL, CHUNKS
   private static final String PURPOSE = "INITIAL";

   public static void main(String[] args) {
      int timeLength = (int) (P2PSimulator.MAXTIME
            / P2PSimulator.STATMEAN + 20);
      double[] totalSeeds = new double[timeLength];
      double[] totalLeechers = new double[timeLength];
      int[] count = new int[timeLength];
      if (PURPOSE.equals("INITIAL")) {
         for (int i = 0; i < RUNS; i++) {
            P2PSimulator runner = new P2PSimulator();
```

```
      ArrayList<LogEvent> runList = runner.run();
      Collections.sort(runList);
      for (LogEvent le : runList) {
         int thisTime = (int) (le.getTime() *
               (1 / sim.P2PSimulator.STATMEAN));
         count[thisTime]++;
         totalSeeds[thisTime] += le.getSeeds();
         totalLeechers[thisTime] += le.getLeechers();
      }
   }
}
if (PURPOSE.equals("CHUNKS") || PURPOSE.equals("ARRIVAL")) {
   for (P2PSimulator.DistributionClass distribution :
      P2PSimulator.DistributionClass.values()) {
      for (P2PSimulator.SelectionPolicy policy : P
            2PSimulator.SelectionPolicy.values()) {
         for (double l = MINDEPARTURE; l <= MAXDEPARTURE; l = l
               + DEPARTUREINTERVAL) {
            for (double k = MINARRIVAL; k <= MAXARRIVAL; k = k
                  + ARRIVALINTERVAL) {
               for (int j = MINCHUNKS; j <= MAXCHUNKS;
                j += CHUNKINTERVAL) {
                  for (int i = 0; i < RUNS; i++) {
                     P2PSimulator runner = new P2PSimulator();
                     P2PSimulator.setDistribution(distribution);
                     P2PSimulator.setPolicy(policy);
                     P2PSimulator.setFileSize(j);
                     P2PSimulator.setArrivalMean(k);
                     P2PSimulator.setDepartureMean(l);
                     ArrayList<LogEvent> runList = runner.run();
                     Collections.sort(runList);
                     for (LogEvent le : runList) {
                        int thisTime = (int) (le.getTime() *
                              (1 / sim.P2PSimulator.STATMEAN));
                        count[thisTime]++;
                        totalSeeds[thisTime] += le.getSeeds();
                        totalLeechers[thisTime] += le
                              .getLeechers();
                     }
                  }
               }
            }
         }
      }
   }
}
if (PURPOSE.equals("USELAMBDA")) {
   for (P2PSimulator.DistributionClass distribution :
```

```
            P2PSimulator.DistributionClass.values()) {
            for (P2PSimulator.SelectionPolicy policy :
               P2PSimulator.SelectionPolicy.values()) {
               for (double l = MINDEPARTURE; l <= MAXDEPARTURE; l = l
                     + DEPARTUREINTERVAL) {
                  for (double k = MINLAMBDA; k <= MAXLAMBDA; k = k
                        + LAMBDAINTERVAL) {
                     for (int j = MINCHUNKS; j <= MAXCHUNKS;
                      j += CHUNKINTERVAL) {
                        for (int i = 0; i < RUNS; i++) {
                           P2PSimulator runner = new P2PSimulator();
                           P2PSimulator.setDistribution(distribution);
                           P2PSimulator.setPolicy(policy);
                           P2PSimulator.setFileSize(j);
                           P2PSimulator.setArrivalMean(1 / k);
                           P2PSimulator.setDepartureMean(l);
                           ArrayList<LogEvent> runList = runner.run();
                           Collections.sort(runList);
                           for (LogEvent le : runList) {
                              int thisTime = (int) (le.getTime() *
                                    (1 / sim.P2PSimulator.STATMEAN));
                              count[thisTime]++;
                              totalSeeds[thisTime] += le.getSeeds();
                              totalLeechers[thisTime] += le
                                    .getLeechers();
                           }
                        }
                     }
                  }
               }
            }
         }
         if (PURPOSE.equals("INITIAL")) {
            for (int t = 0; t < timeLength - 19; t++) {
               double t2 = t;
               String returnString = (t2 / 10 + " " + totalSeeds[t]
                     / count[t] + " " + totalLeechers[t] / count[t])
                     .replace(",", ".");
               System.out.println(returnString);
            }
         }
      }
   }
```

Event.java

```
package model;

import sim.P2PSimulator.EventType;

public class Event implements Comparable<Event> {

private double time;
private Peer peer;

private EventType myType;

public Event(double time, Peer peer, EventType myType) {
this.time = time;
this.peer = peer;
this.myType=myType;
}

public Peer getPeer() {
return peer;
}

public double getTime() {
return time;
}

public EventType getType() {
return myType;
}

//@Override
public int compareTo(Event event) {
if (event.getTime() < this.getTime())
return 1;
if (event.getTime() > this.getTime())
return -1;
if (this.getType().equals(EventType.Departure)) {
return 1;
}

return 0;
}
}
```

File.java

```
package model;

//The File-class only needs to keep track of which pieces are downloaded
public class File {

private double[] chunk;

public File() {
chunk = new double[sim.P2PSimulator.fileSize];
}

public double[] getChunks() {
return chunk;
}

public void loadChunk(int chunkNumber) {
chunk[chunkNumber] = 1;
}

public void loadChunk(int chunkNumber, double amount) {
chunk[chunkNumber] = chunk[chunkNumber] + amount;
}

public boolean hasChunk(int chunkNumber) {
return chunk[chunkNumber]==1;
}

public double getChunkAmount(int chunkNumber) {
return chunk[chunkNumber];
}

public boolean isCompleted() {
for (double b : chunk) {
if (b!=1)
return false;
}
return true;
}

public double getHighestCompletion() {
double amount=0;
for (double piece : chunk) {
amount=(piece>amount) ? (piece<1.0) ? piece : amount : amount;
}
return amount;
}
```

```
}
```

LogEvent.java

```
package model;

public class LogEvent implements Comparable<LogEvent> {

   private double time;
   private int seeds;
   private int leechers;

   LogEvent(double time, int seeds, int leechers) {
      this.time = time;
      this.seeds = seeds;
      this.leechers = leechers;
   }

   public double getTime() {
      return this.time;
   }

   public int getSeeds() {
      return this.seeds;
   }

   public int getLeechers() {
      return this.leechers;
   }

   @Override
   public String toString() {
      return (time + " : " + seeds + " seeders, " + leechers + " leechers");
   }

   // @Override
   public int compareTo(LogEvent le) {
      if (le.getTime() > this.getTime())
         return -1;
      if (le.getTime() < this.getTime())
         return 1;
      return 0;
   }

}
```

Peer.java

```
package model;

import java.util.ArrayList;

public class Peer implements Comparable<Peer> {

   private double transferTime;
   private File file;
   private int id;
   private double arrivalTime;
   private double completedTime;
   private int currentChunk;
   private boolean downloading = false;
   private boolean uploading = false;
   private Peer downloadPeer = null;
   private Peer uploadPeer = null;
   private Event event;
   private double downloadStartTime;

   public Peer(int id, double time) {
      transferTime = time;
      arrivalTime = sim.P2PSimulator.simTime;
      file = new File();
      this.id = id;
   }

   // Start generic getters and setters

   public Event getEvent() {
      return event;
   }

   public void setEvent(Event event) {
      this.event = event;
   }

   public double getDownloadStartTime() {
      return downloadStartTime;
   }

   public void setDownloadStartTime(double downloadStartTime) {
      this.downloadStartTime = downloadStartTime;
   }

   public int getId() {
      return id;
```

```
}

public void setId(int id) {
   this.id = id;
}

public double getTransferTime() {
   return transferTime;
}

public void setTransferTime(double transferTime) {
   this.transferTime = transferTime;
}

public int getCurrentChunk() {
   return currentChunk;
}

public void setCurrentChunk(int currentChunk) {
   this.currentChunk = currentChunk;
}

public Peer getDownloadPeer() {
   return downloadPeer;
}

public void setDownloadPeer(Peer downloadPeer) {
   this.downloadPeer = downloadPeer;
}

public Peer getUploadPeer() {
   return uploadPeer;
}

public void setUploadPeer(Peer uploadPeer) {
   this.uploadPeer = uploadPeer;
}

public File getFile() {
   return file;
}

public boolean isDownloading() {
   return downloading;
}

public void setDownloading(boolean downloading) {
   this.downloading = downloading;
```

```
}

public boolean isUploading() {
   return uploading;
}

public void setUploading(boolean uploading) {
   this.uploading = uploading;
}

public boolean isCompleted() {
   return file.isCompleted();
}

public double getArrivalTime() {
   return arrivalTime;
}

public void setArrivalTime(double arrivalTime) {
   this.arrivalTime = arrivalTime;
}

public double getHighestCompletion() {
   return file.getHighestCompletion();
}

// End generic getters and setters

public double getCurrentChunkAmount() {
   return file.getChunkAmount(currentChunk);
}

public int[] getFileAsIntArray() {
   int[] chunks = new int[sim.P2PSimulator.fileSize];
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      if (file.hasChunk(i)) {
         chunks[i] = 1;
      }
   }
   return chunks;
}

public boolean hasChunks(File f) {
   boolean returnValue = false;
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      if (file.hasChunk(i) && !f.hasChunk(i)) {
         returnValue = true;
      }
```

```
   }
   return returnValue;
}

public boolean hasPartialChunks(File f) {
   boolean returnValue = false;
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      if (file.hasChunk(i)
            && (f.getChunkAmount(i) > 0 && f.getChunkAmount(i) < 1)) {
         returnValue = true;
      }
   }
   return returnValue;
}

public boolean needsChunks(File f) {
   boolean returnValue = false;
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      if (!file.hasChunk(i) && f.hasChunk(i)) {
         returnValue = true;
      }
   }
   return returnValue;
}

public int getRandomDesiredChunk(File f) {
   ArrayList<Integer> al = new ArrayList<Integer>();
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      if (file.hasChunk(i) && !f.hasChunk(i)) {
         al.add(i);
      }
   }
   assert (al.size() > 0);
   int i = al.get((int) (Math.random() * al.size()));
   return i;
}

/*
 * public boolean hasPartialChunks(File f) { boolean
 * returnValue=false; for (int i = 0; i <
 * sim.P2PSimulator.fileSize; i++) { if (file.getChunkAmount(i)!=0
 * && file.getChunkAmount(i)!=1 && f.hasChunk(i)) {
 * returnValue=true; } } return returnValue; }
 */

public int getRandomPartialChunk(File f) {
   ArrayList<Integer> al = new ArrayList<Integer>();
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
```

```
      if (f.getChunkAmount(i) > 0 && f.getChunkAmount(i) < 1
            && file.hasChunk(i)) {
         al.add(i);
      }
   }
   if (al.size() == 0) {
      return -1;
   }
   int i = al.get((int) (Math.random() * al.size()));
   // assert(this.hasChunk(i));
   // assert(f.getChunkAmount(i)>0 && f.getChunkAmount(i)<1);
   return i;
}

public boolean hasChunk(int chunk) {
   return file.hasChunk(chunk);
}

private boolean checkCompleted() {
   if (isCompleted()) {
      completedTime = sim.P2PSimulator.simTime;
      return true;
   }
   return false;
}

public boolean addChunk() {
   file.loadChunk(currentChunk);
   return checkCompleted();
}

public boolean addChunk(int chunk) {
   file.loadChunk(chunk);
   return checkCompleted();
}

public boolean addChunk(int chunk, double amount) {
   file.loadChunk(chunk, amount);
   return checkCompleted();
}

public double getDownloadTime() {
   return completedTime - arrivalTime;
}

// @Override
public int compareTo(Peer otherPeer) {
   if (otherPeer.getHighestCompletion() < this
```

```
            .getHighestCompletion())
         return 1;
      if (otherPeer.getHighestCompletion() > this
            .getHighestCompletion())
         return -1;
      return 0;
   }
}
```

PeerFactory.java

```
package model;

public class PeerFactory {

   private int idCount = 0;

   private double generateArrivalTime() {
      // use this for a constant mean arrival time.
      // double interval = sim.P2PSimulator.ARRIVALMEAN;
      // exponentially distributed arrival times
      double interval = -Math.log(Math.random())
            * sim.P2PSimulator.arrivalMean;

      return interval;
   }

   public double generateDepartureTime() {
      double interval = -Math.log(Math.random())
            * sim.P2PSimulator.departureMean;

      return interval;
   }

   public Peer generatePeer() {
      double transferTime = TransferTimeGenerator.getTransferTime();
      double arrivalTime = generateArrivalTime()
            + sim.P2PSimulator.simTime;
      Peer peer = new Peer(idCount, transferTime);
      peer.setArrivalTime(arrivalTime);
      idCount++;
      return peer;
   }

   public Peer generateInitialSeed() {
      Peer peer = generatePeer();
```

```
      // If the initial seed has the transfer time set as the mean
      if (sim.P2PSimulator.USEMEANFORINITIALSEED) {
         peer.setTransferTime(TransferTimeGenerator.meanChunkTime);
      }

      for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
         peer.addChunk(i);
      }

      return peer;

   }
}
```

PeerList.java

```
package model;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.PriorityQueue;
import java.util.Vector;

public class PeerList extends Vector<Peer> {

   private static final long serialVersionUID = 1L;

   public PeerList() {
      super();
   }

   public Peer getFastest(Peer peer) {
      File file = peer.getFile();
      Iterator<Peer> it = this.iterator();
      PriorityQueue<Peer> availableQueue = new PriorityQueue<Peer>();
      while (it.hasNext()) {
         Peer otherPeer = it.next();
         if (!otherPeer.isUploading() && otherPeer.hasChunks(file)) {
            availableQueue.add(otherPeer);
         }
      }
      if (!availableQueue.isEmpty()) {
         it = availableQueue.iterator();
         Peer uploadPeer = it.next();
         return uploadPeer;
      }
```

```
   return null;
}

public Peer getClosestRate(Peer peer) {
   File file = peer.getFile();
   Iterator<Peer> it = this.iterator();
   ArrayList<Peer> availableList = new ArrayList<Peer>();
   while (it.hasNext()) {
      Peer otherPeer = it.next();
      if (!otherPeer.isUploading() && otherPeer.hasChunks(file)) {
         availableList.add(otherPeer);
      }
   }
   if (!availableList.isEmpty()) {
      it = availableList.iterator();
      Peer uploadPeer = it.next();
      while (it.hasNext()) {
         Peer currentCandidate = it.next();
         if (Math.abs(peer.getTransferTime()
               - currentCandidate.getTransferTime()) < Math
               .abs(peer.getTransferTime()
                     - uploadPeer.getTransferTime())) {
            uploadPeer = currentCandidate;
         }
      }
      return uploadPeer;
   }
   return null;
}

public Peer getRandomUploader(Peer peer) {
   File file = peer.getFile();
   Iterator<Peer> it = this.iterator();
   ArrayList<Peer> availableList = new ArrayList<Peer>();
   while (it.hasNext()) {
      Peer otherPeer = it.next();
      if (!otherPeer.isUploading() && otherPeer.hasChunks(file)) {
         availableList.add(otherPeer);
      }
   }
   if (!availableList.isEmpty()) {
      Peer uploadPeer = availableList
            .get((int) (Math.random() * availableList.size()));
      return uploadPeer;
   }
   return null;
}
```

```
   public Peer getRandomDownloader(Peer peer) {
      File file = peer.getFile();
      Iterator<Peer> it = this.iterator();
      ArrayList<Peer> availableList = new ArrayList<Peer>();
      while (it.hasNext()) {
         Peer otherPeer = it.next();
         if (!otherPeer.isDownloading()
               && otherPeer.needsChunks(file)) {
            availableList.add(otherPeer);
         }
      }
      if (!availableList.isEmpty()) {
         Peer downloadPeer = availableList
               .get((int) (Math.random() * availableList.size()));
         return downloadPeer;
      }
      return null;
   }
}
```

Tracker.java

```
package model;

import java.util.ArrayList;
import java.util.Iterator;

/**
 * The Tracker-class simulates one function of an actual BitTorrent
 * tracker. It currently only keeps track of the amount of seeders and
 * leechers for the simulation.
 */
public class Tracker {

   private double oldTime = 0;
   private int seeds = 1; // Always has 1 seed at the start
   private int leechers = 0;
   private double seederTotal = 0;
   private double leecherTotal = 0;

   private ArrayList<LogEvent> log = new ArrayList<LogEvent>();

   public void addEvent() {
      double simTime = sim.P2PSimulator.simTime;
      seederTotal += (simTime - oldTime) * seeds;
      leecherTotal += (simTime - oldTime) * leechers;
```

```
   oldTime = simTime;
   log
         .add(new LogEvent(sim.P2PSimulator.simTime, seeds,
               leechers));
}

public int getSeeds() {
   return seeds;
}

public void setSeeds(int seeds) {
   this.seeds = seeds;
}

public int getLeechers() {
   return leechers;
}

public void setLeechers(int leechers) {
   this.leechers = leechers;
}

public void addLeecher() {
   leechers++;
   // addEvent();
}

public void removeSeed() {
   seeds--;
   // addEvent();
}

public void completedLeech() {
   leechers--;
   seeds++;
   // addEvent();
}

public ArrayList<LogEvent> getResults() {
   return log;
}

/**
 * Called at the end of the simulation, prints out a list of the
 * statistics for further analysis
 */
public void listResults() {
   /*
```

```
       * Iterator<LogEvent> it = log.iterator(); while (it.hasNext())
       * { LogEvent le = it.next(); System.out.println(le); }
       */
      double simTime = sim.P2PSimulator.simTime;
      seederTotal += (simTime - oldTime) * seeds;
      leecherTotal += (simTime - oldTime) * leechers;
      double seederAverage = seederTotal / simTime;
      double leecherAverage = leecherTotal / simTime;
      System.out.println("Mean : " + seederAverage + " seeders, "
            + leecherAverage + " leechers");
   }
}
```

TransferTimeGenerator.java

```
package model;

import sim.P2PSimulator;

/**
 * TransferTimeGenerator.java - Static class used for generating
 * random variables from different distribution classes
 *
 */
public class TransferTimeGenerator {

   public static double meanChunkTime;

   private static double generateExponentialVariable(double mean) {
      double transferTime = -Math.log(Math.random()) * mean;
      return transferTime;
   }

   private static double generateWeibullVariable(double scale,
         double shape) {
      double transferTime = scale
            * Math.pow(-Math.log(Math.random()), (1 / shape));
      return transferTime;
   }

   private static double generateHyperExponentialVariable(
         double[] weight, double[] mean) {
      double exponentialSelector = Math.random();
      int selectedDistribution = 0;
      while (weight[selectedDistribution] < exponentialSelector) {
         exponentialSelector -= weight[selectedDistribution];
```

```
      selectedDistribution++;
   }
   double transferTime =
    generateExponentialVariable(mean[selectedDistribution]);
   return transferTime;
}

private static double generateErlangVariable(double mean, double n) {
   double transferTime = 0;
   for (int i = 0; i < n; i++) {
      transferTime += generateExponentialVariable(mean);
   }
   return transferTime;

}

public static double getTransferTime() {
   double meanTotalTime = sim.P2PSimulator.COMPLETIONMEAN;
   meanChunkTime = meanTotalTime / sim.P2PSimulator.fileSize;

   double weibullK = 0.8;
   double weibullLambda = meanTotalTime / 1.1330030963
         / sim.P2PSimulator.fileSize;

   //double[] hyperExponentialWeights = { 0.5, 0.25, 0.25 };
   //double[] hyperExponentialMeans = { 0.5, 1.0, 2.0 };
   double[] hyperExponentialWeights = { 0.9, 0.1 };
   double[] hyperExponentialMeans = { 0.1, 9.1,  };
   for (int i = 0; i < hyperExponentialMeans.length; i++) {
      hyperExponentialMeans[i] = hyperExponentialMeans[i]
            * meanTotalTime / sim.P2PSimulator.fileSize;
   }

   double erlangN = 10;
   double erlangMean = meanChunkTime / erlangN;

   double transferTime = 0;
   switch (P2PSimulator.distribution) {
   case Exponential:
      transferTime = generateExponentialVariable(meanChunkTime);
      break;
   /*case Weibull:
      transferTime = generateWeibullVariable(weibullLambda,
            weibullK);
      break;*/
   case HyperExponential:
      transferTime = generateHyperExponentialVariable(
            hyperExponentialWeights, hyperExponentialMeans);
```

```
            break;
         case Erlang:
            transferTime = generateErlangVariable(erlangMean, erlangN);
            break;
         case Constant_rate:
            transferTime = meanChunkTime;
            break;
         }
         return transferTime;
      }
}
```

SelectionStrategy.java

```
package strategy;

import model.*;

public interface SelectionStrategy {

public Event assignPeer(Peer peer, PeerList peers);
public Event setPeers(Peer downloadPeer, Peer uploadPeer, int chunk);
public Event offerChunks(Peer peer, PeerList peers);
}
```

AbstractStrategy.java

```
package strategy;

import sim.P2PSimulator;

import model.*;

/**
 * AbstractStrategy.java An abstract peer selection policy that
 * defines the setPeers-method which is in common among all of the
 * actual selection policies.
 */
public abstract class AbstractStrategy implements SelectionStrategy {

   //@Override
   public Event assignPeer(Peer peer, PeerList peers) {
      return null;
   }
```

```
   //@Override
   public Event offerChunks(Peer peer, PeerList peers) {
      return null;
   }

//   @Override
   public Event setPeers(Peer downloadPeer, Peer uploadPeer, int chunk) {
      assert (!downloadPeer.isDownloading());
      assert (!uploadPeer.isUploading());
      assert (downloadPeer.getEvent()==null);
      assert (uploadPeer.hasChunk(chunk));
      assert (!downloadPeer.hasChunk(chunk));
      uploadPeer.setUploading(true);
      uploadPeer.setDownloadPeer(downloadPeer);
      downloadPeer.setDownloading(true);
      downloadPeer.setCurrentChunk(chunk);
      downloadPeer.setUploadPeer(uploadPeer);
      downloadPeer.setDownloadStartTime(sim.P2PSimulator.simTime);
      double transferTime = 0;
      if (!P2PSimulator.MAXCOMPLETIONTIME) {
         transferTime = (downloadPeer.getTransferTime())
               * (1 - downloadPeer.getCurrentChunkAmount());
         assert (transferTime>0);
      } else {
         transferTime = (Math.max(downloadPeer.getTransferTime(),
               uploadPeer.getTransferTime()))
               * (1 - downloadPeer.getCurrentChunkAmount());
         assert (transferTime>0);
      }
      Event ce = new Event(sim.P2PSimulator.simTime + transferTime,
            downloadPeer, sim.P2PSimulator.EventType.Completion);
      downloadPeer.setEvent(ce);
      if (sim.P2PSimulator.printEvents) {
         System.out.println(sim.P2PSimulator.simTime
               + " Assigned peer " + downloadPeer.getId()
               + " to download chunk " + chunk + " from peer "
               + uploadPeer.getId());
      }
      return ce;
   }
}
```

AbstractRarestStrategy.java

```
package strategy;

import java.util.ArrayList;

import model.Event;
import model.Peer;
import model.PeerList;

public abstract class AbstractRarestStrategy extends AbstractStrategy {

   public Event assignPeer(Peer peer, PeerList peers, boolean rarest,
         boolean prioritizePartialDownloads) {
      boolean[] availableChunk = new boolean[sim.P2PSimulator.fileSize];
      boolean anyAvailableChunk = false;
      int firstAvailableChunk = sim.P2PSimulator.fileSize + 1;
      for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
         availableChunk[i] = false;
      }

      boolean[] availablePriorityChunk =
       new boolean[sim.P2PSimulator.fileSize];
      boolean anyAvailablePriorityChunk = false;
      int firstAvailablePriorityChunk = sim.P2PSimulator.fileSize + 1;

      for (Peer otherPeer : peers) {
         if (!otherPeer.isUploading()) {
            for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
               if (!peer.hasChunk(i) && otherPeer.hasChunk(i)) {
                  availableChunk[i] = true;
                  firstAvailableChunk = Math.min(i,
                        firstAvailableChunk);
                  anyAvailableChunk = true;
                  if (peer.getFile().getChunkAmount(i) > 0
                        && prioritizePartialDownloads) {
                     availablePriorityChunk[i] = true;
                     anyAvailablePriorityChunk = true;
                     firstAvailablePriorityChunk = Math.min(i,
                           firstAvailablePriorityChunk);
                  }
               }
            }
         }
      }

      if (anyAvailableChunk) {
         int desiredChunk = 0;
         if (anyAvailablePriorityChunk && prioritizePartialDownloads) {
            desiredChunk = firstAvailablePriorityChunk;
```

```
   for (int i = firstAvailablePriorityChunk + 1; i <
        sim.P2PSimulator.fileSize; i++) {
      if (rarest) {
         if (availablePriorityChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] <
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      } else {
         if (availablePriorityChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] >
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      }
   }
} else {
   desiredChunk = firstAvailableChunk;
   for (int i = firstAvailableChunk + 1; i <
        sim.P2PSimulator.fileSize; i++) {
      if (rarest) {
         if (availableChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] <
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      } else {
         if (availableChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] >
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      }
   }
}

ArrayList<Peer> availableList = new ArrayList<Peer>();
for (Peer otherPeer : peers) {
   if (!otherPeer.isUploading()
         && otherPeer.hasChunk(desiredChunk)) {
      availableList.add(otherPeer);
   }
}

Peer uploadPeer = availableList
      .get((int) (Math.random() * availableList.size()));
Event ce = setPeers(peer, uploadPeer, desiredChunk);
return ce;
```

```
   } else {
      if (sim.P2PSimulator.printEvents) {
         System.out.println(sim.P2PSimulator.simTime + " Peer "
               + peer.getId() + " is idle");
      }
      return null;
   }
}

public Event offerChunks(Peer peer, PeerList peers,
      boolean rarest, boolean prioritizePartialDownloads) {

   boolean[] availableChunk = new boolean[sim.P2PSimulator.fileSize];
   boolean anyAvailableChunk = false;
   int firstAvailableChunk = sim.P2PSimulator.fileSize + 1;
   for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
      availableChunk[i] = false;
   }

   boolean[] availablePriorityChunk =
    new boolean[sim.P2PSimulator.fileSize];
   boolean anyAvailablePriorityChunk = false;
   int firstAvailablePriorityChunk = sim.P2PSimulator.fileSize + 1;

   for (Peer otherPeer : peers) {
      if (!otherPeer.isDownloading()) {
         for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
            if (peer.hasChunk(i) && !otherPeer.hasChunk(i)) {
               availableChunk[i] = true;
               firstAvailableChunk = Math.min(i,
                     firstAvailableChunk);
               anyAvailableChunk = true;
               if (otherPeer.getFile().getChunkAmount(i) > 0
                     && prioritizePartialDownloads) {
                  availablePriorityChunk[i] = true;
                  anyAvailablePriorityChunk = true;
                  firstAvailablePriorityChunk = Math.min(i,
                        firstAvailablePriorityChunk);
               }
            }
         }
      }
   }

   if (anyAvailableChunk) {
      int desiredChunk = 0;
      if (anyAvailablePriorityChunk && prioritizePartialDownloads) {
         desiredChunk = firstAvailablePriorityChunk;
```

```
   for (int i = firstAvailablePriorityChunk + 1; i <
        sim.P2PSimulator.fileSize; i++) {
      if (rarest) {
         if (availablePriorityChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] <
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      } else {
         if (availablePriorityChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] >
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      }
   }
} else {
   desiredChunk = firstAvailableChunk;
   for (int i = firstAvailableChunk + 1; i <
        sim.P2PSimulator.fileSize; i++) {
      if (rarest) {
         if (availableChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] <
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      } else {
         if (availableChunk[i] == true
               && (sim.P2PSimulator.systemChunks[i] >
                   sim.P2PSimulator.systemChunks[desiredChunk])) {
            desiredChunk = i;
         }
      }
   }
}

ArrayList<Peer> desiredList = new ArrayList<Peer>();
for (Peer otherPeer : peers) {
   if (!otherPeer.isDownloading()
         && !otherPeer.hasChunk(desiredChunk)) {
      desiredList.add(otherPeer);
   }
}

Peer downloadPeer = desiredList
      .get((int) (Math.random() * desiredList.size()));
Event ce = setPeers(downloadPeer, peer, desiredChunk);
return ce;
```

```
      } else {
         if (sim.P2PSimulator.printEvents) {
            System.out.println(sim.P2PSimulator.simTime + " Peer "
                  + peer.getId() + " is idle");
         }
         return null;
      }
   }
}
```

RarestFirst.java

```
package strategy;

import model.Event;
import model.Peer;
import model.PeerList;

public class RarestFirst extends AbstractRarestStrategy {

   @Override
   public Event assignPeer(Peer peer, PeerList peers) {
      return assignPeer(peer, peers, true,
            sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS);
   }

   @Override
   public Event offerChunks(Peer peer, PeerList peers) {
      return offerChunks(peer, peers, true,
            sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS);
   }
}
```

MostCommonFirst.java

```
package strategy;

import model.Event;
import model.Peer;
import model.PeerList;

public class MostCommonFirst extends AbstractRarestStrategy {

   @Override
   public Event assignPeer(Peer peer, PeerList peers) {
```

```
      return assignPeer(peer, peers, false,
            sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS);
   }

   @Override
   public Event offerChunks(Peer peer, PeerList peers) {
      return offerChunks(peer, peers, false,
            sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS);
   }
}
```

RandomChunk.java

```
package strategy;

import java.util.ArrayList;

import model.Event;
import model.Peer;
import model.PeerList;

public class RandomChunk extends AbstractStrategy {
   @Override
   public Event assignPeer(Peer peer, PeerList peers) {
      boolean prioritizePartialDownloads =
         sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS;
      ArrayList<Integer> availableChunk = new ArrayList<Integer>();
      ArrayList<Integer> availablePriorityChunk = new ArrayList<Integer>();

      for (Peer otherPeer : peers) {
         if (!otherPeer.isUploading()) {
            for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
               if (!peer.hasChunk(i) && otherPeer.hasChunk(i)) {
                  if (!availableChunk.contains(i)) {
                     availableChunk.add(i);
                  }
                  if (peer.getFile().getChunkAmount(i) > 0
                        && prioritizePartialDownloads) {
                     if (!availablePriorityChunk.contains(i)) {
                        availablePriorityChunk.add(i);
                     }
                  }
               }
            }
         }
      }
```

```
   if (availableChunk.size() > 0) {
      int desiredChunk = 0;
      if (availablePriorityChunk.size() > 0
            && prioritizePartialDownloads) {
         desiredChunk = availablePriorityChunk.get((int) (Math
               .random() * availablePriorityChunk.size()));

      } else {
         desiredChunk = availableChunk
               .get((int) (Math.random() * availableChunk.size()));
      }

      ArrayList<Peer> availableList = new ArrayList<Peer>();
      for (Peer otherPeer : peers) {
         if (!otherPeer.isUploading()
               && otherPeer.hasChunk(desiredChunk)) {
            availableList.add(otherPeer);
         }
      }

      Peer uploadPeer = availableList
            .get((int) (Math.random() * availableList.size()));
      Event ce = setPeers(peer, uploadPeer, desiredChunk);
      return ce;
   } else {
      if (sim.P2PSimulator.printEvents) {
         System.out.println(sim.P2PSimulator.simTime + " Peer "
               + peer.getId() + " is idle");
      }
      return null;
   }
}

@Override
public Event offerChunks(Peer peer, PeerList peers) {
   boolean prioritizePartialDownloads =
      sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS;
   ArrayList<Integer> availableChunk = new ArrayList<Integer>();
   ArrayList<Integer> availablePriorityChunk = new ArrayList<Integer>();

   for (Peer otherPeer : peers) {
      if (!otherPeer.isDownloading()) {
         for (int i = 0; i < sim.P2PSimulator.fileSize; i++) {
            if (peer.hasChunk(i) && !otherPeer.hasChunk(i)) {
               if (!availableChunk.contains(i)) {
                  availableChunk.add(i);
               }
               if (otherPeer.getFile().getChunkAmount(i) > 0
```

```
                        && prioritizePartialDownloads) {
                     if (!availablePriorityChunk.contains(i)) {
                        availablePriorityChunk.add(i);
                     }
                  }
               }
            }
         }
         if (availableChunk.size() > 0) {
            int desiredChunk = 0;
            if (availablePriorityChunk.size() > 0
                  && prioritizePartialDownloads) {
               desiredChunk = availablePriorityChunk.get((int) (Math
                     .random() * availablePriorityChunk.size()));

            } else {
               desiredChunk = availableChunk
                     .get((int) (Math.random() * availableChunk.size()));
            }

            ArrayList<Peer> desiredList = new ArrayList<Peer>();
            for (Peer otherPeer : peers) {
               if (!otherPeer.isDownloading()
                     && !otherPeer.hasChunk(desiredChunk)) {
                  desiredList.add(otherPeer);
               }
            }

            Peer downloadPeer = desiredList
                  .get((int) (Math.random() * desiredList.size()));
            Event ce = setPeers(downloadPeer, peer, desiredChunk);
            return ce;
         } else {
            if (sim.P2PSimulator.printEvents) {
               System.out.println(sim.P2PSimulator.simTime + " Peer "
                     + peer.getId() + " is idle");
            }
            return null;
         }
      }
   }
```

RandomPeer.java

```
package strategy;
```

```
import model.*;

/**
 * RandomPeer.java A peer selection policy which selects a random peer
 * to upload or download to
 */
public class RandomPeer extends AbstractStrategy {

   @Override
   public Event assignPeer(Peer peer, PeerList peers) {
      Peer uploadPeer = peers.getRandomUploader(peer);
      if (uploadPeer != null) {
         int whichChunk = sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS ?
          uploadPeer.getRandomPartialChunk(peer.getFile()) : -1;
         if (whichChunk == -1) {
            whichChunk = uploadPeer.getRandomDesiredChunk(peer
                  .getFile());
         }
         Event ce = setPeers(peer, uploadPeer, whichChunk);
         return ce;
      } else {
         if (sim.P2PSimulator.printEvents) {
            System.out.println(sim.P2PSimulator.simTime + " Peer "
                  + peer.getId() + " has idle download");
         }
         return null;
      }
   }

   @Override
   public Event offerChunks(Peer peer, PeerList peers) {
      Peer downloadPeer = peers.getRandomDownloader(peer);
      if (downloadPeer != null) {
         int whichChunk = sim.P2PSimulator.PRIORITIZEPARTIALDOWNLOADS ? peer
               .getRandomPartialChunk(downloadPeer.getFile())
               : -1;
         if (whichChunk == -1) {
            whichChunk = peer.getRandomDesiredChunk(downloadPeer
                  .getFile());
         }
         Event ce = setPeers(downloadPeer, peer, whichChunk);
         return ce;
      } else {
         if (sim.P2PSimulator.printEvents) {
            System.out.println(sim.P2PSimulator.simTime + " Peer "
                  + peer.getId() + " has idle upload");
         }
```

```
            return null;
        }
    }
}
```