MCFTP (Multicast File Transfer Protocol): Simulation and Comparison with BitTorrent

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Dominic Papritz
2010

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Abstract

Peer-to-Peer (P2P) file dissemination networks are large and popular fields of today's research. Application Layer Multicast is based on P2P technology and evolved to an operational alternative to IP Multicast, which is still not widely deployed. Therefore, we propose a new P2P file dissemination protocol – the Multicast File Transfer Protocol (MCFTP) – completely based on multicast, which can be run on both multicast types – native or overlay. Based on this new protocol, we describe a hybrid and a pure P2P version of the protocol. Furthermore, to evaluate the protocol, we implemented both versions on the network simulator ns-2. Additionally, we also implemented BitTorrent on the simulator to compare a unicast-based P2P file dissemination protocol with our multicast-based approach.

Acknowledgment

I would like to thank Prof. Dr. Torsten Braun who gave me the opportunity to work out my Diploma thesis in his research group Computer Networks and Distributed Systems. My special thanks go to Marc Brogle for supervising this work and providing the resources and material needed for my Diploma thesis.
Further thanks go to all people for their patience, encouragement and support, especially my family, my colleagues and my friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Peer-to-Peer (P2P) file dissemination networks are large and popular fields of today's research. Application Layer Multicast is also based on P2P technology and evolved to an operational alternative to IP Multicast, which is still not widely deployed. Therefore, we propose a new P2P file dissemination protocol – the Multicast File Transfer Protocol (MCFTP) – completely based on multicast. It supports native IP Multicast and Overlay Multicast. Based on this new protocol, we present a hybrid and a pure P2P version of the protocol. Furthermore, we implemented both protocol versions in the network simulator ns-2 to evaluate MCFTP. Additionally, we also implemented BitTorrent in this simulator in order to compare a unicast-based P2P file dissemination protocol with our multicast-based approach.
We simulate different flash-crowd scenarios based on two different topologies to evaluate both protocol versions and comparing them to BitTorrent. The first topology intents to simulate both protocol versions in a very heterogeneous Internet-like manner, using asymmetrical links (in terms of bandwidth) and native IP Multicast. The second topology is used to simulate primarily the behavior of MCFTP running on an overlay-like shared multicast forwarding tree using symmetrical links. This shared multicast forwarding tree is built once with optimal (QoS supporting) bandwidth assignments and once with bandwidth assignments that provoke packet drops. The scenarios range from tens to thousands of peers with file sizes of 50 and 100 MB. Some scenarios are also evaluated with file sizes of 200 and 500 MB.
We show that MCFTP always faster disseminate files among peers than BT. But, the protocol overhead of MCFTP, mainly the download protocol overhead, is very high and depends on the number of nodes and file size. Therefore, we also propose solutions to decrease the overall protocol overhead of MCFTP and to keep it independent of the number of nodes and file size. Additionally, we present an analytical estimation of the protocol bandwidth usage of MCFTP in relation to payload bandwidth for further protocol optimizations.

This diploma thesis is structured as follows.
Chapter 2 makes the reader familiar with the theoretical background used in this thesis. First, it gives an overview on P2P networks to understand and to classify proposed and existing P2P networks. Second, it looks at the different multicast services, which are needed by MCFTP. Then, it gives a brief overview onf the networks simulator ns-2. Finally, some related work is presented.

Then, Chapter 2.3 presents BitTorrent, which is used to compare MCFTP with. Chapter 3 presents the proposed Multicast File Transfer Protocol. This chapter about MCFTP gives an overview on the core protocol first. Then, the hybrid and pure P2P versions of the protocol are described in detail. For each protocol version, some additional aspects are discussed such as protocol traffic on the central protocol ”bus”, the problem of multicast address management and finally, some security and cooperation considerations.
The next two chapters are related to the simulations. Chapter 4 looks at the actual implementation of BitTorrent and the two MCFTP versions on the simulator. The simulation runs are presented in Chapter 5. First, the creation of the two different topologies is described that are used to create the network topologies. Then, the created networks are specified and some properties of them are described. Finally, the scenario runs are presented.
The simulation results and their interpretation are shown in Chapter 6. Finally, the conclusion of this thesis and an outlook to open questions and further research is presented in Chapter 7.

Chapter 2

Related Work

2.1 Overview

This Chapter makes the reader familiar with the theoretical background used in this thesis. First, it gives an introduction on Peer-to-Peer (P2P) networks. It makes the reader able to understand such networks. It allows him to classify the proposed Peer-to-Peer network in the big range of such networks. Second, BitTorrent is presented, that is used to compare the proposed P2P network. Then, it looks at the underlaying network infrastructure, which must be provided to implement and run the proposed Peer-to-Peer network. Then, it presents some file dissemination protocols that are based on multicast or P2P networks. And at last, it gives a brief overview on the simulator used in this thesis.

2.2 Peer-to-Peer Networks and File Sharing

2.2.1 Definition and Characterization of Peer-to-Peer Networks

The traditional network model is the client-server model, where a powerful resource provider – the server – exists with one or more clients that request resources through the network. In this model, the responsibilities of the server and of the clients are split disjunctively. In the P2P network model, each participating peer acts as server and as client. Therefore, a peer is not only a resource requester, but is also a resource provider. Each peer has the same responsibilities.
P2P networks are further divided into pure P2P networks and hybrid P2P networks. A pure P2P network consists of peers that are all equal in their behavior. Thus, a peer can be randomly chosen and removed from the network without degrading the network or the service the network provides. In a hybrid P2P network, one or more additional entities exist, which provide some special service to the peers. Mostly, this additional entities act as a server for those peers. In Fig. 2.1(a), the client-server model is shown. In Figures 2.1(b) and 2.1(c), the two P2P models are shown. A strict definition of P2P can be found in [8].

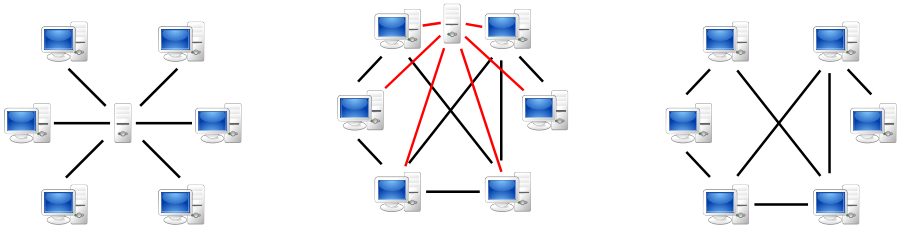
The following list gives an overview of some characteristics of P2P networks. Not all of them must be strictly met to speak of a P2P network. But it gives a better idea of the principles of P2P networks.

- peers form their own overlay network on top of a physical network
- peers act as servers and as clients, also called servants
- decentralization:
  - no central point of coordination
  - no central data storage
  - each peer knows only its neighborhood
  - no central point of failure
- self-organization: local interactions between peers results in a dynamic, complex and global system
- autonomy: peers act based on their local rules
- fault tolerance: peers can disconnect, messages may not arrive

P2P networks are not one of the latest inventions. One of the earliest P2P implementations was the distributed version of the Bellman-Ford shortest path algorithm [9] - a distance vector routing protocol - designed and used in the ARPANET in 1969. All routers formed a P2P network by only knowing their direct neighbor routers. When the routing table of a router changes, it sends its entire routing table to its neighbors. If the receiving router finds a new route or a better route, it updates its routing table and thus sends this information to all its neighbors.
Another example is the Usenet invented in 1979. Here, the news servers form a P2P network. Every news server knows some other news servers. A news server tries to synchronize its articles with the articles provided at the other servers. Therefore, the latest articles are copied from server to server. In the best case, every news server has all articles. But normal clients connect to the servers with the traditional client-server model.
With the growing of the Internet in space and performance and its wide popularity, P2P networks were reinvented as file sharing networks in 1999. The first popular file sharing network was Napster [10]. Napster has a single server, which maintains an index of all files in the system with references to peers, which provide these files. All peers must register at the server and provide a list of shared files. A requesting peer queries the server, which responds with the hits found in its collected index. Then, the peer directly connects to the peer providing the requested file and downloads it. This behavior is very similar to a client contacting a DNS server to get an IP address to a corresponding domain name and afterwards connecting to the webserver with the matching address. But the direct connections between equally behaving peers for transferring the data implies also the P2P model. Napster is a very distinct hybrid P2P network. The first pure P2P file sharing tool was Gnutella [10] released in 2000. It is described later in this Chapter. Since then, a lot of P2P file sharing tools have been released.

P2P networks are not only limited to file sharing. They can also provide other services. For instance, *Skype* [11] forms also a P2P network, which provides Internet telephony and instant messaging services. *Chord* [1] forms a P2P network, which provides a distributed-hash table.

P2P networks can also provide services at the network level instead of the application level. For instance, Overlay Multicast networks provide multicast services to end hosts.



(a) server-client based (b) hybrid P2P (c) pure P2P

Figure 2.1: The server-based, the hybrid P2P and the pure P2P based network models

Because P2P networks form their own abstract networks on top of the physical network, P2P networks model an Overlay Network. Almost all P2P networks provide resource location systems. Each peer maintains its resources that are often identified with a key. Thus, most P2P networks provide the ability to locate the peer, which maintains the requested resource.

2.2.2 Classification of P2P Networks

P2P networks can be classified on the basis of different properties [12, 13]:

Structure: In *unstructured P2P* networks, each peer only knows its own resources it provides. It does not know which resources its neighbors or generally any peer maintains. The placement of the resources is unrelated to the structure of the modeled P2P network. Therefore, in an unstructured P2P network, directed forwarding of a specific query to locate a resource is not possible. Queries are mostly flooded through the P2P network. The advantage of unstructured P2P networks is their simplicity and the independence of the peers, which makes these networks robust and fault tolerant. But the location of resources is very expensive and has not a deterministic behavior.
In *structured P2P* networks, each peer maintains information about resources managed by other peers. In most structured P2P networks, there exists a fixed mapping between the resources and the peers. Thus, the placement of the resources is related to the structure of the P2P network. Queries to locate a specific resource can be forwarded directed. The advantage of structured P2P networks is the efficient location of resources. But, the disadvantage is the overhead resulting of the additional information maintained by each peer. For instance, peers maintain routing tables, which must be kept up to date.

Hierarchy: In P2P networks with a *flat hierarchy*, there is no differences in the role and behavior of each peer. All peers are equal. These P2P networks are more difficult to destroy.

But they have a lower distribution of computation.
In P2P networks that are *hierarchical*, there exist some peers with distinct functionality. This makes the P2P network easier maintainable and the distribution of computation is higher, but with an additional overhead.

Coupling: A P2P network has a *loose coupling* if multiple groups could coexist. Such groups can be merged or split. In contrast, a P2P network with *tight coupling* can only have one global group to which every peer belongs to. Each peer has its static and logical identification and it has its fixed role with respect to the group. Thus, such networks can not be split or merged.

2.2.3 Examples of P2P Networks and File Sharing

In this Section, we present some P2P networks and their usage. This examples should make the classification of P2P networks as described in the previous Section more understandable. Table 2.1 gives an overview of the presented P2P networks and their classification.

	Structuring		**Hierarchy**		**Coupling**	
	unstructured	structured	flat	hierarchical	lose	tight
Gnutella v0.4	X		X		X	
Gnutella v0.6	X			X	X	
Chord		X	X			X

Table 2.1: Classification of some P2P networks

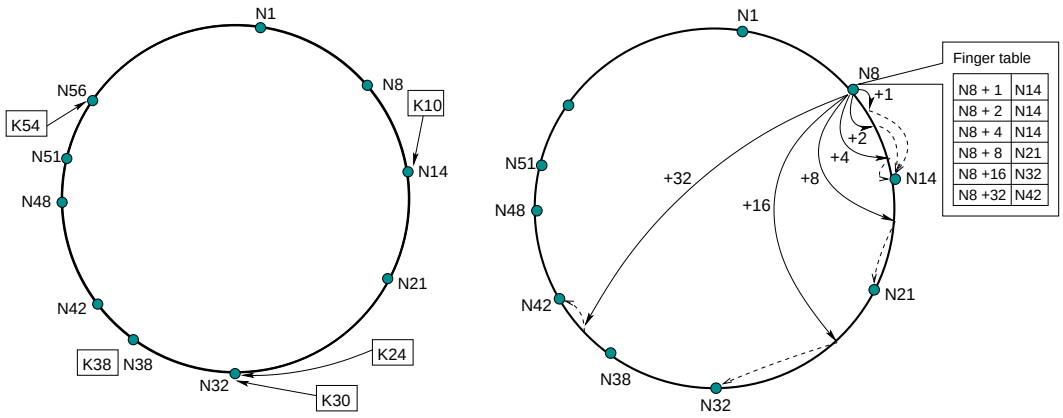
Gnutella v0.4 [14] is a pure P2P file sharing network. All peers act completely the same. Thus, Gnutella version 0.4 has a flat hierarchy. There are only 5 messages. Two request messages, Ping and Query, and two respond message, Pong and QueryHit. The fifth message is used to initiate a connection for the file transfer from the sender instead of the receiver. All request messages in the Gnutella network are simply flooded. Each message is identified by a Descriptor ID and has a Time-To-Live (TTL) field. An incoming request message is forwarded to all neighbors, except to the one that delivered it, if the TTL is greater than 0. And to prevent loops, the messages are only forwarded if the Descriptor ID is not already known. For all request messages, the reverse path is stored for a specific Descriptor ID by all forwarding peers. The response messages are identified with the Descriptor ID of the corresponding request messages and travel back the paths the request messages came from. Connections are made with direct TCP/IP connections between neighbors. The file transfer itself is done by a direct connection between the file requester and the file provider using HTTP.
A peer that wants to join the Gnutella network must know at least one already connected peer. It then tries to connect to this peer. Then, it must first send a *Ping* message to its new peer. With a Ping message, a peer wants to discover other peers. This Ping message is flooded through the network as previously described. All peers that receive a Ping message respond with a Pong message. A Pong message mainly contains the Port and IP address of the peer sending it. Now, the peer can make more connections to other peers discovered with the Ping-Pong mechanism.

A user initiated query is also flooded through the network containing a string as search criteria. But peers only respond with a QueryHit message, if they have a file or files, which match the search criteria. The QueryHit message contains connection information of the responding peer and one or more file information. Then the query initiating user can select his desired file from the list formed of all incoming QueryHit messages. With the information of the QueryHit message, the peer then makes a direct connection to the peer providing the file and downloads it.
The Gnutella v0.4 network is unstructured, because Query messages must be flooded through the network. A forwarding peer has no information about which files its neighbors have, and thus can also not decide to which peer it should forward the message. It is also not guaranteed that the query finds a file matching the search criteria, although a peer exists that has such a file. Gnutella is loosely coupled, because the network can be partitioned and afterwards merged without loosing its functionality.

Gnutella v0.6 [15] is a hierarchical P2P network, because it consists of leaf nodes and ultrapeers. Each node decides by itself if it becomes an ultrapeer or acts as a leaf node. The protocol is almost similar to Gnutella version 0.4. Leaf nodes only make a few connections to ultrapeers. Ultrapeers act as proxies in the Gnutella network for their leafs. Ultrapeers are connected to other ultrapeers and leaf nodes. The purpose of ultrapeers is to reduce the traffic generated by the flooded query messages. Thus, the Gnutella network scales better.
Each leaf node reports its shared resources to its ultrapeers. This is done by a list of hashes. The ultrapeers can now decide on the basis of the list of hashes received by each of their leafs, if they should forward a query to a specific leaf or not. A query is flooded among the ultrapeers like in version 0.4 among the simple nodes. But leafs receive only query messages they can also answer with a QueryHit messages. Leaf nodes do not forward query messages to other leaf nodes or ultrapeers.

Chord [1] is an implementation of a Distributed-Hash-Table (DHT). In DHTs, peer identifiers and resource identifiers are mapped into the same key space by using a hash function, which results in a *m*-bit identifier in the key space. With this mapping, each peer is responsible for a specific sub space of the whole key space.
In Chord, the key space is variably distributed among the participating peers. The peers are arranged in a circle according to the numerical value of their IDs in the key space. Each peer is responsible for the key range from its predecessor (without the predecessor itself) until its own key (including its key). Each peer keeps track of its successor. Figure 2.2(a) shows an example Chord ring. The key space uses 6 bit identifiers. Thus, a maximum number of 64 peers and keys, or more generally 2^m peers, can exist in this example.

If each peer would only know its successor, a look-up of a specific key would take $\Theta(N)$, where *N* is the number of peers in the ring. Thus, each peer maintains a finger-table. A finger table is a list of *m* peers, containing the node's key, IP address and port. On peer *n*, the entry at position *i* ($1 <= i <= m$) of the finger-table contains the peer, which is responsible for the key $n+2^{(i-1)} mod 2^m$. To complete the example, Fig. 2.2(b) shows the finger-table of peer 8.



(a) Example Chord ring with a 6-bit key space. Containing 10 nodes and 5 keys (source [1])

(b) Finger-table of node 8 (source [1])

Figure 2.2: Example Chord ring with a finger-table

With this finger-table, a look-up can be done in $\Theta(\log N)$. Each peer in the ring must periodically check its successor and rebuild the finger-table, because peers can join or leave. Also the keys must be replicated among other peers to have redundancy.
Chord's peers are highly coupled, because they must maintain their successors list and the finger-table. Keys of peers, which have left the overlay network must be reallocated. Chord is structured, because the identifier of a peer implicitly assigns responsibility for a specific part of the key space. A peer could also forward lookups to a peer closer to the searched key instead of an arbitrary peer. There is no hierarchy between peers in the network. Each peer has the same functionality and responsibilities.

2.3 BitTorrent

2.3.1 Introduction

BitTorrent [40, 41, 42] (BT) is a Peer-to-Peer (P2P) file sharing protocol, designed and implemented by Bram Cohen in 2001. Since then BT gained more and more popularity, and today it is the most used P2P file sharing protocol. The German company ipoque stated in its Internet traffic report of the year 2008 that BT is responsible for roughly 27-55% of all Internet traffic depending on the geographical region.
BitTorrent builds for each file a single swarm and each swarm is independent of the other ones. Thus, BT provides no indexing or lookup services for files. Such lookup services are mostly provided by websites. BT only provides a fast dissemination mechanism for a certain file among peers that have joined the appropriate swarm.
Since the first release of the BT client created by Cohen, a lot of other BT clients were released.

BitTorrent consist of three main parts: the metainfo file, at least one tracker per torrent, and the participating peers. A metainfo file, also called torrent file, is the main entry point for all users that want to download a specific file. A metainfo file contains all meta data about the file needed to successfully download it. Therefore, a new file is made available by a user creating a metainfo file of the file he wants to publish. In the process of creating a metainfo file, the new file is split into several pieces of equal size except the last piece, which can be smaller. For each such piece, a checksum is created and stored in the metainfo file along with the piece size and the file length.
The BT protocol works on these pieces of the file. This allows the protocol to distribute the whole file among the participating peers although most peers do not have a complete copy of the file. It also lets peers verify the correctness of what they already have, before they offer this to other peers. This makes the protocol robust against peers leaving the swarm and malicious peers.
The originating user must also provide an URL of a tracker, which is responsible for the new swarm. The user has several possibilities to chose a tracker. He could set up its own tracker or could use a public tracker. After creating the metainfo file, the publisher has to start his BitTorrent client as the initial seed of the new file and has to publish the metainfo file on a platform, where other users can find it. This could be done as previously mentioned on a website offering such services. The initial publishing user must then run his BT client until the whole file was uploaded at least once. The BT specification names peers, which have the entire file, as seeds. It names peers, which have not the complete file, as leechers.
Users, who want to download a specific file, must first search for the right metainfo file. After downloading the metainfo file and opening it in their preferred BT client, the client software connects to the tracker given in the metainfo file. The peer requests the tracker to answer with a list of other randomly chosen peers in the corresponding swarm. Then, the peer can connect to these other peers and the download can start. Each peer periodically requests new peers from the tracker to discover new or better peers.
BitTorrent tries to incorporate fairness, cooperation and download optimization by using a variant of tit-for-tat. Each peer uploads data to the peers, from which it can download the fastest.
Trackers were the single method for BT peers to discover other peers for a long time and it is still widely used. But newer BT clients can also directly exchange their list of known peers in addition to the tracker. But because the tracker is a single point of failure for a swarm, BT was adapted to allow more than one tracker per swarm. This was not only necessary to have backup trackers, it was also a good idea to implement load balancing on the trackers for big swarms. Another method to mitigate the tracker problem is the *trackerless* torrent feature. All peers, which have implemented the trackerless torrent feature, form a DHT based on Kademlia [43]. Hence, each peer is a tracker for certain swarms. Unfortunately, there are two trackerless implementations, which are not compatible with each other.

BitTorrent without the trackerless feature is a hybrid P2P network. With the trackerless feature it becomes a pure P2P network. There is no hierarchy between the participating peers. Every peer

has the same responsibilities and functionality as the other peers. BT also has loose coupling. Each peer connects to some peers randomly. Thus, with a really small probability and only for a short time there could be two separated clouds in one swarm. But most peers would nevertheless download their pieces or even not notice the problem. BT is not really structured on the swarm level because peers are randomly assigned. It is structured by the different swarms and structured between the peers directly connected. But, when a peer tries to download its last few pieces to complete the file, it can not be delegated to peers, which have those pieces. It can only wait until a peer it is already connected to has just downloaded one of the missing pieces or until the tracker proposes a peer, which has one or more of the missing pieces. The DHT used by the trackerless feature implies a tight coupled and structured P2P network, but only for the DHT, and not for the BT protocol itself.

2.3.2 The BitTorrent Peer Wire Protocol

The BitTorrent peer wire protocol is used between two connected peers. After receiving peer information from the tracker, a peer tries to make several connections to other peers by using the BT peer wire protocol. The protocol runs directly on TCP and uses some messages to signal state changes and to transfer file data.

BitTorrent uses blocks, also called sub-pieces, as the smallest unit transferred between peers. The use of blocks instead of whole pieces in the actual transfer of data makes it easier to saturate the connections and it is also necessary for peers with low bandwidth.

Each peer has two states associated to each connection to a remote peer: **choked** and **interested** states. The *choked* state describes whether the remote peer is willing to upload data to the peer or not. The *interested* states describes whether the remote peer is interested in some pieces, which it does not have yet. This implies that this two states have to be kept up-to-date.
Each new connection between two peers starts with a simple handshake by first sending a handshake message. After a successful handshake, different messages are exchanged until the connection gets closed. The different messages are described in Table 2.2.

2.3.3 Algorithms and Strategies

Choking Algorithm

The choking algorithm is the linchpin of the whole BitTorrent protocol. It implements a tit-for-tat strategy, which tries to enforce cooperation between peers. It also tries to make fair resource allocations, so that the download of one peer corresponds to its upload. The choking algorithm is a local optimization strategy, which tries to achieve Pareto efficiency. Pareto efficiency means that in a system of some participators, no two counterparties can make some changes, which leads to an overall improvement.
The choking algorithm is simple and straight forward. Each peer downloads from every remote peer, but each peer uploads only to the remote peers, from which it gets the best download rates. Peers reciprocate good download rates by uploading to the peers providing them. This should

Message	Description
handshake	is the first message sent and must be immediately sent after a new successful connection is established. The handshake message contains information about the protocol version used, the enabled features and the peer and file identification.
bitfield	is immediately sent once after the handshake. This message contains only a bitvector as payload, where each bit represents a piece. A peer initially signals its set of already downloaded pieces to its remote peer.
choke	signals to the remote peer that the originating per chokes it.
unchoke	signals to the remote peer that the originating peer has unchoked it.
interested	signals the interest in some pieces of the remote peer, but not which pieces these are.
not interested	signals the remote peer that it has no piece the originating peer is interested in.
have	is used to inform the connected peers that the originating peer has successfully downloaded and verified a new piece.
request	is used to request blocks of pieces by the remote peer. For each block one request message is sent. The peer itself and the remote peer must keep track of requested blocks. If choking by the remote peer occurs, all pending block requests are discarded.
piece	contains the actual transferred data and corresponds to the *request messages*. A piece message usually contains the whole requested block. Requested blocks are mostly not split into two or more piece messages.
cancel	cancels a previously requested block. This messages is typically used in the *endgame mode*, which will be described later.
keep-alive	is sent when no other message was sent by the peer in the last two minutes.

Table 2.2: Summary of the different messages used in the BitTorrent peer wire protocol

lead to several connections that transfer data in both directions. Uploading to a remote peer means that this remote peer is unchoked. Choking a remote peer means that it gets temporary no data or upload from the peer, but the peer can still download from the remote peer until it also gets choked by the remote peer. The connection between two such peers does not need to be renegotiated if choking or unchoking happens. The choking algorithm also ensures, that a peer always uploads data, if it is connected to peers that are interested in some pieces.

For an optimal resource allocation seeking Pareto efficiency, high bandwidth peers should serve other high bandwidth peers and low bandwidth peers should serve low bandwidth peers.

To get good TCP performance, each peer has always a fixed number (currently 4) of unchoked interested peers. Which peers to choke or unchoke is only decided once every ten seconds. This should be a long enough time period that TCP data transfers reach their maximum transfer rates and thus the remote peer can really decide from which peer it gets the best download rate. Also fibrillation (rapid change between choking and unchoking) is simply avoided with this behavior. To find other peers, which provide better download rates as the ones currently used, BT always selects a remote peer, which gets unchoked regardless of its download rate and interest state. Which peer is **optimistic unchoked** is selected every third choking/unchoking period.
When a peer has downloaded the complete file, it can not use the download rate to decide which peer to unchoke. Thus, it uses the upload rates to decide which peer to unchoke. The optimistic unchoke is still performed to search for remote peers, so that the peer can better saturate its upload.

Piece Selection

The selection of new pieces to download is an important task in BitTorrent. The overall performance of the system is heavily influence by it. Therefore, the BT protocol specifies some strategies, which must be implemented.

First of all, when blocks of a new piece are requested from a remote peer and some blocks are already downloaded, it is necessary to finish that piece before beginning with others. A peer can only offer whole pieces to other peers.

The next point is to choose which new piece to download next. It is important that the different pieces are well distributed among peers. If a peer selects pieces, which a lot of its remote peers have, then the chance is really small that a peer wants to download these pieces. Thus BT uses a **rarest first** strategy for piece selection. If a peer chooses the piece that is the rarest one when looking at its other remote peers, the chance is very high that a remote peer becomes interested in that piece.
Rarest first makes it also easier for a seed to push out the complete file for the first time. Because a peer will select other pieces than its remote peers do.
When a peer starts for the first time and thus has not yet any pieces, it must download a piece

as quickly as possible, so it can start to upload to other peers. Therefore, this first initial piece should be selected randomly instead of using the rarest first strategy. But immediately after finish downloading the first piece, the strategy switches to rarest first.
The rarest first strategy also implies that the most common pieces are downloaded last. But, it could happen that some blocks are requested from remote peers with a very low upload rate. To mitigate this problem, BT uses a strategy called **endgame mode**. If a peer has requested all its missing blocks, it also requests all those blocks from all other remote peers. But, to not waste bandwidth by downloading duplicates, a peer sends cancel messages to all remote peers it has also requested a specific block, after successfully downloaded this block from another peer.

Pipelining

It is necessary for the choking algorithm to always have saturated connections, because it makes its decisions according to the achieved download rates. If a peer makes one request and waits until it has finished downloading the requested block before it sends the next request, the download rate will never achieve the possible maximum. This is due to delays between the last received piece message and the arrival of the next requested block. Thus, it is necessary to always have some requests pending. This is accomplished by having always five block requests queued on a remote peer. When a peer gets unchoked by a remote peer, it starts with five block requests at once. If a block has arrived, the peer immediately requests the next block from the remote peer. This should guarantee that a remote peer can always send data without any interruption as long as it unchokes the peer.

2.4 Multicast

Multicast is a technology, which simultaneously delivers data to a specific group of receivers instead to only one receiver like unicast does or to all like broadcast does. The differences of broadcast, multicast and unicast are show in Fig. 2.3. Unicast is a one-to-one association between two hosts in a network. Each host has a unique address to identify itself. Multicast is a one-to-many association, or more generally, a many-to-many association between hosts. An address identifies a group of receiver hosts. Broadcast is also a many-to-many association between hosts. An address identifies a set of receiver hosts, but depending on the network topology. In multicast, a receiver host voluntarily belongs to a multicast address. In broadcast, a receiver host belongs to broadcast addresses depending on its subnet.

To perform multicast, a distribution tree has to be formed, with the source as the root and the receivers as the leafs. At the non-leaf vertexes, packets have to be replicated.

In the optimal case, each data packet traverses each link only once, and the path the packet travels from the source to the receivers is the shortest for each receiver. Thus, packets have to be duplicated within the network at the routers if needed. This two intuitive characteristics are called stress and stretch. Stress is a value that is defined for individual links or routers. It is counted how many times the same packet or a duplicate of it traverse a link or a router. On the

other hand, stretch is defined for each receiver of a multicast group. It is the ratio between the path length the packet actually travels and the shortest path to the receiver. Thus, an optimal multicast network has a stress and a stretch of one. There are other metrics to characterize multicast protocols, especially for Application Layer Multicast protocols [16].

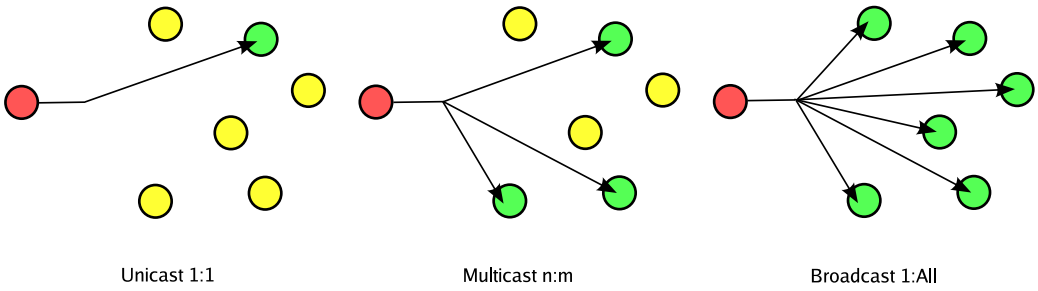


Figure 2.3: Different routing schemes

The file sharing protocol proposed in this thesis needs some extra requirements. The number of receivers in a multicast group can be very high. Therefore a multicast protocol should be able to handle a group size equal to the number of peers in the swarm. The join and the leave latency of a multicast protocol must be not too high. The join latency must be small enough, so that peers can receive the data. If the join latency is too high, a peer joins a specific group after the data has been sent. The leave latency must be small enough, so that peers do not waste their bandwidth receiving data not needed or in the worst case congesting their links.

2.4.1 IP Multicast

IP Multicast runs directly on the IP infrastructure in the network layer of the Internet protocol suite and some link layer protocols and technologies supporting IP Multicast natively, like Ethernet or FDDI. With IP Multicast, a host can send packets to a multicast address only once and without knowing any receiving host. The IP infrastructure, mainly the routers providing it, are responsible to deliver such packets to all joined receivers and replicate the packets when needed. Thus, the packets follow a multicast distribution tree, where a packet never crosses twice or more the same link or router, and the packets follow the shortest path to each receiver in most cases. Thus, IP Multicast has a stress and stretch of one. The key concepts and requirements for using IP Multicast are described in [17] and [18]. IP Multicast distinguish two different models. The traditional many-to-many association, which is called the Any-Source Multicast (ASM) [17]. The newer one-to-many association, which is called Source-Specific Multicast (SSM) [19]. The ASM model also includes one-to-many associations. But the main difference between ASM and SSM is that in ASM, a receiver wants to receive all data corresponding to a group independent of the number of sources. In SSM, a receiver wants to receive all data corresponding to a channel, where a channel is defined with a group address and a source unicast address.
In the next subsections, only some key aspects are explained in more detail.

IPv4 Multicast Addressing

IPv4 addresses are 32 bit identifiers. But for addressing multicast groups the addresses with a binary ”1110” prefix are reserved. This gives a maximum of 268’435’456 different multicast group addresses ranging from 224.0.0.0 to 239.255.255.255 (224/4), which seems a lot. But RFC3171 [20] and the corresponding assignments of the IANA limit the free dynamically usable multicast address ranges. The blocks within the entire multicast address range are the SDP/SAP block, the GLOB block for ASM, and the SSM block, which all can be of interest for a file sharing protocol. All other blocks are reserved, statically assigned or not globally routed, and thus are not interesting for a file sharing tool.
Within the SDP/SAP block, only the addresses from 224.2.128.0-224.2.255.255 can be freely and dynamically allocated and are valid in the global scope. But using such addresses requires to announce them with the Session Announcement Protocol (SAP) [21] using the Session Description Protocol (SDP) [22].
The SSM block is reserved for source-specific multicast, as stated in its name. SSM addresses include all addresses from 232.0.0.0 to 232.255.255.255 (232/8).
Table 2.3 summarizes the address range sizes of IPv4 and the above mentioned multicast address blocks and shows the number of addresses available within each block.

Address type	**Number of addresses within**
overall IPv4 address space	2^{32} (4 billions)
multicast address range	2^{28} (268 millions)
SDP/SAP block	2^{15} (32768)
overall GLOB block	2^{24} (16 millions)
GLOB block	255 per AS
SSM block	2^{24} (16 millions)

Table 2.3: IPv4 multicast addresses

IPv6 Multicast Addressing

IPv6 addresses are 128 bit identifiers. In IPv6, all addresses starting with FF00::/8 are multicast group addresses [23]. Thus, a IPv6 address starts with eight bits all set to 1. The next four bits are used as flags. For instance, the least significant bit of the four flag bits, the T flag, is set to 0 if the address is statically assigned or set to 1 if it is a dynamic or transient address. The next four bits represent the scope of the multicast address. For instance, ”0xE”, in base 16, represents the global scope or ”0x8” the organization-local scope. This replaces the need to assign blocks of IP ranges to different scopes or using some well defined TTL values for scoping. Thus, 2^{112} different group addresses are possible. But similar to IPv4, not all of them can be used for a file sharing tool. The group addresses must be globally routable and dynamically allocable. According to RFC 4291 [23], such addresses must have an address prefix of FF1E::/16, and thus have a group ID length of 112 bits.

But more restrictions apply. According to RFC 3307 [24], the group ID is restricted to the last 32 bits and for dynamic address assignment, the group ID must be between 8000:0 and FFFF:FFFF. This restriction comes from the fact that 128 bit group addresses are mapped to 32 bit link-layer address. With this assignment rules, statically or dynamically assigned multicast addresses can also be distinguished at the link-layer by looking at the most significant bit.
The IANA has reserved the range from FFEE::2:8000 to FFEE::2:FFFF for SAP dynamic assignments in the global scope, which works the same as for IPv4. It also provides the same number of usable addresses in the global scope. There exist more multicast group address formats like Unicast-Prefix-based IPv6 Multicast Addresses [25] or Embedded Rendezvous Point (RP) Addresses [26], which can be used for dynamic address allocation. The first mentioned address format simplifies the process of allocating dynamic multicast addresses. The second address format simplifies the inter-domain and intra-domain multicast configuration and deployment. A unicast-prefix-based address is similar to an address in the GLOB block in IPv4. The main difference between them is that in a GLOB block address, the AS number is used. In the unicast-prefix-based address, the 64bit network prefix is used. All unicast-prefix-based multicast addresses start with FF3E::/12, and all RP embedded multicast addresses start with FF7E::/16.
According to the IANA and RFC 3307 [24], SSM should use addresses in the range from FF3E::8000:0 to FF3E::FFFF:FFFF. SSM addresses are like unicast-prefix-based addresses, but with a 0 unicast-prefix.
The IPv6 enhanced address space enhances also the multicast address space. In contrast to IPv4, scoping in IPv6 and the identification of transient or statically assigned addresses is straightforward. Table 2.4 summarizes the address range sizes of IPv6 and its interesting address blocks available. It must be mentioned that the number of available addresses in this table must be halved, when taking RFC 3307 into account.

Address type	**Number of addresses within**
overall IPv6 address space	2^{128}
multicast address range	2^{120}
transient addresses per scope	2^{112}
SDP/SAP block per scope	2^{15} (32768)
all unicast-prefix-based addresses per scope	2^{96}
unicast-prefix-based addresses per scope and network prefix	2^{32} (4 billions)
SSM address range per scope	2^{32} (4 billions)

Table 2.4: IPv6 multicast addresses

The Internet Group Management Protocol or the Multicast Listener Protocol

The Internet Group Management Protocol, IGMP, and the Multicast Listener Discovery, MLD, are used between hosts and routers for reporting and querying interests in specific multicast groups. IGMP is used in IPv4 and MLD is used in IPv6. There exist different versions of both

protocols. The latest version of IGMP is version 3 [27] and the latest version of MLD is version 2 [28]. IGMPv2 and MLDv1 have almost the same functionality, while IGMPv3 is similar to MLDv2. The main difference between IGMPv2/MLDv1 and IGMPv3/MLDv2 is that the higher versions support source-specific multicast, which additionally needs the address of the source. The older versions only support any-source multicast. IGMP uses directly the Internet Protocol (IP) like ICMP does. MLD is a sub-protocol of ICMPv6. IGMP is not restricted to be only used by receivers. Multicast routers can also use IGMP or MLD to inform other multicast routers in their interest to a multicast group like a receiver does.

If a host wants to join a specific multicast group, it sends a report to its designated multicast router that it is interested to receive data of a certain multicast group. The multicast router then uses a multicast routing protocol, e.g. Protocol Independent Multicast or Distance Vector Multicast Routing Protocol, to route the multicast traffic from the source via remote multicast routers to its receiver. The multicast router then periodically sends queries to all its multicast capable hosts and looks if group members still exists. If not, it can cancel the multicast packet forwarding. To not waste unnecessary bandwidth, group members can also send a leave message, which initiates the multicast router to immediately send a group-specific query. If no host reports its constant interest, the forwarding can also be canceled.

Multicast Routing Protocols

There exist several multicast routing protocols. Most of them are used to deliver data packets belonging to multicast groups to their corresponding receivers within a domain or autonomous system (AS). And some of them are used to interconnect the domains or ASs. RFC 5110 [50] gives also a good overview on the Internet multicast routing architecture.

Intra-domain multicast routing protocols
All intra-domain protocols have in common that they build a forwarding tree from the source or a better placed node to the receivers and using reverse-path-forwarding (RPF) to prevent loops. In reverse-path-forwarding, a router looks at the source address instead of the destination address. If a packet enters a router, the router checks in its routing table if the incoming interface of that packet is also the interface to reach the source IP address. If it finds a route, the router forwards the packet, if not, it drops the packet.

The Distance Vector Multicast Routing Protocol (DVMRP) [51] was the earliest multicast protocol used. It is analogous to the unicast routing protocol RIP, but maintains its own routing table and message exchange. Thus, it needs a separate routing protocol for unicast routing and one for multicast routing. It builds source based shortest path trees to distribute the data to group members. Initially, it sets up a broadcast tree from each source, which is later pruned for branches that have no group members. DVMRP uses IGMP to exchange routing messages. It also uses RPF, to decide if a router should forward the packets or not. DVMRP additionally uses Truncated Reverse Path Broadcasting to truncate leaf subnets from the distribution tree. If a multicast router detects via IGMP that on its subnet no group member exists, it reports this to its parent multicast router, which then prunes the distribution

tree. Therefore DVMRP also maintains parent child relations in its routing table according to the different routing destinations. But to detect new group memberships, the data must be reflooded in constant intervals. Routers do not have a mechanism to actively join a multicast distribution tree, which is propagated towards the source. Thus, the reflooding interval determines the join-latency of a receiver. DVMRP has also a mechanism to tunnel multicast traffic through networks that do not support multicast. DVMRP was first used in the Mbone [29].

The Multicast Open Shortest Path First Protocol (MOSPF) [52] is an extension to the open shortest path first unicast routing protocol. MOSPF adds an new link-state-advertisement (LSA), which contains group memberships. These group-membership-LSA are produced from multicast routers, which directly have receivers on their connected subnets. Thus, each router can compute the source based shortest path tree from each source to each receiver, because all routers in a open shortest path first area maintains a database containing the status of the whole topology. But, this distribution tree computation can be a heavy task, if many sources and receivers exists or receivers have a high join-leave churn. MOSPF also implies the running of OSPF as unicast routing.

Protocol-Independent Multicast (PIM) is a multicast routing protocol family that uses the existing routing tables, regardless of how they were constructed. The two main modes of PIM are the Dense-Mode (PIM-DM) [53] and the Sparse-Mode (PIM-SM) [54]. These two modes have different distribution methodologies. PIM-DM tries to push the multicast data from the source towards it receivers. In PIM-SM, the receivers pull the data towards them.

PIM-DM initially floods the multicast traffic using reverse-path-forwarding based on the existing routing table. Thus, a source-rooted tree from each source to its joined members is constructed. Routers can then prune interfaces, where no receivers exist. A router can also send a prune message to the upstream router if all of its interfaces are pruned. For quicker rejoining such a pruned branch of a specific source-group pair, a router can send graft messages to its upstream router. Or a router can immediately send a join message to its upstream router to overwrite a prune message from another router on the same link. Like in DVMRP, PIM-DM periodically refloods the network to discover new group members.

PIM-SM is the most used multicast routing protocol. PIM-SM constructs a forwarding tree for each multicast group, but it uses a Rendezvous Point (RP), which acts as the root of the multicast distribution tree, called the RP tree or RPT. An AS can have one RP for all groups, different RPs for different multicast groups, or dynamically assign multicast groups to RPs by using bootstrap routers.
When a multicast source starts sending, the designated multicast router (DR) knows the according RP, either statically or through the bootstrap router. Then, the DR forwards the multicast data to the RP encapsulated in a PIM Register message by an unicast connection as presented in Fig. 2.4. If the new source sends more multicast data packets, the DR also encapsulate them in Register messages and forwards them to the RP. The RP decapsulates the multicast data from the PIM message and forwards it down the RPT, if receivers already exist. The RP then sends a

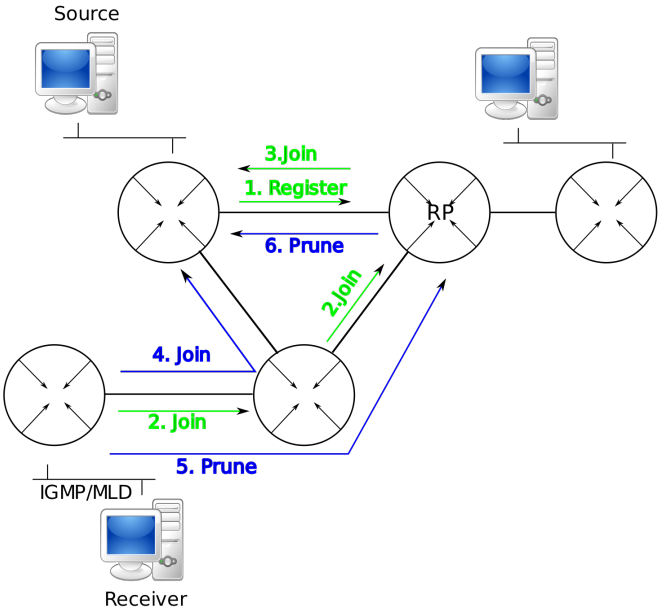


Figure 2.4: Creating the RPT and the SPT in PIM-SM

PIM Join message back towards the source. Thus, the RP joins the source-specific shortest path tree (SPT) from the source to itself. When it receives the packets from the source via the source specific tree, it sends a Register-Stop message to the DR of the source to prevent receiving the Register messages containing the duplicated multicast data which is not shown in Fig. 2.4. If no receivers are present, the RP sends directly a Register-Stop message back. If a new receiver arrives, via IGMP or MLD and a PIM join message, the RP sends a PIM join towards the source or sources.

A peer, which wants to newly join a multicast group, reports its interest using IGMP or MLD to its DR. The DR can then send a PIM join message towards the RP. This join message travels towards the RP and sets up the forwarding tree on each passing of the multicast routers. When the message arrives at the RP or at a router that is already a member of the requested RTP, the multicast traffic starts flowing downstream to the new receiver. Join messages towards the RP are periodically sent as long as receivers remain in the group. If a DR detects that no receivers exist in its subnet, it sends a prune message towards the RP for that multicast group. Thus, the branch is pruned at the RP or at the first router, which still needs the packet flow.
For optimizing latency or bandwidth efficiency, a DR can initiate a transfer from the shared tree (RPT) to a source-specific shortest path tree (SPT). For this tree change, the DR sends a join message towards the source for the specific group. This instantiates the SPT until the source is reached or until a router is reached that already is a member of the SPT. Therefore the packet flow starts propagating directly from the source to the DR and its receivers. The DR now receives two duplicated data flows. Thus, when the first traffic from the SPT arrives at the DR,

it sends a Prune message towards the RP to stop the data flow of this specific source and drops the packets of this source arriving via the RTP. If more than one source exists, this switching from the shared-tree to the source-specific tree must be done for each source. If a new source starts sending packets to a specific group, these packets are received via the RPT.
To perform source-specific multicast with PIM-SM, a DR joins directly the SPT and not the RPT.

The dense mode multicast protocols like DVMRP and PIM-DM have the advantage that the control overhead is small. But, with their flooding and pruning mechanisms, bandwidth is not efficiently handled. It is suitable for domains with a low number of different multicast groups and a high number of receivers for each group. Sparse-Mode multicast protocols like PIM-SM have the advantage that bandwidth is efficiently handled. It has no flooding and pruning mechanisms, receivers explicitly join a group. A disadvantage is that the distribution tree is not always optimal, and thus delays can also be higher. The concentrated traffic to and from the RPs is also a disadvantage. But it scales well with a lot of different groups.

Inter-domain multicast routing protocols
Today there exists only one protocol suite usable for any-source inter-domain multicast routing. It consists of PIM-SM for intra-domain routing, and of the Multiprotocol Border Gateway Protocol (MBGP) [55] for the inter-domain route advertisings. In IPv4, also the Multicast Source Discovery Protocol (MSDP) [56] for advertising new sources in a domain to other domains is part of that suite. In IPv6, the RP can be directly accessed when using RP-embedded addresses. Therefore, MSDP is not used because the RP is know from the multicast address itself. PIM-SM works as described in the previous section. With MBGP, the multicast routing table must not be the same as used for unicast routing. Thus, the multicast routing topology can be different from the unicast topology. Therefore, it can follow different policies and rules. The multicast routing information in MBGP is used for RPF checks instead of destination lookups. MSDP must be run on all RPs, which want to serve globally routed multicast sessions. Such MSDP enabled routers are connected to some other MSDP enabled routers in other domains. When a RP in a PIM-SM domain first learns of a new sender, it sends a ”Source-Active” message to all its MSDP peers. This message contains the address of the source, the multicast group address, and the IP address of the RP. Such messages are cached on the receiving MSDP RPs and are forwarded away from the originating RP to the other MSDP peers using RPF checking. If a RP from another domain receives such a source-active message and has already group members within its domain, the RP sends a join message towards the data source, which sets up a branch of the source-specific tree to this domain. The RP forwards the received multicast data itself down the shared-tree to the receivers. If a RP in a domain receives a PIM Join message for a new group, it must look up its source-active cache and sends a join message for each match towards the source.

Deployment Status

IP Multicast started with the experimental multicast backbone (Mbone) [29] in 1992. The idea was to connect IP Multicast enabled network islands with IPv4 unicast tunnels using the functionality of DVMRP. It started with 40 network islands and has quickly grown to several thousands. But Mbone became obsolete, because the normal backbone routers became multicast

routing capable. Also the pending switch to IPv6, which includes IP Multicast, should make multicast usable for all Internet participants. But today, most end-users connected with DSL or cablemodem still can not profit from multicast, because providers do not enable it.
Most bigger ISPs provide IP Multicast for their commercial customers, and most universities and research laboratories are also IP Multicast capable and peering with each other. But the lack of good admission and accounting mechanisms for IP Multicast still frightens some ISPs.
Figure 2.5 shows the IP Multicast capable AS in relation to all ASs as seen from AS 16517. This shows that IP Multicast is still not widely deployed.

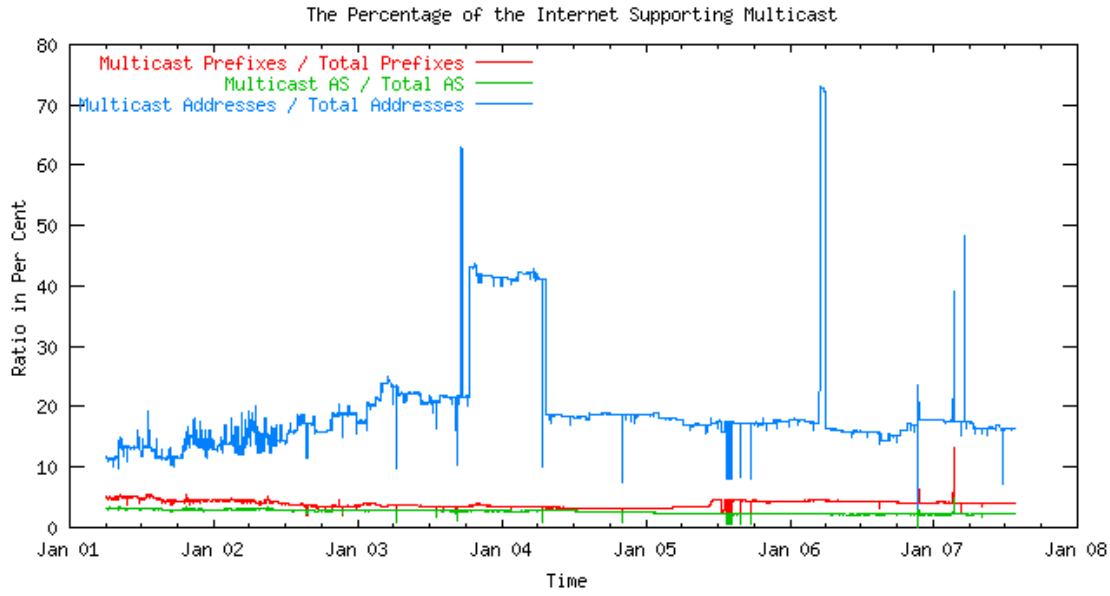


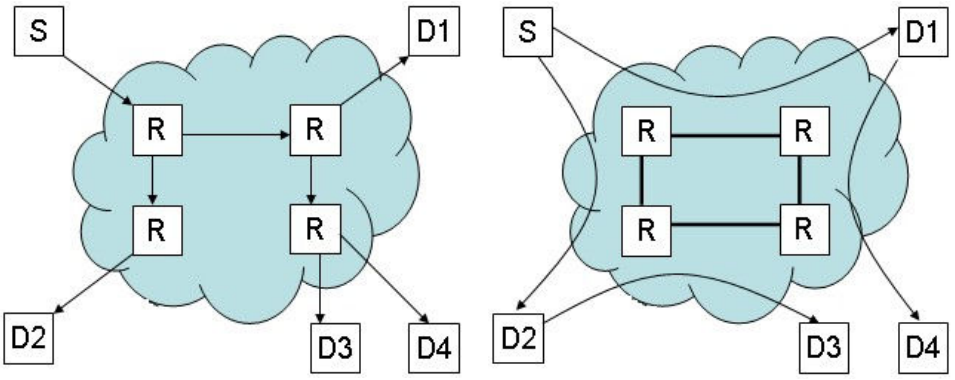
Figure 2.5: Deployment status of MBGP routers within the Internet (source [2])

2.4.2 Application Layer Multicast - Overlay Multicast

Overview

Application Layer Multicast (ALM), also called overlay multicast, became popular in the last few years, because IP Multicast is still not widely deployed. Thus, with the ongoing research on P2P and overlay networks, a lot of ALM networks were invented. The main difference between IP Multicast and ALM is shown in Fig. 2.6. Figure 2.6(a) shows IP Multicast. The packets are replicated at routers when needed. No packet is traversing a link more than once and each packet follows the shortest path from the sender to the receiver. Thus, link stress and stretch are 1.
In Fig. 2.6(b), an overlay network is used, which performs Application Layer Multicasting. Now, the packets are replicated and forwarded on the end-hosts. The end-hosts use unicast connections between them according to the ALM network topology. Thus, the routers only

see normal unicast connections. Because the replication and forwarding of packets is now performed on the end-host and not on the routers, some links used have a link stress higher than 1. For instance, the links of D1 and D2 to their corresponding routers must transport the same multicast packet twice. Also, a link stretch of one can not be guaranteed with ALM. For instance, D1 receives a packet via the same path like the packet would traverse with IP Multicast, therefore the link stretch is 1. But D3 receives the packet via D2 and its nearest router, which lays not in the shortest path according to the IP Multicast example. Thus, the link stretch must be higher than 1. The advantage of ALM is that it needs no additional infrastructure, and thus works on today's Internet. The drawbacks of ALM compared to IP Multicast are higher delays, smaller bandwidth and higher overhead. A closer look on ALM is presented in [3].
There are also hybrid approaches, where several IP Multicast islands are connected together with an overlay multicast protocol. Each IP Multicast island has its own dedicated proxy server. The proxy-servers participate in overlay multicast networks to cross the multicast incapable parts of the Internet. This mechanism is transparent for the end-hosts. They only use IP Multicast. But, because this proxy-based multicast approach also needs additional infrastructure deployed by the ISPs, it is only mentioned for completeness.



(a) IP Multicast (b) Application Layer Multicast

Figure 2.6: IP Multicast and Application Layer Multicast (source [3])

An Example of ALM

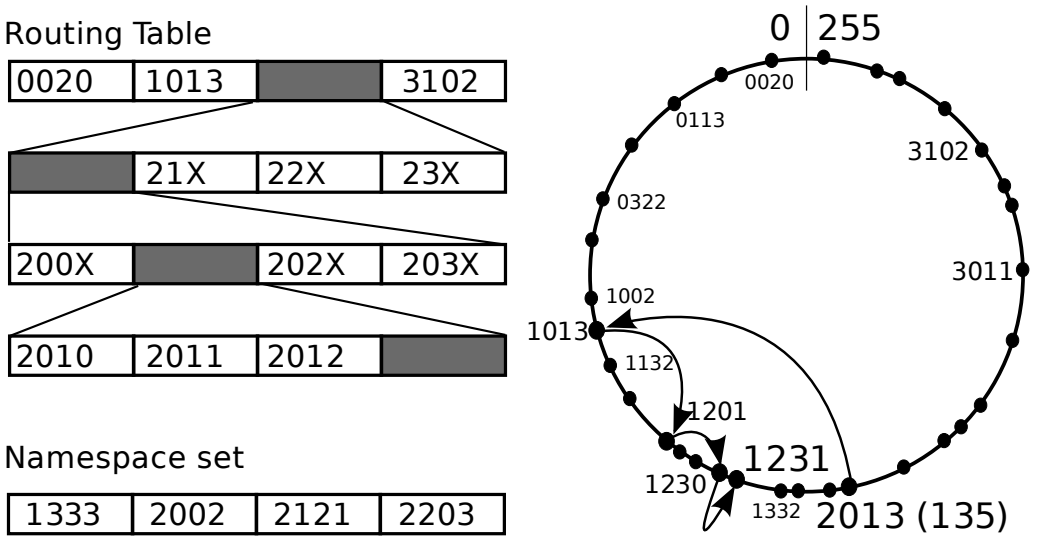
We present Scribe [30] as an example of ALM. Scribe can handle large group sizes and multiple sources, which is also a requirement in this thesis. Scribe supports one or more groups per overlay network.

Scribe

Scribe is based on Pastry [31]. Pastry is an overlay network, which implements a DHT, or more generally, an object location and routing overlay network similar to Chord. It is structured,

has a flat hierarchy and the peers are highly coupled. Like Chord, Pastry also uses a circular namespace ranging from 0 to $2^{128} - 1$.
First, we briefly explain Pastry and then we show how Scribe is implemented on top of Pastry. Like all structured P2P networks Pastry maps object identifiers to node identifiers. In Pastry, the object identifiers can be longer than the node identifiers, but must be at least as long as the node identifiers. But, the object location and routing capabilities of Pastry only work on the 128 most significant bits of the object identifier, or more generally, on the chosen 2^B key-/node-identifier-space. But, here we assume that the object identifiers (objectId) have the same length as the node identifiers (nodeId). Like in other structured P2P networks, Pastry uses the SHA-1 hashing algorithm to derive keys that are uniformly distributed and have a good collision resistance.
Pastry stores an object at the node, which has the nodeId numerically closest to the objectId. This also implies that any node can route a message with a given key to the node, which is responsible for that key, the node with the nodeId numerically closest to the key. Pastry sub-divides each nodeId of *B* bits into a sequence of levels, where each level is made up by *b* contiguous bits. The parameter b has a typical value of 3 or 4. This splits each key into B/b levels, where each level contains 2^b different domains. Then, at each hop, a message is routed to a node, which has a least one more level equal to the key of the message as the current forwarding hop. This is repeated until the numerically closest node is found. If no such node can be found in the routing table, the routing algorithm chooses a node, which is numerically closer to the message key as the nodeId of the current forwarding hop.
Therefore, each node maintains a routing table, a neighborhood set and a namespace set. For routing messages, only the routing table and the namespace set are used. Thus, we only describe these two in more detail. The neighborhood set is used for self-configuration and adaptation of Pastry, especially when new nodes arrive or existing nodes leave the overlay network.
The routing table at each node contains at each level *l*, $2^b - 1$ nodeId - IP address pairs, which have the same nodeId prefix as the local node up to level $l - 1$, but a different domain at level *l*. Figure 2.7(a) shows an example routing table for a node with a nodeId of 2013 in base 4, *B* is set to 8 and *b* is set to 2. The *X* stands for an arbitrary suffix. The nodes listed in the routing table could be chosen randomly for a given level and domain, but Pastry demands to always chose the closest node, in terms of network proximity e.g round-trip time, as the representative for a given domain. In the example, routing table each domain has set a nodeId, but this can not be achieved in real scenarios, because the number of existing nodes in a Pastry network is always much smaller as the maximum possible. Therefore, each routing table has some missing entries, mainly in the domains of higher levels (bottom of the routing table).
The namespace set *L* contains an even number of nodeId - IP address pairs that are numerically closest to the current nodeId and centered around it. The namespace set contains 2^b entries. It is also used to store replicas of a new inserted object on a subset of the namespace set, which leads to more reliability and robustness.

As an example, Fig. 2.7(b) shows a message with the key 1233 that is routed from node 2013 through the Pastry network to the node, which has the nodeId numerically closest to the message key. First, node 2013 looks in its namespace set if it already knows a node with the nodeId closest to the message key. Because the namespace set does not contain such a node, it looks in its routing table an looks for a node with a nodeId, which has at least the correct domain at level



(a) Routing table and namespace set of node 2013, IP addresses are omitted

(b) Example of routing a message with key 1233 from node 2013

Figure 2.7: Scribe: example of a routing table and a message forwarding

0. Therefore, the node forwards the message to node 1013. Node 1013 has no adequate node in its namespace set, and thus forwards the message to node 1201. Again node 1201 forwards the message to node 1230, which knows node 1231 and node 1332 from its namespace set. Node 1230 can now decide that node 1231 has the numerically closest nodeId to the message's key 1233 and that it must be the end point of the message route.

Scribe, built on top of Pastry, uses the routing mechanism of Pastry to implement multicast distribution trees for each existing group to the joined receivers. Scribe implements multicast trees like PIM-SM and uses a scheme similar to reverse path forwarding. The scribe API is very simple and it also supports some sort of access control features, which are not described here. The Scribe API only provides 4 methods:
A *create(groupId)* method, which creates a multicast group. A groupId is the SHA-1 hash value of a textual group name and is routable by Pastry. Before any group can be used, it must be created with this method. This method sets up the node, which is numerically closest to the groupId as the Rendezvous Point (RP).
A *join(groupId)* method, which sets up a branch of the multicast tree of the specified groupId to the originating node. It works similar as PIM-SM, but it uses Pastry to route the JOIN message to the RP.
A *leave(groupId)* method, which lets the invoking node leave the specified group. This works also similar to PIM-SM. But, Scribe has to respect that the originating node could be a forwarding node for other group members.
A *multicast(groupId, message)* method, which sends messages down the specified multicast tree.

Scribe can well react on node failures. Each parent node in a multicast tree sends heartbeats to its children. If a node detects that its parent node becomes faulty, because it does not receive its heartbeats, the node can simply send a new JOIN message towards the RP. Pastry will find a new and functional route to the RP.
Scribe is also able to handle the failure of the RP. The sources can immediately detect the failure of the RP, because they directly send to the RP. Therefore, each source uses Pastry again to deliver a multicast message to the new RP, which is now the node numerically closest the groupId. The new RP answers with its IP address again.
The RP uses its namespace set to replicate its group state among some of its closest nodes. Thus, when the RP's immediate children detect the failure of the RP, they also use Pastry to route a new JOIN message to the new RP. With high probability, the new RP already has a replica of the state associated to the group, and therefore the new RP can serve the multicast group almost without any interruption.

2.5 File Dissemination Protocols based on Multicast or P2P Networks

2.5.1 FTP-M

FTP-M[5] is a FTP like multicast application. It extends the FTP[33] protocol and its user interface. FTP-M focuses on the application layer instead of the transport layer. It is built on top of an existing strict-reliable multicast protocol like TCP-M[34], MFTP[35] or IRMA[36].
FTP-M can be used to push data on several FTP-servers at once. This behavior can be used in several scenarios like mirroring of content among many FTP-servers. But, FTP-M does not support multicast in the other direction because this would imply the synchronization of the clients, which is not applicable.
The FTP-M application suite only requires some adaption of FTP-servers and FTP-clients. FTP-M is implemented to coexists with the traditional FTP mode. Then, user can initiate the FTP-M mode on their FTP-M capable clients. For the proof of concept, the inventors of FTP-M has used TCP-M as the reliable multicast transport protocol.

In the FTP user interface, users can initiate the FTP-M mode by the command *multi*. With this command, FTP-M capable clients can initiate more than one connection to FTP-M capable servers. After switching the mode, users connect to the servers they want to push their file. FTP-M clients open normal *control connections* to each server by repeatedly invoking the *open* command. User can moving, deleting or renaming files or changing the working directory on the servers as usual. But, at any time, only one control connection is active. Therefore, user can switch the active control connection by again invoking the *open* command.
When on each server the right working directory is chosen, users can start the multicast file transfer to the servers. This is accomplished by the *put* command as on normal FTP clients. Internally, clients sent an new protocol command, *PASM*, to each server. This command is related to the *PAS* protocol command in the normal FTP mode, which initiates a passive data

transfer. But in the multicast mode, servers must create a listening TCP-M socket and reply to clients with their IP and port address of the TCP-M socket. Then, clients can start sending the file through their TCP-M socket to the receiving servers. Figure 2.8 shows one client and two connected servers with their involved components. TCP-M handles all tasks related to multicast the data to both servers.

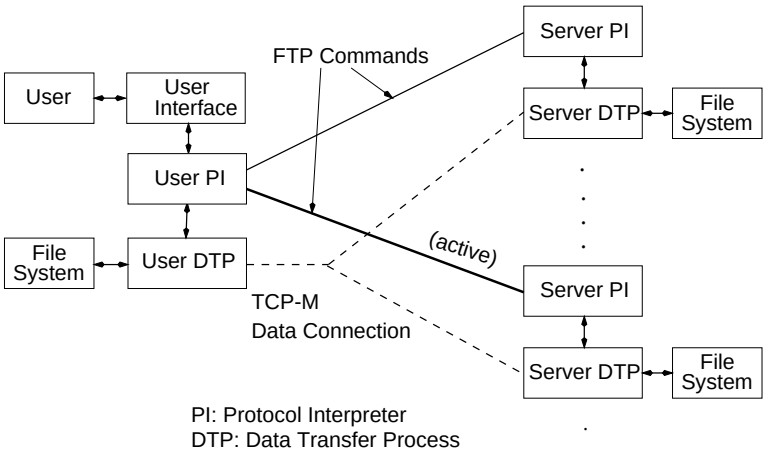


Figure 2.8: The FTP-M mode (source [5])

TCP-M uses an additional protocol layer between the TCP and the IP protocol layer, called Group Module (GM), which hides the multicast components from senders and receivers. Senders and receivers assume that they are normally connected to one host. The GM establishes the multicast group used to disseminate the data – TCP packets – to all receivers. Receivers send normal ACKs for the received data, which the GMs on the receivers send to the GM of the sender. The GM of the sender fuses the received ACKs and passes a summarized ACK to the above TCP socket.

2.5.2 FastReplica

FastReplica [37] is a replication mechanism for large files within Content Deliver Networks (CDN). It uses an algorithm for a fast file replication limited to 10 - 30 nodes, called FastReplica in the Small. Then, this algorithm is iteratively applied to provide file replication for hundreds and thousands of nodes, called FastReplica in the Large.

FastReplica in the Small assumes a low number of n nodes. The file F is divided in n equal subfiles $F_1, ..., F_n$. Then, the originating node N_0 makes n concurrent connections to the n nodes and sends node N_i the subfile F_i and a list of nodes to which the subfile F_i must be sent in the next step. After node N_i received the subfile F_i, it opens $n - 1$ concurrent connections to the nodes stated in the previously received list and sends the subfile F_i to them. At the end of this second step all n nodes received the entire file.

To scale to a higher number of receivers, FastReplica in the Large is used. It partitions the set of nodes into replication groups, each consisting of k nodes. Then FastReplica in the Small is iteratively applied to all of these groups. It starts with the first replication group, where the file originator replicates the file to the first k nodes. In the next step, these k nodes act as the originators for the next k replication groups, where again FastReplica in the Small is applied. This is repeated until all nodes have received the file.
FastReplica also incorporates mechanisms to deal with node failures and thus provides reliability, and to deal with the situations where the number of nodes is not a multiple of k.

2.5.3 Bullet

Bullet[6] is a high bandwidth data dissemination system that uses an overlay mesh. It tries to overcome the problems of IP Multicast like reliability and congestion control. IP Multicast does also not consider bandwidth when constructing its distribution trees. Bullet also tries to overcome the problems of high bandwidth data dissemination and reliability in pure tree-based multicast overlays. In tree-based overlay networks, a lot of bandwidth is wasted for the controlling and building of the tree because bandwidth has to monotonically decrease from the root to the leaves. The bandwidth probing that is implied by the demanded bandwidth decreasing also uses bandwidth, which cannot be used for the dissemination. If a node failure or data loose occurs high up in the tree all nodes lower down in the tree will also suffer from that loose. And any node's bandwidth is limited by the bandwidth of its single parent. Bullet uses TFRC[38] to be TCP-friendly.

Bullet also uses a distribution tree as its basic network topology. Figure 2.9 shows a simplified Bullet overlay network. The continuous arrows represent the distribution tree. But instead of using a reliability mechanism along the tree, Bullet assumes that missing data can be received from other peers in the tree, which have sufficient free upload bandwidth, shown by the dashed arrows in Fig. 2.9. Thus, Bullet does also not demand that nodes have to be able to forward the whole data to all its children.
Bullet achieves high bandwidth dissemination and reliability by an additional mesh overlay laying above the arbitrary overlay tree and a clever data dissemination. Each parent node tries to send all received data to its children. But, if the peer is not able to send all data, it sends disjoint data sets to its children. The level of disjointness is determined by the available bandwidth to each of its children. Therefore, the whole data has to be split into sequential blocks.

Then, peers can look for other peers that have their missing block with a mechanism called RanSub [39]. RanSub is an algorithm that is able to distribute random subsets of participating nodes throughout the tree. Distribution messages of RanSub contain summary tickets, which describe in a compact fashion which blocks a peer has. Then, each peer can choose and connect to the peers that provide the most missing blocks and download them.

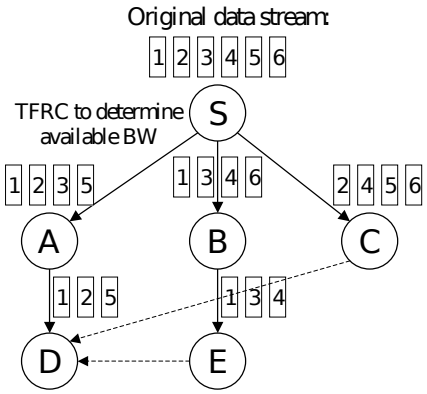


Figure 2.9: A simple Bullet network (source [6])

2.5.4 Slurpie

Slurpie [7] is used to discharge a server providing bulk data. Clients that are interested to download a file from an ordinary FTP or HTTP server by using Slurpie form a mesh and try to download most data from each other instead of the server as shown in Fig. 2.10.

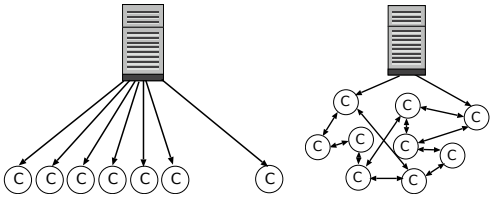


Figure 2.10: Difference between traditional bulk data download and downloading using Slurpie (source [7])

The functionality of Slurpie can be used without any modifications of the FTP or HTTP servers. Slurpie splits each file in blocks of 256 kilobytes. An additional topology server is used as entry point for each client using Slurpie. Clients ask the topology server to respond with a list of other clients downloading the specified file. After getting this list, clients connect to some of the provided clients and therewith become members of the Slurpie mesh. Then, clients start to exchange update messages that contain connection information and a block-list of the message originating peer. According to the received block-lists, clients can decide which blocks they can download from other clients and which blocks they have to request at the server.
Slurpie incorporates some mechanisms to reach good TCP performance, to not congesting servers, and to control the number of open connections to other clients – mesh degree.

The number of data transferring connections is controlled by a simple bandwidth estimation technique. The congestion of servers is avoided with a random backoff interval each client must wait before contacting the server. The mesh degree is controlled by a local group size estimation to maintain a mesh degree of $\log(n)$.

2.6 The Network Simulator - ns-2

The network simulator ns-2 [32] is a discrete-event simulator, that is widely used in academic research. It supports different networking models like wired, mobile, satellite or LAN. For this thesis, only the wired network model is used, because with this model, the Internet could be modeled. It performs packet-level simulation which means that discrete events are events related to packets, like a packet is enqueued or dequeued or enters a node.

2.6.1 Components in ns-2

The scheduler

The scheduler is the main component of ns-2. As its name implies, it schedules the events in the whole simulation. It also contains all functions necessary to build a simulation. The scheduler provides functions for creating nodes, connecting them with links, adding link-monitors, adding tracing capabilities, connecting agents with nodes, selecting the routing model, and as main task adding, removing and dispatching events.
The scheduler has different scheduling algorithms implemented. ns-2 provides a simple list scheduler, a heap scheduler and a calendar queue scheduler. The calendar queue is the fastest priority queue and selected as default.

Nodes

The wired network model has two different node types. One node type is for unicast routing and the other node type is for multicast routing. Figure 2.11 shows an unicast node and Fig. 2.12 shows a multicast node. Both node types have an address assigned, store a list of their neighbors, a list of their agents and a routing module.
In the unicast node, the address-classifier is responsible for packet forwarding. There are several different address-classifiers, each corresponding to a specific routing module. If a packet's destination is the nodes address, the address-classifier handles the packet to the port-classifier. Then, the port-classifier passes the packet to the right agent. This addressing scheme is like TCP/IP's addressing, but it is not limited to it.
In a multicast node, there exist two different address classifiers. One for the unicast routing and one for the multicast routing. When a packet first enters a node, it is handled depending on the destination address by the right address classifier. When the address is a multicast address, the multicast classifier handles the packet depending on its source and destination address to the replicators. The routing module sets up the replicators, which really only replicate packets and forward them to the different desired links or an agent on the node.

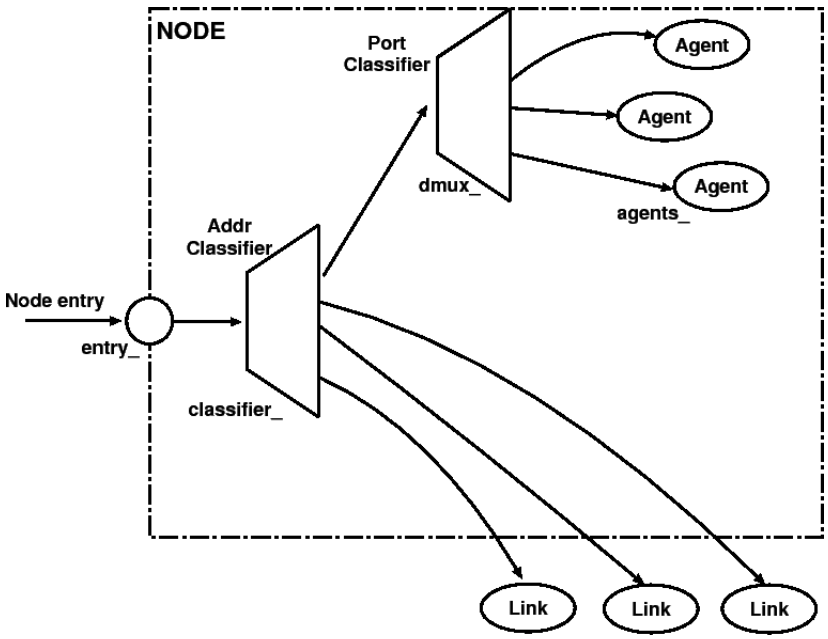


Figure 2.11: A unicast node in ns-2 (source [4])

Links

A link is the most important component of the simulator, because it consists of a queue, the link itself and a TTL checker. Figure 2.13 shows such a link. All parts of the link ending with ...T_ are used for tracing and are not relevant for the link behavior. The drophead part is simple an entry point for packets being dropped on the link and is also used for tracing capabilities.
The queue resides in ns-2 on the link rather than on the router. Thus, the first part a packet enters, is the queue. In ns-2, several different implementations of queues exist, such as DropTail or Random Early Drop queues. The link itself models the specified delay and bandwidth of the link, and before the link handles the packet to the next node, the TTL value is checked. For routing algorithms, a cost as simple numeric value can be assigned to a link.

Agents

Agents are the packet producers and consumers in ns-2 and are used to implement protocols at different network layers. In ns-2, several Agents are already implemented. ns-2 provides UDP, different TCP, RTP and many other agents.

Packets

All packets in ns-2 are also events. For instance, each link simulates the transmission and propagation delay of the packets flowing through it, by rescheduling the packets in the scheduler. A packet is formed of nested packet headers. Therefore, all packets have a common header, which

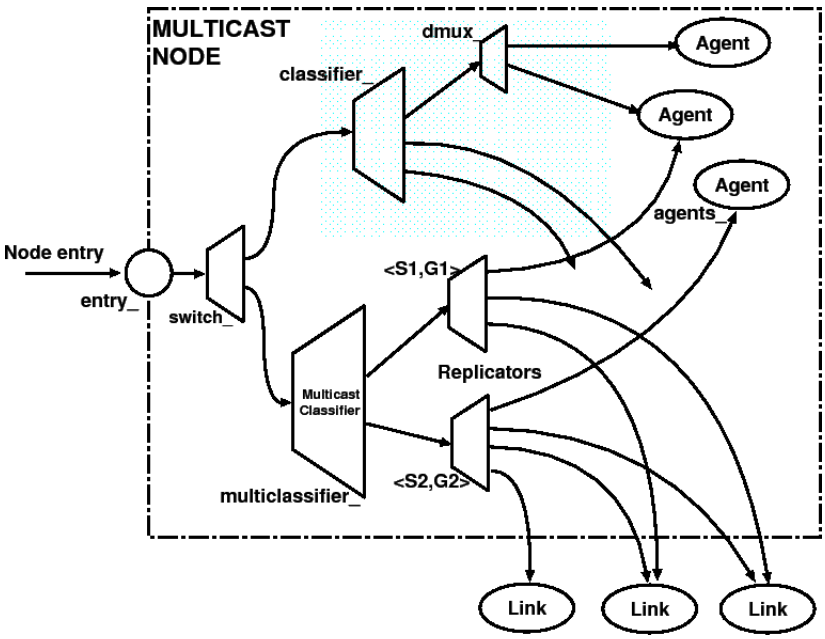


Figure 2.12: A multicast node in ns-2 (source [4])

contains the unique id, a packet type and a size. A packet could be extended with an IP header, which contains a source and a destination address, and further with a TCP header, which contains fields related to TCP. Each agent that provides its own network protocol, can use its own packet header.

2.6.2 Unicast and Multicast Routing in ns-2

Unicast Routing

ns-2 supports different unicast routing strategies.
The simplest routing strategy is **static routing**, where all routes are computed once at simulation start. This module knows the whole topology from the simulator itself, and sets up the classifiers in the nodes accordingly. Static routing runs a Shortest Path First algorithm of Dijkstra. Thus, this strategy does not support any changes in the topology after the simulation has started.
The **session routing** strategy acts like static routing, but whenever the topology changes, it recomputes all routes.
ns-2 also supports a **dynamic routing** strategy, which makes use of extra routing agents that exchange messages with each other to build a routing table for each node. The classifiers are set up accordingly to the routing tables. ns-2 already provides a Distributed Bellman-Ford algorithm and a link state routing protocol.
Routes can also be manually added to nodes which is referred to as **manual routing**.

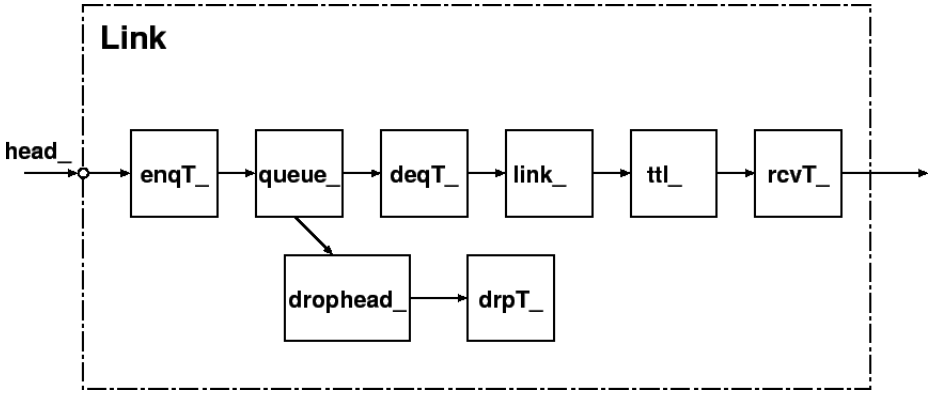


Figure 2.13: A link in ns-2 (source [4])

Multicast Routing

ns-2 provides different multicast routing protocols.
The **dense mode** is an implementation of a dense-mode-like protocol. It can run in two different submodes. The default one acts like PIM-DM and the other one is more like DVMRP, because it maintains parent-child relationships to reduce packet flooding.
The **centralized multicast** mode implements multicast similar to PIM-SM. It builds Rendezvous Point rooted shared trees for the multicast groups. But in contrast to PIM-SM, it does not send any join, prune or graft messages to set up the tree. It uses a centralized computation instance to set up the multicast trees among nodes. Data packets originating from the source are encapsulated and sent using unicast to the RP. The RP decapsulated the packets and sends them to the joined receivers. Encapsulated packets are sent to the RP even if there are no receivers, which also differs from PIM-SM.
The **shared tree** mode implements a simplified sparse mode protocol. The RPs are set up for each group before the simulation starts. Nodes send a graft message towards the corresponding RP when joining a group. To leave a group, they send a simple prune message.

Chapter 3

The Multicast File Transfer Protocol - MCFTP

3.1 Overview

With the Multicast File Transfer Protocol (MCFTP) peers are able to download files by using multicast groups instead of unicast connections. First, the core protocol is explained in Section 3.2. Then, the additional protocol mechanisms and messages are introduced to use the protocol in a centralized manner, Section 3.3, and a fully decentralized manner, Section 3.4.
The two proposed MCFTP versions form both an unstructured, loosely coupled P2P network with a flat hierarchy. The centralized version is a hybrid P2P network and the decentralized version is a pure P2P network.
The protocol itself has no mechanism to search for files, like Gnutella or Napster have. It does not form a swarm of peers to make queries on it. Instead, it is used to form swarms one for each specific file, like BitTorrent.

3.2 Core Protocol

MCFTP consists of four main components. The first component is the FileDescriptor which is a text file that contains meta data about the file itself. Second is the FileManagementGroup, which is used as main communication channel of the swarm. Third are the SendingGroups, where the data transfer takes place, and the fourth component is the participating peers. Figure 3.1 shows the four main components of the multicast file transfer protocol. This figure does not only present one MCFTP swarm, but three. Each swarm starts with the file, because the files represent the points of entry.

3.2.1 FileDescriptor

Each file, which is downloadable by MCFTP, must have a FileDescriptor. A FileDescriptor is a text file, which contains all meta data for the corresponding file. It is similar to the torrent file in BitTorrent. The FileDescriptor must at least contain a unique file identifier, the size of the file,

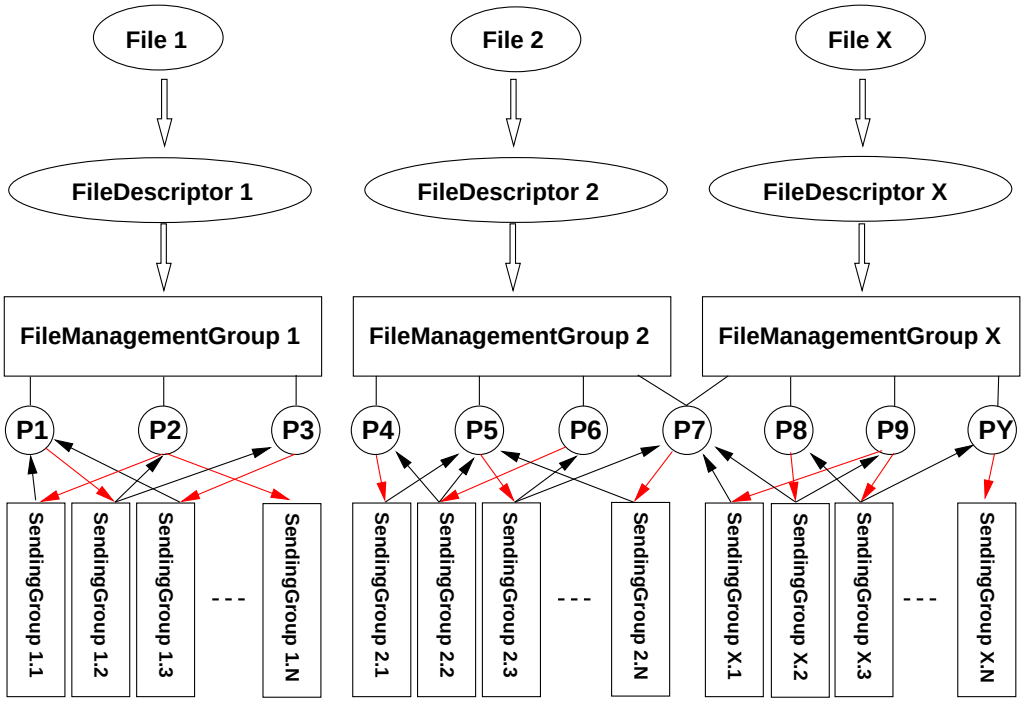


Figure 3.1: Relations between Files, FileDescriptors, FileManagementGroup and SendingGroups

the size of one chunk and a list of hash values, one for each chunk.

The file identifier is a hash value over the whole file. A hash function can guarantee that each file gets a unique identifier and it can be also used to validate the correctness of the entire file after download. The Secure Hash Algorithm One (SHA-1) [44] is a good choice for this purpose, because it is widely used and several free implementations exist. Although some collision attacks are known and cryptographic experts advise to not further use it for cryptographic applications, this does not restrict the use as identifier and integrity check. It does not make sense to choose a stronger hash algorithm, which produces bigger fingerprints and needs longer computation. This would only result in much bigger FileDescriptor files.

Each file is split into chunks of equal size, except the last chunk, which could be smaller. These chunks are the smallest transferred units. Chunks are indexed starting with zero at the beginning of the file. To make not a too high number of chunks, which would increase the costs of administrating the chunks, the size of the chunks is chosen to split a file in a maximum number of 2^{16} (65536) chunks. The smallest chunk size is 2^{18} (262144) Bytes and must be doubled each time the maximum number of chunks is reached. But, these are only mandatory upper boundaries to have chunk indexes below 65536. The user is free to choose a bigger chunk size. Perhaps he wants to make a smaller FileDescriptor file or he wants to have smaller administrative complexity. Table 3.1 illustrates the maximum possible file size for a given

Number of chunks	**Maximum file size** (bytes)	**Chunk size** (bytes)
65'536	17'179'869'184	262'144 (0.25 MB)
	34'359'738'368	524'288 (0.5 MB)
	68'719'476'736	1'048'576 (1 MB)
4096	1'073'741'824	262'144 (0.25 MB)
	2'147'483'648	524'288 (0.5 MB)
	4'294'967'296	1'048'576 (1 MB)

Table 3.1: The relation between maximum possible file size and chunk size for a given number of chunks

chunk size, so that the maximum number of chunks is not exceeded. The upper three rows show the file size for a maximum of 65536 chunks and the lower three rows show the file size for a maximum of 4096 chunks.

Because the chunks are the smallest transferable units, a hash value is provided for each chunk to check their integrity after download. These hash values also need no cryptographic strength. They are purely used to validate the correctness of the downloaded chunks. Thus, the SHA-1 hash algorithm is a good choice for that matter as well.

The FileDescriptor contains additional information that is necessary for the usability of MCFTP. But these are not mandatory for proper functioning of MCFTP. The filename of the file is nice to have, but is purely advisory. Also a free text comment or a creation date of the file is good for end-users or for searching specific files, but they are not needed by the protocol itself. The FileDescriptor can be extended with more such optional information as needed.
The FileDescriptor can be encoded as XML or Bencoded [42] like the torrent files in BitTorrent.

The first person or organization who wants to make a new file available, must create the FileDescriptor for this specific file. The FileDescriptor needs then to be published on a web-site, FTP server, etc. Because the FileDescriptor is a text file, everybody else can download and republish it.

3.2.2 FileManagementGroup

The FileManagementGroup (FMG) is an ordinary multicast group. Via this multicast group, all coordination messages are exchanged. Therefore, joining the FileManagementGroup means joining the swarm. Depending on which MCFTP version is used, different messages are sent via the FileManagementGroup. For every file, exactly one FileDescriptor exists and for each FileDescriptor, exactly one FileManagementGroup exists.

3.2.3 SendingGroups

A SendingGroup (SG) is also an ordinary multicast group. SendingGroups are used to transfer chunks of the file. For each SendingGroup it is specified which chunk is sent from which

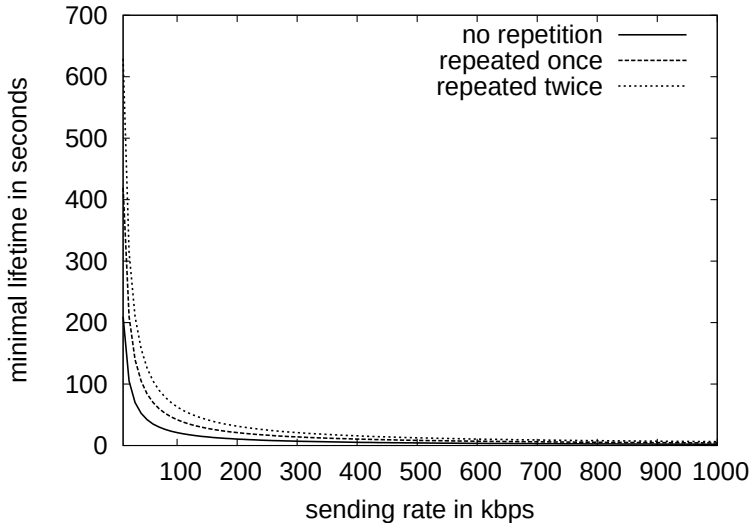


Figure 3.2: The relation between sending rate and lifetime of SendingGroups

peer with which bandwidth and for how many times using which multicast group. New SendingGroups are announced via the FileManagementGroup only once, when they are newly created. The peers, which have joined the FMG, receive these SendingGroup-announcements and can decide if they want to join a specific SendingGroup or not depending on their own state.

SendingGroups have relative small lifetimes. A SendingGroup's lifetime begins with the announcement and ends right after the last packet of the chunk that has been sent. With the information published in the SendingGroup-announcement, every peer can calculate each SendingGroup's lifetime as well. Figure 3.2 shows three graphs with the minimal lifetime of SendingGroups corresponding to the sending rate. The size of a chunk is set to 256 KB in all graphs and how many times a chunk is repeatedly sent within a SendingGroup – the chunk repetition – ranges from none to two. The three graphs show that the lifetime of a SendingGroup depends more on the chosen sending rate than on how often a chunk is repeated in a SendingGroup.
This minimal lifetime of each SendingGroup must be extended with some constant additional timeouts to avoid collisions on the senders, to honor propagation delays and to give the peers some time to join the multicast group.

The data itself is transferred within a SendingGroup with simple messages. A **Data** message contains the index of the chunk, the byte offset of the payload within the chunk, the length of the data payload and the payload itself. The first byte of the chunk is at index zero. A sender starts to send at the beginning of a chunk. It splits the chunk into several Data messages of the same size, except the Data message containing the last bytes of the chunk, which can be smaller.

3.2.4 Peers and Protocol Operation

Every peer has a randomly chosen identifier (ID). This ID is a twenty bytes long string and must be unique within the FileManagementGroup. This ID is automatically set by the client when it is first started. For more anonymity of the peers in the swarm, the ID can be changed every time the client starts again. The size of the ID is long enough to contain a 160 bit fingerprint of a public key as identification.

The user must set the maximum usable upload and download bandwidth. Because multicast traffic flows often do not scale and adapt like TCP, these parameters do advise a client to not use its entire available bandwidth. These parameters must be set to a reasonable value. If they are set too high, packet drops will occur. Most multicast transport protocols, like UDP used atop of IP Multicast, are not reliable, thus this could lead to an effective download rate of zero.

When a user wants to download a specific file, he must search the corresponding FileDescriptor first. Because FileDescriptors are like torrent files, the search for a specific FileDescriptor can be done like the search for a specific torrent using regular websites or other publishing platforms. After downloading a FileDescriptor, the user has all necessary information for downloading the file, except the multicast group identifier to join the FileManagementGroup of the file. The multicast group identifier can not be stored in the FileDescriptor because the relation between the FileDescriptor and the FileManagementGroup is not static. If this assignment would be static, every newly published file must exclusively reserve its own multicast group. This results in an endless growing list of reserved multicast groups. Therefore, this relation between the FileDescriptors and the FileManagementGroups must be dynamic.

In the previous paragraph the abstract term multicast group identifier is used, because it must be distinguished between IP Multicast and Application Layer Multicast (ALM). In IP Multicast this identifier simply is an IP Multicast address. But when using ALM, a peer must first know some other peers to be able to join the Overlay Network.

This mapping between file ID and multicast group identifier is stored in a Distributed Hash Table (DHT). The key is the file identifier also used in the FileDescriptor and the value is the multicast group identifier of the corresponding FileManagementGroup. The DHT must be formed of all peers running an MCFTP client. To join the DHT, a new peer must know at least one peer that is already a member of the DHT. Thus, the user must configure at least one address of a bootstrap server, that serves as entry point or the peer has some history information of participating peers from previous sessions.

To also work with ALM the returned value must contain some members of the Overlay Network. If the Overlay Network allows only one multicast group per Overlay Network, the returned value contains only IP address and port pairs of several peers that have already joined the Overlay Network. But if the Overlay Network allows more than one multicast group per Overlay Network, the returned value must contain some IP address and port pairs of some globally joined peers and the group identifier for the FileManagementGroup.

When the peer has successfully joined a FileManagementGroup, it can listen to the SendingGroup announcements. If a SendingGroup contains a chunk that the peer does not yet have and the sending rate is smaller than its download capacity left, the peer can join the corresponding multicast group and download that chunk. The peer musts also interact with the swarm as requested depending of the MCFTP version used.

3.3 MCFTP with Central SendingGroup Management

The centralized Multicast File Transfer Protocol, cMCFTP, introduces an additional component called the FileLeader. This is the instance where the central SendingGroup management takes place. The FileLeader collects information about the participating peers and on the basis of this information, it decides which SendingGroups should be created. Therefore, exactly one FileLeader must always exists per FileManagementGroup. But, each peer has to be able to act as the FileLeader, so that no single-point of failure exists. Therefore, a negotiation mechanism has to subsist to select one peer as the FileLeader.

3.3.1 Messages between the FileLeader and the Peers

The FileLeader and the participating peers communicate via the FileManagementGroup. The FileLeader sends StatusRequest and KeepAlive messages to the peers. The peers send FullStatus and PartialStatus messages back to the FileLeader.

The **StatusRequest** messages are periodically sent by the FileLeader to the peers. This message asks the peers to return a FullStatus message. The time between two messages is called request interval I_R of the FileLeader. The message contains the peer ID of the FileLeader and a backoff interval in seconds. The backoff interval is equal to the request interval of the FileLeader. When a peer receives a StatusRequest message, it uses the backoff interval value to produce a uniformly distributed random value between zero and the backoff interval and starts a timer with this random value. When the timer has expired, the peer returns a FullStatus message. The use of these backoff timers makes the network load of the FullStatus messages from the peers towards the FileLeader uniformly distributed.

A **FullStatus** message contains all information about a peer, which is used by the FileLeader for creating and maintaining SendingGroups. A FullStatus message consists of:

- the ID of the originating peer
- the upload bandwidth of the peer
- the download bandwidth of the peer
- the uptime of a peer (how long it is running)
- a bit vector, showing which chunk the peer has and which are missing

Depending on how many peers have joined the swarm, the FileLeader can adapt the request interval. Because the backoff interval is equal to the request interval, this has also direct influence on the incoming network load on the FileLeader.

Because these FullStatus messages are big in size, a smaller message is also used to inform the FileLeader of newly downloaded chunks. This smaller message is called **PartialStatus** message. It contains the same fields like the FullStatus message except the bit vector of the FullStatus is replaced with a list of the newly downloaded chunks. To really save bandwidth, these smaller messages are only triggered if at least some unreported chunks are newly downloaded or if a fraction of the backoff interval has expired. The PartialStatus message can also be made even smaller if it only would contain the peer ID and the newly downloaded chunks indexes.

When a peer newly joins a specific FileManagementGroup, it must immediately communicate its status to the FileLeader. This is done with a FullStatus message. Thereafter, it could report the newly downloaded chunks only with PartialStatus messages. But it is not assured that all PartialStatus messages arrive at the FileLeader, because multicast communication generally is not reliable, and hence packets can be lost. This leads to a fragmentary status of peers at the FileLeader. Thus, with these PartialStatus messages, the network load towards the FileLeader can be reduced by increasing the request interval, but the FullStatus messages are still needed.

KeepAlive messages are periodically sent by the FileLeader to its peers. They are used to announce new SendingGroups. The peers can track possible FileLeader interruption or failure if these messages have not been received for a while. If no new SendingGroups are newly created by the FileLeader, then the KeepAlive message contains no SendingGroup entries and the message is sent as pure KeepAlive message. To prevent packet fragmentation, the KeepAlive message contains a maximum number of SendingGroups, which also limits the maximum number of active SendingGroups in the swarm.

A KeepAlive message contains:

- the ID of the FileLeader
- the size of the swarm
- a list of new SendingGroups (0 - N),
  where each entry contains:
  - the peer ID of the sending peer
  - the chunk index
  - the multicast address
  - the sending rate
  - the number of repetitions

When receiving KeepAlive messages, peers react differently.
A peer, which is mentioned as a sending peer, must start sending the suggested chunk after a certain waiting period. This sender has received all information with the KeepAlive message to perform the sending. Because of the information maintained by the FileLeader, a sender will have enough upload bandwidth free to serve the assigned SendingGroup. A certain sender can still have some other SendingGroups to serve, but its maximum reported upload bandwidth will not be exceeded by the FileLeader's sending instructions.
Peers, which do not have the chunk, decide by themselves depending on their available download bandwidth, if they want to join a specific SendingGroup. The downloading peers are responsible by themselves to saturate their possible download rate. Peers can act as senders and downloaders at the same time.

For each SendingGroup, a multicast group is used. Therefore, the number of multicast groups used is at least equal to the number of SendingGroups currently active in the swarm. But each group should be inactive for some timeout interval before reusing it. If a group is directly reused, for a short period there could exist two senders of different SendingGroups, which use the same multicast group. Thus, a peer joining one of the two SendingGroups receives the traffic of both. This can lead to a congestion and packet drops, if the overall download rate of such a peer is above its limit. Therefore, the total number of multicast groups a FileLeader uses is bigger than the number of active SendingGroups. The total number of multicast groups used depends on the number of peers in the swarm, the KeepAlive interval and the upload bandwidth of the peers respectively the sending rate of the SendingGroups.

3.3.2 Load on the FileManagementGroup and Solutions

Via the FileManagementGroup, all control messages are sent. Keep in mind that the StatusRequest messages and the KeepAlive messages are sent from the FileLeader and must be received by all participating peers. The FullStatus and the PartialStatus messages are sent from the peers and are received by the FileLeader. The presented equations are verified with the simulation results in Section 6.6.1.

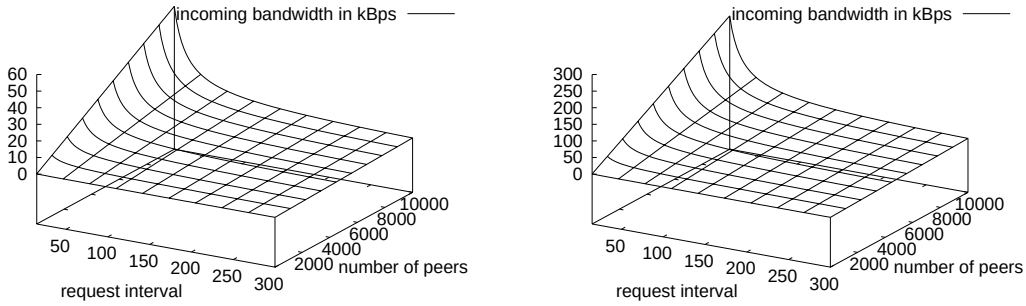
Incoming Traffic on the FileLeader

The incoming traffic on the FileLeader consists only of Status messages. The traffic generated by the FullStatus messages can be calculated with Eq. 3.1. It shows the simple relation between the number of peers in the swarm N_P, the size of the FullStatus message S_F in bytes, the request interval I_R in seconds and the incoming bandwidth B_{IF} as bytes per second.

$$\frac{N_P \times S_F}{I_R} = B_{IF} \tag{3.1}$$

As a example, Fig. 3.3 shows a 3D graph of the incoming bandwidth on the FileLeader using the Eq. 3.1. In this case, the FullStatus message has a size of 386 bytes. Both messages include 48 bytes used for the peer ID, the up and download bandwidth of the peer, the uptime and four bytes as common message header. The smaller FullStatus message contains a bit vector of 400

chunks, which represents a file of 100 MB assuming a chunk size of 256 KB. The bigger one contains a bit vector of 2800 chunks, which represents a file of 700 MB. The number of peers ranges from 30 to 10000 and the request interval ranges from fifteen seconds to five minutes.



(a) with 400 chunks

(b) with 2800 chunks

Figure 3.3: Incoming bandwidth in kilobytes per second consisting of the FullStatus messages on the FileLeader depending on the number of peers and the request interval

Both graphs show that the FileLeader can maintain the incoming bandwidth below 20 kilobytes per second (kBps) in these two examples, even with 10000 peers. In Fig. 3.3(a) this can be accomplished with a request interval of at least 43 seconds. This request interval should be enough short for the FileLeader to have enough information about the swarm to perform well. In Fig. 3.3(b) this low incoming bandwidth is only accomplished with a request interval of at least 193 seconds.
The worst case is a FullStatus message containing a bit vector of the maximum allowed 65536 chunks. To maintain an incoming bandwidth of twenty kBps, the request interval should be at least 4114 seconds! With this very long request interval, the FileLeader can not possibly perform as it should. Thus, the incoming bandwidth is heavily influenced by the number of chunks in a file and the request interval.

Equation 3.2 shows the bandwidth used for PartialStatus messages. How many PartialStatus messages can occur in a request interval for one peer is the idea behind this equation. This is represented with the term in the round brackets. The effective average download rate D_A in bytes per seconds of all peers is multiplied with the request interval I_R, which results in the total bytes transferred during this interval. Dividing this with the size of one chunk S_C in bytes and the number of chunks triggering a PartialStatus message T_C gives the number of PartialStatus messages sent in one request interval. And if a peer has at least one unreported chunk and half of the backup interval has expired, a PartialStatus message is also sent. This is represented with the addition of one. Then, the total number of PartialStatus messages of one peer in a request interval must be multiplied with the number of peers N_P, the size of a PartialStatus message and at the end it must be divided through the request interval I_R that results in the incoming

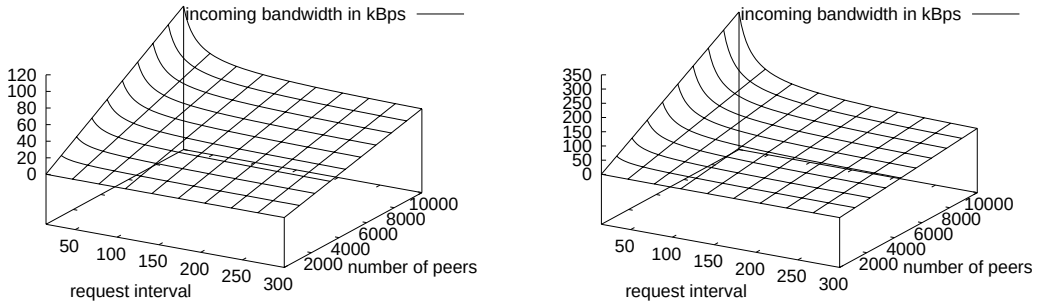
bandwidth produced by the PartialStatus messages B_{IP} in bytes per second.

$$\frac{S_P \times N_P}{I_R} \times \left(\frac{D_A \times I_R}{S_C \times T_C} + 1 \right) = B_{IP} \tag{3.2}$$

Adding both equations together gives the total incoming bandwidth B_I on the FileLeader as shown in Eq. 3.3.

$$\frac{N_P \times S_F}{I_R} + \frac{S_P \times N_P}{I_R} \times \left(\frac{D_A \times I_R}{S_C \times T_C} + 1 \right) = B_I \tag{3.3}$$

The example above is now extended with the PartialStatus messages. Therefore, the next two 3D graphs in Fig. 3.4 represent Eq. 3.3. The two FullStatus messages are of equal size as in the previous example. Also the shown range of the x-axis (request interval) and the y-axis (number of peers) are the same as in the previous example. A full PartialStatus message contains a peer ID, three chunk indexes each 2 bytes long, and the common header of four bytes. Thus, the message has a size of 30 bytes. The average download rate of all peers is set to 100 kBps.



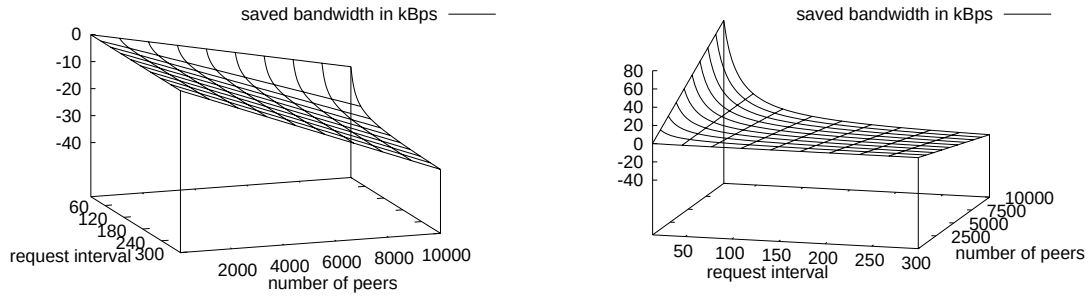
(a) with 400 chunks

(b) with 2800 chunks

Figure 3.4: Incoming bandwidth in kilobytes per second consisting of FullStatus and PartialStatus messages on the FileLeader depending on number of peers and the request interval

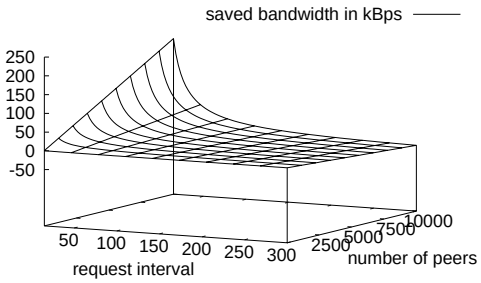
It should immediately attract attention that the additional incoming bandwidth is not insignificant. Especially in Fig. 3.4(a), the incoming bandwidth is almost doubled, compared to Fig. 3.3(a). When precisely looking at Eq. 3.2 the bandwidth does not depend on the number of chunks in the file. Therefore, the additional incoming traffic produced by the PartialStatus messages is always the same for a specified number of peers, request interval, chunk size and average download speed. When looking at Fig. 3.4(b), the additional bandwidth used for the PartialStatus messages is really the same as in Fig. 3.4(a), but the ratio between the bandwidth used for the FullStatus messages and the bandwidth used for the PartialStatus messages is not so bad. Thus, this ratio gets better with a raising number of chunks.

PartialStatus messages were introduced to save bandwidth but still to keep the FileLeader up to date. This is accomplished by increasing the request interval of FullStatus messages and by using PartialStatus messages. Therefore, Fig. 3.5 shows how much bandwidth can be saved if the request interval is doubled and PartialStatus messages are used instead of only using FullStatus messages. This is accomplished by calculating the difference between Equations 3.1 and 3.3 if the request interval used in Eq. 3.1 is inserted doubled in Eq. 3.3.
A small example to understand the graphs follows. In a swarm with 2000 nodes and a file containing 400 chunks, 6696 bytes per second more incoming bandwidth is used when the request interval is set to 60 seconds and PartialStatus messages are used, instead of using only FullStatus messages with a request interval of 30 seconds. See Fig. 3.5(a) at time of 30 seconds request interval and 2000 nodes, resulting in -6696 Bps, which means no bandwidth is saved.



(a) with 400 chunks

(b) with 2800 chunks

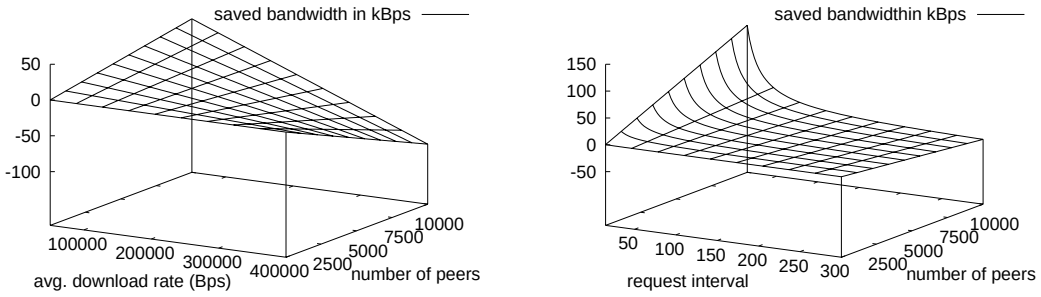


(c) with 6144 chunks

Figure 3.5: Saved bandwidth on the FileLeader when using a doubled request interval and PartialStatus messages instead of simple FullStatus messages

These graphs show that not in all situation incoming bandwidth can be saved. When the number of chunks in a file is small as shown in Fig. 3.5(a), it can not be beneficial to use PartialStatus messages. But starting from a certain number of chunks, this can be beneficial as shown in Fig.

3.5(b) and in Fig. 3.5(c). A next factor that influences bandwidth saving is the request interval. If it is small, more bandwidth can be saved, if it is big, more bandwidth is wasted.
When the equation used to calculate bandwidth saving is transformed so that the number of nodes is a single factor, the rest of the term can be interpreted as the slope of the graph. Thus, the number of nodes in the swarm does only enforce actual bandwidth saving independent of a negative or positive saving. So for any given number of chunks, the average download rate, how many times the request interval is increased and when a PartialStatus message is triggered, the maximum useful request interval can be calculated independent of the number of nodes. With a request interval bigger than that maximum, no incoming bandwidth could be saved.
To complete our considerations about incoming traffic, two further graphs are shown in Fig. 3.6. Figure 3.6(a) shows the influence of the average download rate of the peers. In this example, the request interval is fixed to 30 seconds. The bigger the average download rate is, the more chunks can be downloaded in one request interval. Therefore, bandwidth saving is smaller or even negative with bigger average download rate.
Figure 3.6(b) shows the influence on bandwidth saving, if the request interval is three times bigger instead of only doubled. It shows that with small request intervals more bandwidth can be saved and that the request interval where saving bandwidth changes to wasting bandwidth is also bigger as with a doubled request interval compared to Fig. 3.5(b).
Thus, in a request interval range between 30 and 100 seconds, incoming bandwidth on the FileLeader can be saved, if the number of chunks is not too small.



(a) depending on average download bandwidth

(b) with 3 times bigger request interval

Figure 3.6: Saved bandwidth on the FileLeader when using PartialStatus instead of simple FullStatus messages and 2800 chunks

Outgoing traffic on the FileLeader

The outgoing traffic on the FileLeader is easier to calculate as the incoming traffic because the StatusRequest and the KeepAlive messages are all periodically sent. Although the KeepAlive messages can contain different numbers of SendingGroups, the maximum bandwidth used can

be calculated. Equation 3.4 shows the maximum outgoing traffic produced by KeepAlive messages. The size of a pure KeepAlive message without SendingGroups is denoted as S_{KA}. The size of one SendingGroup is denoted as S_{SG} and the number of SendingGroups as N_{SG}. The interval between two KeepAlive messages is denoted as I_{KA}, which is independent of the request interval.

$$\frac{S_{KA} + N_{SG} \times S_{SG}}{I_{KA}} = B_{FOKA} \tag{3.4}$$

The outgoing bandwidth produced by StatusRequest messages is an easy calculation as well. StatusRequest messages are of fixed length and sent in fixed intervals. Equation 3.5 shows the required bandwidth in bytes per second depending on the StatusRequest message size S_{SR} and the request interval I_R.

$$\frac{S_{SR}}{I_R} = B_{FOS} \tag{3.5}$$

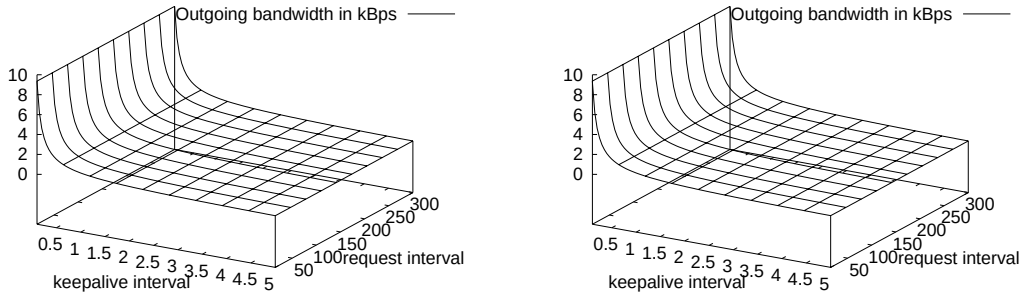
Summing up Eq. 3.4 and Eq. 3.5 results in the total outgoing traffic on the FileLeader as shown in Eq. 3.6.

$$\frac{S_{SR}}{I_R} + \frac{S_{KA} + N_{SG} \times S_{SG}}{I_{KA}} = B_{FO} \tag{3.6}$$

To get an impression of the above equations, two graphs are presented in Fig. 3.7. They show the outgoing bandwidth used for the StatusRequest and KeepAlive messages. In Fig. 3.7(a) the StatusRequest message contains only the peer ID of the FileLeader (20 bytes) the backoff interval (2 bytes) and the already introduced common header (4 bytes). In Fig. 3.7(b) the StatusRequest message additionally contains the peer IDs of 4 backup FileLeaders. The usage of backup FileLeaders is described in Appendix F.1.2. Therefore, a StatusRequest message has a size of 26 bytes in Fig. 3.7(a) and a size of 106 bytes in Fig. 3.7(b).
A KeepAlive message consists of the common header (4 bytes) the peer ID of the FileLeader (20 bytes) the size of the swarm (4 bytes) and the SendingGroups. The size of one SendingGroup is 31 bytes and one KeepAlive message can contain a maximum of 30 SendingGroups. This results in a message size of 958 bytes. Both graphs show, the outgoing bandwidth depending on the request interval in the range of 15 seconds to 3 minutes and the KeepAlive interval in the range of 0.1 to 5 seconds.
The two graphs in Fig. 3.7 shows that the outgoing bandwidth is moderate. It never exceeds 10 kBps. It also shows, that the outgoing bandwidth is not influenced by the StatusRequest messages, because they are too small in contrast to the KeepAlive messages and the request interval is much bigger than the KeepAlive interval.

Protocol Traffic on Peers

The outgoing bandwidth used for the protocol messages on a participating peer is really small. A peer sends during each backoff interval, which is equal to the request interval, a FullStatus message. If enough chunks are downloaded, a PartialStatus message is sent. Equation 3.7 shows this outgoing bandwidth usage B_{OP} in bytes per second. This equation is almost similar to Eq.



(a) StatusRequest without backup FLs

(b) StatusRequest containing 4 backup FLs

Figure 3.7: Outgoing bandwidth on the FileLeader produced by the StatusRequest and KeepAlive messages depending on the Request and KeepAlive interval

3.3. But the single summands are not multiplied with the number of peers and the effective average download rate D_A is now not an average over all peers, but instead is the average or the current download rate of the peer examined.

$$\frac{S_F}{I_R} + \frac{S_P}{I_R} \times \left(\frac{D_A \times I_R}{S_C \times T_C} + 1 \right) = B_{OP} \tag{3.7}$$

The incoming protocol traffic on one peer is bigger as the incoming traffic on the FileLeader. Although the own outgoing traffic must not be counted, the traffic of StatusRequest and KeepAlive messages are added. This is shown in Eq. 3.8.

$$\frac{S_{SR}}{I_R} + \frac{S_{KA} + N_{SG} \times S_{SG}}{I_{KA}} + \frac{(N_P - 1) \times S_F}{I_R} + \frac{S_P \times (N_P - 1)}{I_R} \times \left(\frac{D_A \times I_R}{S_C \times T_C} + 1 \right) = B_{PI} \tag{3.8}$$

The two equations above do not take into account that the Data messages also contain information, which counts as protocol traffic. Although this traffic does not additionally stress the FileManagementGroup because Data messages are only used in SendingGroups, it nevertheless counts as protocol traffic and stresses the access links of the peers. The pieces of a Data message that count as protocol are the common header, the chunk index, the offset, and the payload length field. Thus, each data message of size S_D also contains S_{PD} bytes of protocol data. Equation 3.9 shows the protocol traffic produced by the Data messages depending on the current incoming or outgoing pure payload bandwidth. This additional protocol traffic must be added to Equations 3.7 and 3.8 to represent the total incoming and outgoing protocol traffic.

$$\frac{S_{PD}}{S_D - S_{PD}} \times B_P = B_{PD} \tag{3.9}$$

Splitting the FileManagementGroup

As introduced in the previous sections about the incoming and outgoing protocol traffic, the incoming protocol traffic on the FileLeader and the incoming protocol traffic on the peers can be very high. But, peers must primarily download chunks and should not waste bandwidth with messages they do not use. To mitigate this problem, the FileManagementGroup can be split.

A first solution, the FileManagementGroup can be split into two independent groups. The Update Group, which only the FileLeader joins and the peers send their messages to, and the Management Group, which all peers join and the FileLeader is the single sender.
Therefore, the Full- and PartialStatus messages from all peers must be sent to the Update Group. StatusRequest and KeepAlive messages from the FileLeader must be sent to the Management Group. When a new FileLeader has to be negotiated that is described in Appendix F.1, the Negotiation messages from the peers also need to be sent to the Management Group. Thus, the incoming traffic on a peer is not represented with Eq. 3.8, but instead is reduced to the outgoing traffic of the FileLeader as shown by Eq. 3.6. The FileLeader still has the same incoming and outgoing network load, hence Eq. 3.3 and 3.6 are still valid.

The second solution is to not split the FileManagementGroup depending on who is receiving what, but instead is split into different FileManagementGroups depending on the bandwidth capabilities of the peers. A peer joins accordingly to its download rate a specific FileManagementGroup. All messages are distributed as previously described through one group. Thus, the network load of one FileManagementGroup is distributed to many.
Equations 3.8 and 3.7 for the peers and Equations 3.3 and 3.6 for the FileLeader are still valid. The reduced loads only result of the reduced number of peers N_P in the swarms.

The main advantage of the first solution is that the peers are fully freed of the traffic destined for the FileLeader. The swarm is also not split, therefore low bandwidth peers can benefit from high bandwidth peers. The opposite approach is also possible but not so probable. This splitting can be simply implemented. The query of the DHT must return two multicast group addresses instead of one. A disadvantage is the equal load on the FileLeader like no splitting would be done.

In contrast to the first solution, the second solution decreases the network traffic of the FileLeader, which is the main advantage. But the overall swarm is also split, thus fewer SendingGroups are offered to peers. Perhaps, the FileManagementGroup of the peers with the lowest bandwidth has no peer, which can act as a FileLeader. Therefore, a peer must be selected from a higher group, which offers its bandwidth to serve a lower group. When the swarm is split into several pieces, there also has to be a mechanism to handover some seeds from one FileManagementGroup to another. If all peers statically stay in their groups, starvation of one or more groups can occur.
Also, where should the lower and upper bandwidth boundaries be set? This could be dynamically set because it is a bad idea to make several small swarms, each performs bad and the total number of peers could be simply handled by one FileLeader. Thus, the FileManagementGroups

must be dynamically split. Also, the lookup to retrieve the right FileManagementGroup from the DHT is not as trivial as in the first solution. The key to search has to consist of the file ID and the bandwidth of the peer. Therefore, it is a combined query of a key, which has to be exactly matched, and a numerical value, which has to match a range. There exists some work on range and multi attributed queries [45] and [46]. But, these approaches have to be customized to fit the needs of cMCFTP.

Thus, the first solution is the best applicable, although the network load on the FileLeader is not minimized. This can be only achieved by increasing the request interval and by making the messages smaller.

3.3.3 Multicast Address Management

There are several issues when maintaining the multicast addresses. First, they must be allocated and reserved in a global scope. The addresses allocated by different swarms may not interfere with each other and with other multicast applications. Second, in each swarm, the addresses must be also maintained, especially if a FileLeader change occurs. The new FileLeader has to know which addresses are allocated and which of them are currently in use. Third, the number of multicast addresses available can be limited. It also has to be differentiated between native IP Multicast and Application Layer Multicast because the solutions are different.

Global Multicast Address Management

MCFTP allocates a lot of multicast groups. Each swarm has at least one FileManagementGroup and several SendingGroups.

In IP Multicast each FileManagementGroup has to have its own any-source-multicast group. But in IPv4, there do not exist many globally routable and dynamic usable addresses. When looking at Table 2.3 in Section 2.4.1 it can be only used the SDP/SAP block and the GLOB block. And the use of addresses in the GLOB block should also be arranged with the provider, which owns a specific GLOB block. Therefore, the SDP/SAP block is the sole really dynamically usable address block. But with only 32 thousand addresses available, this is quickly exhausted.
IPv6 brings more flexibility because the address space is really big. Looking at Table 2.4 in Section 2.4.1 the SDP/SAP block still has 32 thousand addresses. But, the unicast-prefix-based addresses become interesting. With a total of usable addresses equal to the half of the whole IPv4 address space per unicast-prefix, there exist enough free addresses to use as FileManagementGroups.

Each swarm also needs multicast groups for its SendingGroups. The problem of the limited number of IPv4 multicast addresses as described in the previous paragraph makes the use of any-source multicast group addresses almost impossible. But this problem can be avoided by using source-specific IP Multicast. To use source-specific multicast, cMCFTP must be extended, so that the FileLeader also knows the unicast IP addresses of the peers. This can be

accomplished by adding the unicast IP address to the FullStatus messages. The SendingGroups also have to be extended by adding the unicast IP address of the sender. Thus, the peers, which want to join a specific SendingGroup, make a source-specific join with the multicast and the unicast address. The FileLeader only has to be aware of not simultaneously assigning the same multicast address to the same sender. A big advantage of using source-specific multicast is that the addresses do not need to be globally allocated and reserved because a channel can not interfere with channels from other sources, although the same multicast address is used.
In IPv6, enough any-source multicast group addresses exist to use them also for SendingGroups. But the advantages of source-specific multicast are so big that any-source multicast should not be used for SendingGroups.

For IPv4 and IPv6, the used any-source multicast group addresses should be globally allocated and reserved to not interfere with other multicast applications. This can be achieved with the multicast address allocation architecture described in [47], although today this is not widely deployed. The multicast addresses within the SAP/SDP block require their own allocation and reservation system by using a distributed session directory [21].

Application Layer Multicast does not have the problem of a limited address space. If the Overlay Network used supports more than one group, a whole swarm maintains its own namespace. Therefore, different swarms can not interfere with each other. But, the Overlay Network has to provide an address allocation mechanism, to not have address collisions within the overlay.
If the Overlay Network used only supports one group, different swarms can also not interfere with each other. Also the multicast groups used for the SendingGroups in one swarm can not interfere with each other. Each peer has to setup some Overlay Networks ahead. The peer informs the FileLeader of its different Overlay Networks with the FullStatus messages. Thus, the FullStatus message must be extended by a unicast IP address and a list of ports. Each port corresponds to one Overlay Network. Also the SendingGroups do not contain a multicast address, instead they contain the IP-port pair of the Overlay Network initialized by the corresponding sender.
Therefore, Application Layer Multicast needs no global allocation and reservation mechanisms.

Local Multicast Address Management

The FileLeader maintains the multicast identifiers used to create the SendingGroups. As long as no FileLeader hand over occurs, everything works fine. The backup FileLeaders that are introduced in Appendix F.1.2 can also maintain a history of used multicast identifiers when they process the KeepAlive messages. Unfortunately, they can not be sure if their history contains all allocated multicast identifiers. But the chance to have an identifier clash is small if a backup FileLeader tracks the KeepAlive messages long enough. If a completely new FileLeader is negotiated, it can not know which identifiers are allocated and in use. Of course each peer can also maintain its own multicast address history. But, this is a computational overhead because the chance for one peer to become the FileLeader after a negotiation is really rare.
Therefore, another mechanism is used to maintain the multicast identifiers used to create the SendingGroups. By using an additional Distributed Hash Table (DHT) that is formed of the

peers of one swarm, the FileLeader can maintain a list of all allocated multicast identifiers. A distinction between IP Multicast and Application Layer Multicast has to be done.

Local Multicast Address Management in IP Multicast

When using source-specific multicast, no addresses need to be stored in the DHT. The FileLeader randomly assigns the multicast addresses used for the SendingGroups. The probability of having the same address assigned to the same sender at the same time is really small and this could only happen if a FileLeader change occurs. This address clash could be prevented by managing the source-specific multicast group addresses like the any-source multicast group addresses, but the additional costs are too high to legitimate this approach.

If any-source multicast is used for the SendingGroups, then a DHT is needed. Therefore, the FileLeader stores the multicast addresses used under a well known key, for instance a zero address or the address of the FileManagementGroup. If a specific multicast address is currently in use, this could also simply be stored in a separate entry with the address as key and an expire time as value. An expire time of zero is interpreted as not currently used. The list of all allocated addresses under the designated key have to be used because the new FileLeader can not check every address if it exists in the DHT or not.

Local Multicast Address Management in Application Layer Multicast

If the Overlay Network supports more than one group per overlay, the same solution as used for any-source IP Multicast can be used.
If the Overlay Network supports only one multicast group, only little has to be changed. To store the different existing Overlay Networks used for the SendingGroups in the DHT, the FileLeader can simply use IP-port pairs instead of single addresses. The rest works like described for any-source IP Multicast.
Application Layer Multicast, which supports more than one group, could have already implemented a mechanism to check if an address is already allocated and in use.

3.3.4 Security and Cooperation Problems

A security risk consists in the nature of multicast. Any peer could gather information about the whole swarm by processing the Status messages from the other peers. In Section F.2.1 a mechanism is presented, which makes it harder to gather this information. But it can not prevent it.
A next security problem looks at the possibilities to make a whole swarm not working. One simple possibility is to flood the FileManagementGroup with random data. Depending on the link capacity of such a disturber, or when attacking with a botnet, the FileLeader does not receive Status messages and peers do not receive KeepAlive messages. Peers can solve this problem by using a source-specific join to only receive messages from the FileLeader. But this implies that the IP address of the FileLeader is known, or the peer identifier in the multicast

Overlay Network. Also, the FileLeader negotiation is more complex when using source-specific multicast. The FileLeader still has to receive the messages from any source. Thus, it can not protect itself with a source-specific join.
Another possibility is to write wrong information into the DHTs. Therefore, a peer could not resolve the right FileManagementGroup for a given file ID. The DHT must be writable by all, hence it could not be protected from malicious peers.

In a multicast environment, any peer can join a multicast group. Thus, a peer can join the FileManagementGroup and does not report its membership with Full- and PartialStatus messages. This peer only listens to the KeepAlive messages and joins the SendingGroups it needs. It is not guaranteed that the peer can download the whole file, because the FileLeader does not make a SendingGroup for a chunk that only this peer does not have. But if the swarm is big enough, the chance to get the entire file is high.
A peer that reports its membership can not be forced to act as a sender. Although the FileLeader creates SendingGroups, which have that particular peer assigned as sender. cMCFTP has no mechanism like BitTorrent's tit-for-tat to enforce cooperation of the peers.

3.4 MCFTP with Distributed SendingGroup Management

The central SendingGroup Management approach of the Multicast File Transfer Protocol causes a lot of messages that have to be transferred via the FileManagementGroup in order to keep the FileLeader up to date. Also negotiating the FileLeader is a very complex task. Therefore, in this Section we propose a version of the Multicast File Transfer Protocol called dMCFTP, which has no FileLeader and where all participating peers are equal. SendingGroups are not created and announced by the FileLeader, but instead the peers decide by themselves which chunks they want to send. To achieve this, each peer maintains its own information about the swarm.

3.4.1 Messages between Peers

Only two different messages are sent by the peers. The first message is the SendAnnouncement message. Each SendAnnouncement message announces only one new SendingGroup. Thus, this message must contain the multicast group identifier, the ID of the chunk, the sending rate the chunk is sent with and how often it is repeated. The message can also contain the peer ID as an optional field, but this is not necessary. Peers receive a lot of those SendAnnouncements, which offer to join the SendingGroups they need. Using dMCFTP, peers are also responsible by themselves to saturate their download links.

The second message is used to inform other peers, which download bandwidths exist in the swarm. This BandwidthSignal message is periodically sent by each peer and contains only its download bandwidth. Other peers can now maintain a small history of the maximum and minimum download bandwidths available in the swarm. Peers use this information to determine the sending rate of SendingGroups they want to create.

3.4.2 Load on the FileManagementGroup

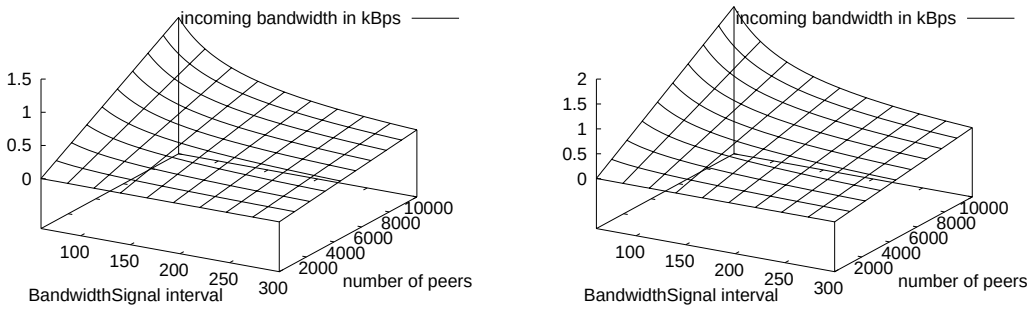
Via the FileManagementGroup, SendAnnouncement messages and BandwidthSignal messages are sent, which are received by all peers in the swarm. The presented equations are verified with the simulation results in Section 6.6.2.

Incoming Protocol Traffic on Peers

First, the traffic generated by the BandwidthSignal messages is analyzed. The BandwidthSignal messages are periodically sent. Therefore, the size of a BandwidthSignal message S_{BS} in bytes must be multiplied with the number of peers N_P in the swarm which represents the total bytes sent in one interval I_{BS}. Equation 3.10 shows the bandwidth B_{BS} used in bytes per second.

$$\frac{N_P \times S_{BS}}{I_{BS}} = B_{BS} \tag{3.10}$$

Figure 3.8 shows the bandwidth generated by the BandwidthSignal messages. In Fig. 3.8(a), the message size is set to 8 bytes, which includes the standard 4 bytes message header and the bandwidth encoded with 4 bytes. In Fig. 3.8(b), the bandwidth is encoded in 8 bytes, which results in a total messages size of 12 bytes. The interval in which the BandwidthSignal message is send ranges from one to five minutes and the swarm size ranges from thirty to ten thousand nodes.



(a) message size 8 bytes

(b) message size 12 bytes

Figure 3.8: Bandwidth in kilobytes per second generated by the BandwidthSignal messages

The two graphs show that the traffic caused by the BandwidthSignal messages is moderate. Thus, even peers with a very low access link should not suffer from this traffic. This traffic can be again reduced by making the sending interval of the BandwidthSignal depend on the distances to the maximum and minimum download rates that were communicated earlier. If a peer notices that it is not significantly contributing to determine the maximum or minimum download bandwidth, it should not send its BandwidthSignals that frequently as a more

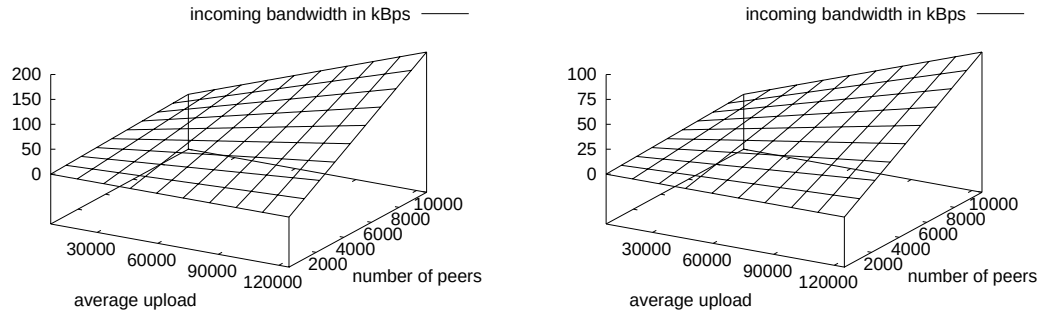
significant peer. This additionally reduces the traffic produced by BandwidthSignal messages.

The SendAnnouncements produce the main amount of the traffic in the FileManagementGroup. If a peer serves many SendingGroups at the same time, each of them has a low sending rate assigned in order to not exceed the upload bandwidth of the peer. Therefore, it will take a long time until the next SendingGroups can be created and announced by that peer. If the same peer creates only one SendingGroup, which consumes its whole upload, this group will expires much faster and the peer can announce the next SendingGroup earlier. But when the number of newly created SendingGroups is averaged over time, the number of newly created SendingGroups per second only depends on the total amount of bytes transferred in one SendingGroup and the upload rate of a specific peer.

Equation 3.11 shows the maximum traffic produced by SendAnnouncements. When U_A is the average upload bandwidth in bytes per second, S_C the size of one chunk in bytes and R how many times a chunk is sent in a SendingGroup, then the term $\frac{U_A}{S_C \times R}$ represents the average number of SendingGroups that one peer can newly create in one second. Multiplied with the number of peers in the swarm and the size of the SendAnnouncement messages, this results in the total traffic produced by the SendAnnouncement messages in bytes per second.

$$\frac{U_A}{S_C \times R} \times S_{SA} \times N_P = B_{SA} \tag{3.11}$$

Again, to have a better understanding of the traffic generated by the SendAnnouncement messages Fig. 3.9 shows two graphs as examples. The size of a SendAnnouncement is summed up to 39 bytes, which represents a big message including the peer ID and the sending rate encoded with 8 bytes. In Fig. 3.9(a), the chunk is sent once and in Fig. 3.9(b) it is sent twice. The chunk size is set to 256 KB, the average upload bandwidth ranges from 56 kbps to one Mbps and the number of peers ranges from 30 to 10'000.



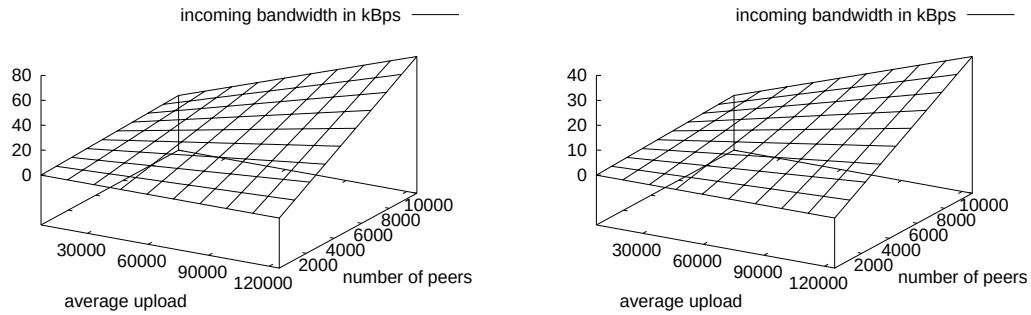
(a) no chunk repetition

(b) one chunk repetition

Figure 3.9: Bandwidth generated by the big SendAnnouncement messages

Of course Fig. 3.9(b) shows half the bandwidth of Fig. 3.9(a), because with no chunk repetition this is halved. But both graphs clearly show that with higher upload rates much more SendingGroups are announced which can lead to a big problem for peers with a low download rate on their access links. Also for peers with higher download rates, this theoretical maximum wastes too much bandwidth.

Thus, Fig. 3.10 shows almost the similar graphs, but with smaller SendAnnouncement messages. Here, the optional peer ID is omitted, also the sending rate is now encoded with only 4 bytes, which leads to a message size of 15 bytes.



(a) no chunk repetition

(b) one chunk repetition

Figure 3.10: Bandwidth generated by the small SendAnnouncement messages

Although the traffic caused by SendAnnouncement messages is reduced, a total incoming protocol traffic of almost 36'000 bytes per second can not be handled by every peer. The message size can not be farther reduced, but the size of the chunks can be increased. If the chunk size is doubled, the traffic is halved.

Adding both equations together results in the total traffic occurring in the FileManagement-Group. Also the number of peers in the swarm has to be reduced by one, because self produced traffic does not have to be taken into account. Equation 3.12 shows the total incoming bandwidth B in bytes per second on one peer. The bandwidth of the BandwidthSignal messages is not significantly contributing to the total incoming bandwidth of one peer. Thus, Fig. 3.9 and 3.10 can also be used to approximately represent the total incoming traffic on one peer, although the peer should not receive its own messages.

$$\frac{(N_P - 1) \times S_{BS}}{I_{BS}} + \frac{U_A}{S_C \times R} \times S_{SA} \times (N_P - 1) = B \tag{3.12}$$

Outgoing Protocol Traffic on Peers

The outgoing traffic consists of BandwidthSignal and SendAnnouncement messages. Equation 3.13 shows the total outgoing protocol traffic at one peer. This equation almost looks similar like Eq. 3.12, but for a single peer, it is not multiplied with the number of peers in the swarm. It should also not be used an average upload rate, instead the upload rate U of a specific peer should be used.

$$\frac{S_{BS}}{I_{BS}} + \frac{U}{S_C \times R} \times S_{SA} = B \tag{3.13}$$

The graph in Fig. 3.11 shows the outgoing protocol traffic depending on the upload rate and the BandwidthSignal interval using Eq. 3.13. The same input values are used as in Fig. 3.10 with one chunk repetitions.

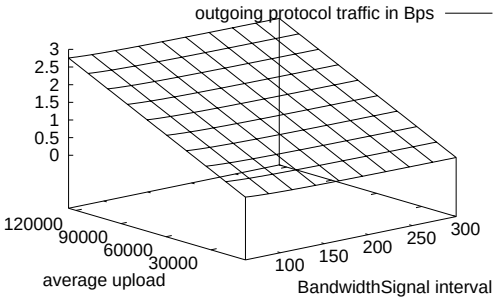


Figure 3.11: Totol outgoing protocol traffic in bytes per second

In dMCFTP, the protocol traffic produced by Data messages is the same as in cMCFTP. Therefore, Eq. 3.9 is also valid for dMCFTP.

Splitting the FileManagementGroup

Because the theoretical maximum incoming protocol traffic is very high with lots of participating peers, the FileManagementGroup can be split according to the upload bandwidth of the peers. Thus, with an upper limit of the upload bandwidth, the maximum incoming traffic on a peer can be reduced.

But, splitting the FileManagementGroup has the same disadvantages as the analog splitting in cMCFTP. First, the swarm is reduced which can decrease performance, also starvation of one group can occur, and a handover of seeds has to be made available. Also the question where set the upper bandwidth limits and how they are dynamically set still needs to be answered. Therefore, splitting the FileManagementGroup is not applicable for dMCFTP.

3.4.3 Multicast Address Management

The same issues of the global address management as described for cMCFTP in Section 3.3.3 exist also in dMCFTP, especially the reservation and allocation of the multicast groups that are used for the FileManagementGroup and the SendingGroups. When using IP Multicast, it is also the best choice to use source-specific multicast groups for the SendingGroups. To use source-specific multicast in dMCFTP, only SendAnnouncement messages need to be adapted to not contain a single multicast address, but a multicast address with the unicast address of the sender. When using Application Layer Multicast, there exist no differences, except that SendAnnouncement messages have to be adapted if an Overlay Network is used, which supports only one group per overlay.

dMCFTP has no FileLeader, thus no local address management is used. But, the peers are still responsible by themselves to globally allocate and reserve multicast addresses used for the SendingGroups.

3.4.4 Security and Cooperation Problems

Almost the same security and cooperation problems of cMCFTP as described in Section 3.3.4 are also applicable to dMCFTP. But using dMCFTP, peers cannot make a source-specific join to the FileManagementGroup to prevent receiving malicious data, because they need to receive SendAnnouncement messages of every peer. In contrast to cMCFTP, a not cooperating peer can still receive the whole file in any case, because peers do not know which chunks other peers have.

Chapter 4

Protocol Implementations for the Simulation

4.1 BitTorrent Implementation

We implemented the BitTorrent (BT) protocol for the network simulator ns-2. The implementation is inspired by the libtorrent library [48] written by Arvid Norberg and released under the BSD License. Mainly, piece/block management and peer connection handling were analyzed and implemented for ns-2.
A simple client and a simple tracker were implemented for the simulations. No extra extensions were implemented, only the core features described in Chapter 2.3.2 and mentioned in [40] were realized.

The tracker is a simple implementation. It stores each peer with its peerId and the address of the peer in a simple list. When it receives a request, it randomly selects peers from its peer list and includes them in its response. The tracker performs no strategy, which can lead to a better peer distribution.
The tracker does not implement the HTTP protocol layer, instead it uses a simple message exchange directly on the TCP layer provided by ns-2. This simplification has no significant impact on download times. But, the BT clients have a little more bandwidth for downloads.

The tracker request message is simplified for our implementation. File and peer identifiers are only 4 bytes long instead of 20 bytes SHA-1 hash values. The address of the peer is included in the message, because this made it easier to implement the address extraction. The port field is not needed for the simulation, although it is crucial for real implementations of BT.

The implemented client does not process a metainfo file. Instead, clients receive all their required data via parameters, which are set at the simulation start.

The BT peer wire protocol is fully implemented as described in Chapter 2.3.2. The implemented peer wire protocol uses the two-way TCP framework provided by ns-2. This TCP framework is almost a full TCP stack implementation and is inspired by the 4.4BSD implementation. But it

simulates no dynamic receiver advertised window. The maximum segment size was set to 536 throughout all simulations.
Piece messages should contain a whole requested block. But, because of some limitations of the TCP message exchange in ns-2, a requested block is split into several piece messages, each containing a maximum of 1024 bytes. This has a big influence on the protocol overhead, which is much smaller in real BT implementations.

A client actively opens new connections to other peers as long it has less than 30 active connections. It rejects new connection attempts from other clients if it already has 55 active connections. These two connection policies were used in the official BT implementation and are also used in all of our simulations. The choking algorithm is implemented as described in Chapter 2.3.3. The piece selection algorithms are sometimes simplified. Termination of already partially downloaded pieces before starting new downloads is implemented as presented in Chapter 2.3.3. Also, the rarest first policy is strictly implemented. But, the piece selection algorithm always selects randomly a piece if it has more than one piece that is considered to be the rarest. Therefore, the random piece selection mechanism is omitted when a peer starts.
A peer switches to the *endgame* mode if only 10 blocks are missing to finish the whole file, instead of the behavior described in Chapter 2.3.3. Pipelining is also implemented as described in Chapter 2.3.3.

4.2 MCFTP Implementation

4.2.1 Overview

This Section presents the implementation of the two MCFTP versions for ns-2. In both MCFTP versions only the parts related to file transfer of a file are implemented in the simulator.
First, the implementation of the core MCFTP protocol features used in both versions is described. This contains sending and receiving of chunks using SendingGroups, the download saturation of the peers and general configuration parameters. Then, the implementation of the version specific parts is explained.

4.2.2 Implementation of Core MCFTP Features

The core features are implemented the same way in both versions. First, we present a brief overview of the implemented components and the parameters influencing them. Then, we show how these parameters influence the performance of the sending and receiving process.

Core Implementation

The processing of the FileDescriptor is also not implemented as the processing of the metainfo file in BitTorrent. This also implies that a chunk is not verified after downloading, and therefore never is corrupted. We assume that every peer sends the correct data in its data messages. Both versions use only one FileManagementGroup (FMG) without any splitting. The implementation of finding the right FileManagementGroup for a given FileDescriptor is also out of the scope of

this thesis. Only the pure file dissemination mechanisms are simulated.
The applications use IP Multicast with a PIM-SM like multicast routing protocol provided by ns-2, which is called **centralized multicast** and is described in 2.6.2. As transport protocol, all peers are using UDP with a maximal UDP packet size of 1400 bytes for the protocol messages transmitted in the FileManagementGroup and the Data messages sent in the SendingGroups. This value is selected such that packets are smaller than 1440 bytes including IP and UDP headers. These packets should be handleable by almost all underlaying networks without any packet fragmentation. The chunk size is set to 262144 bytes for all simulation runs. The simulated file sizes is always a multiple of the chunk size.
Peers have to keep track of their current up- and download rates, because they do not use a reliable transport protocol, which offer a flow control, congestion detection and congestion avoidance mechanisms. Thus, peers are responsible by themselves such that maximum assigned up- and download rates are not exceeded.

Each peer is initialized by some configuration parameters. Table 4.1 shows the parameters that peers of both MCFTP versions have in common with a short description.
Incoming and outgoing protocol traffic is not predictable in advance. Therefore, each peer reserves two percent of its initial assigned transfer rates for the protocol traffic and leaves the rest for the data transfer, which is controllable and thus also predictable. This is controlled by the parameter **bandwidthPayloadUse**.

Parameters	**Description**
peerId	Unique peer identification (in the simulation, this is an integer instead of using a 20 bytes long ID)
uploadBandwidth	Assigned upload bandwidth in kbps
downloadBandwidth	Assigned download bandwidth in kbps
chunkSize	Size of one chunk (in all simulation, the chunk size is set to 256 KB)
chunkQty	Number of chunks in one file, with the file size defined as $filesize = chunkSize \times chunkQty$
bandwidthPayloadUse	Fraction of assigned up- and download bandwidth maximally used for chunk data transfers (set to 0.98 in all simulations)
startSendDelay	Waiting period before starting to send
receiverExtraTimeout	Additional waiting period before receivers not having already finished downloading a chunk but using already expired SGs delete themselves.

Table 4.1: Configuration parameters of peers used in both protocol versions

When a peer has to act as sender for a specific SendingGroup (SG), it creates a **ChunkSender** that inherits the control of sending the specific chunk. The chunk data itself is sent as a sequential stream of simple **Data** messages that consist of the peer ID, the index of the chunk, a

byte offset within the chunk, and the length of the chunk segment sent. The maximal data size included is fixed to 1350 bytes. Therefore, the maximal UDP packet size is never exceeded.

A **ChunkReceiver** is responsible to collect all Data messages arriving at the receiving peer for a specific SG. It handles different tasks: UDP as transport protocol is not reliable and can not guarantee the correct order of the packets arriving. Therefore, ChunkReceivers must be able to reassemble chunks from Data messages. ChunkReceivers can also start downloading at an arbitrary point in the chunk and have to recognize by themselves when they have downloaded the whole chunk. A chunk can be completely downloaded before the SendingGroup's lifetime expires, particularly if the SendingGroup contains repetitions. To not waste download bandwidth, ChunkReceivers expire and are deleted immediately after receiving the whole chunk. Thus, they free the allocated bandwidth. In this implementation, a chunk can not be partially downloaded and finished by another suitable SG. If a SG's lifetime has expired without getting the whole chunk, all data received of that particular chunk is discarded and the chunk's state is restored.

Peers are responsible by themselves to fully saturate their download capacities. Newly received SendingGroups are stored for further usage, ordered by their sending rate, only if a SG contains chunk repetitions. Stored SGs are deleted as soon as a peer has not enough time left to download at least once the full chunk data according to the time SGs were received and depending on their calculated lifetimes.
Peers check if they can join a new SG after deleting a ChunkReceiver or receiving new SG announcements. This checking is also invoked by the below mentioned timer. Peers look through the currently received SGs and the stored SGs starting with the SG with the highest sending rate. Every SG that does not exceed their free download capacity is joined by the peers, but without joining the same chunk twice.

The process of checking if some ChunkSenders, ChunkReceivers or stored SG are expired is either invoked every time a KeepAlive message arrives or by a timer every 0.25 seconds.

Timing of Senders and Receivers

Timing when to start sending the chunk data or deciding how long to wait for the last data messages of one SG is relevant for correct data transfer and for overall download performance.
If the sender would immediately start sending the data after receiving the request to send a specific SendingGroup, some of the peers receive the SG announcement after the sender has already started to send. This issue arises because of different delays in the network. Another factor is join latency of the multicast protocol used, which additionally delays receiving Data messages. Therefore, senders wait a short period before they start to send. The **startSendDelay** parameter controls this latency and is set to 0.5 second in all simulations.

In contrast to the sender, a receiver can not know exactly when the sender has started to send data. Maybe, the receiver has received the announcement earlier as the sender. A Data message

could also be delayed because of congestion in the network. Thus, each ChunkReceiver waits an extra time before it expires and is deleted, but only if it has not already finished downloading the chunk. This **receiverExtraTimeout** parameter must be larger than the *startSendDelay* and is set to 1.5 seconds in all simulations. This gives an additional second to compensate different delays of SendingGroup announcement and Data messages.

These two timing settings influence the download performance. First, the *startSendDelay* timing parameter should be not too large, because in this short period, the reserved upload bandwidth is not contributed to the swarm. This is enforced by SendingGroups with high sending rates. Second, if the *receiverExtraTimeout* is too large, a peer wastes its download capacity with waiting instead of receiving in case the last Data message is dropped. On the other hand, if these two timing parameters are set too small, a potential receiver always joins multicast groups too late, or aborts downloading chunks too early.

4.2.3 Implementation of cMCFTP

In this Section, the core features are extended to have a working cMCFTP swarm. First, the protocol messages between peers and the FileLeader are briefly described. Then, the implementation of peers itself and the FileLeader is described. Although each peer should be able to act as a FileLeader, the FileLeader is implemented as a single dedicated application for the simulation. Thus, FileLeader negotiation is not implemented.

Defined Messages

For the simulation, the four messages listed in table 4.2 are implemented. The data transfer itself uses the already introduced Data messages, which are not anymore listed here. The table shows for each message the different attributes transfered and resulting message sizes.
All messages also contain a static header of 4 bytes. In the simulations, this header is not processed. These messages are not optimized in size.

Message type	**Attributes**	**Size** (bytes)
FullStatus	peer ID, upBw, downBw, uptime, chunk bit-vector	$48 + \lceil \frac{<numberofchunks>}{8} \rceil$
PartialStatus	peer ID, upBw, downBw, uptime, list of chunks.	$48 + x \times 4;\ x \epsilon \{1, 2, 3\}$
Request	peer ID, backoff interval	32
KeepAlive	peer ID, swarm size, list of SendingGroups	$25 + x \times 27;\ x \epsilon \{1, 2, ..., 30\}$

Table 4.2: Messages implemented and used by cMCFTP and their simulated message sizes in bytes (upBw meaning upload bandwidth and downBw download bandwidth)

Peer Implementation

Each peer is started with the configuration parameters listed in Table 4.3 and Table 4.1. Sending and receiving of chunks using SendingGroups is explained in 4.2.2.
The FileManagementGroup (FMG) is not split, therefore each peer receives all Status messages from the other peers. The simulations that use a FMG split into an Update and a Management Group also use only one multicast group, and peers also receive all Status messages. Therefore, the full protocol traffic produced by all peers is transferred on the access-links. But, the protocol overhead with split FMG is determined by simply dropping received Status messages, before these are processed. Thus, these messages are not counted in the bandwidth statistic component of each peer.

Parameters	**Description**
partialStatusMsgTrigger-MinChunks	How many unreported chunks are needed to trigger a PartialStatus message (set to 3 chunks in all simulations)
partialStatusMsgTrigger-MaxWaitingQuotient	Divides the request interval into parts. Between two parts, a PartialStatus message is triggered if at least one unreported chunk is available. (set to 2 in all simulations)

Table 4.3: Peer configuration parameters and their default values

FileLeader Implementation

The FileLeader consists of three parts. The first part handles incoming messages and controls sending of outgoing messages. The second part maintains the information base and the third part creates the SendingGroups by using this information base. The last two parts are controlled by the first part. Therefore, the first part can also be called the main part.

Main Part of the FileLeader
The behavior of the FileLeader is also controlled by some parameters that are provided at startup time of each simulation. Table 4.4 summarizes them with a short description and their default values. Additionally, the two parameters *chunkSize* and *chunkQty* are also needed by the FileLeader. Because of the maximum UDP packet size, each KeepAlive message contains a maximum of 30 SendingGroups.

Information Base of the FileLeader
The FileLeader has an associative container, which contains entries for all active peers. Listing 4.1 shows a peer entry structure. The first block (lines 2 – 5) contains the data the FileLeader directly extracts from Status messages. The next block (lines 7 – 10) is maintained by the FileLeader and contains information about timeout conditions and the validity of the peer. The third block (lines 12 – 14), is used in conjunction with the SendingGroups and their management. The FileLeader maintains how much upload bandwidth it already has assigned to that peer and for how much SendingGroups the upload is used. The last value in this block counts the number

Parameters	Description
requestInterval	At each request interval, the FileLeader sends a StatusRequest message (set to 30 seconds in all simulations)
keepAliveInterval	At each KeepAlive interval, the FileLeader sends a KeepAlive message. (set to 1 second in all simulations)
maxNewSendingGroups	Maximum number of SendingGroups in one KeepAlive message (set to 30 SGs)
sendingGroupExtraTimeout	Extends the lifetime of SendingGroups on the FileLeader. (set to 1.5 seconds)

Table 4.4: FileLeader configuration parameters with a short description and the default values.

of chunks the peer has already downloaded. The last field (line 16) is used to maintain multicast addresses that are exclusively used by the corresponding peer and its assigned SendingGroups.

```
1  struct PeerEntry {
2     double upBandwidth;
3     double downBandwidth;
4     u_int32_t uptime;
5     std::vector<bool> chunks;
6
7     double timeOfLastUpdate;
8     double trackedUptime;
9     bool corrupted;
10    double timeOfLastValidUpdate;
11
12    double uploadBandwidthUsed;
13    u_int32_t numberOfSendingGroups;
14    u_int32_t numChunksHave;
15
16    std::queue<nsaddr_t> addresses;
17 };
```

Listing 4.1: A peer entry struct

To quickly access chunk related information, a second array based structure is used. It additionally holds some pure SendingGroup related information, which is not present in the other structure. In Listing 4.2 an entry of one chunk is shown. Like above the first block (lines 2 – 7) contains information received by Status messages, but in a summarized fashion. Using all peers that request the chunk, the FileLeader evaluates which is the maximum and the minimum download bandwidth available. In contrast, using all peers that have the chunk, the FileLeader evaluates which is the maximum and the minimum upload bandwidth available. The second block (lines 9 and 10) contains information, which the FileLeader maintains for each chunk. First, it counts how much SendingGroups are currently active for this chunk. Second, it stores when it has last announced a SendingGroup for this chunk.

```
1  struct ChunkEntry{
2      u_int32_t     have;
3      u_int32_t     requested;
4      double        minDownloadRate;
5      double        maxDownloadRate;
6      double        minUploadRate;
7      double        maxUploadRate;
8
9      u_int32_t     sendingGroupCount;
10     double        lastSent;
11 };
```

Listing 4.2: A chunk entry struct

The information base also maintains a list of all currently active SendingGroups in the swarm. It tracks the lifetime of SendingGroups and when a SendingGroup has expired, it deletes the SendingGroup. The removal of expired SendingGroups is triggered as first action of the SendingGroup Creation Algorithm.

SendingGroup Creation Algorithm of the FileLeader

The SendingGroup Creation Algorithm returns a list of SendingGroups, which are then sent by the FileLeader. As mentioned above, it returns a maximum of 30 SGs. The SendingGroup Creation Algorithm tries to keep all peers busy. It also tries to distribute the senders of SendingGroups among all peers. But, chunks only available at one peer have the highest priority.

Therefore, the SendingGroup Creation Algorithm maintains a list of such high priority chunks along with the single peer that can provide this chunk. Because of delayed reporting, the chunks are excluded from the list that were included in a SG within the last request interval. These high priority chunks must be disseminated at least to a small number of peers as quickly as possible. Thus, the algorithm tries to create a SG with a sending rate of 80 percent of the maximum available download rate. If the originating peer of a particular chunk has not such a high upload bandwidth, the algorithm assigns the fully reported upload capacity of the corresponding peer as the sending rate. SGs containing high priority chunks have no chunk repetition.

If no high priority chunks exist or if the number of created SGs has not reached the maximum, the SendingGroup Creation Algorithm primarily assigns new SendingGroups to peers that have no SG assigned. The algorithm continuously traverses all peers that have no SendingGroup assigned and picks the next peer. If it does not find any peer that has no SendingGroup assigned, it searches for a peer that has at least $\frac{1}{5}$ of its maximum upload bandwidth not allocated.
Then, the algorithm search for a chunk that this peer can provide and is rare in the swarm. But, rare has different meanings on three different decision levels. In the first level, the criteria of rare means that these chunks were not included in a SG within the last request interval. If no such chunk is found, the algorithm searches for chunks that were not included in a SG within the half request interval. If still no such chunk is found, the algorithm looks at all chunks a peer has. This results in a list of chunks corresponding to the first criteria of rare. In the second level,

the criteria of rare respects the number of active SendingGroups to which the chunk is currently part of. The chunk, which is included the least in a SendingGroup is the rarest. In the last level, the criteria of rare means that a chunk is maximally requested by other peers.
For a better understanding, we present a simple example. First, the algorithm searches for chunks the peer has and that are not included in a SG in the last request interval. Assuming that chunks 13, 20, 56 and 72 meet this criteria, the algorithm selects from this list chunks, which are the least in an active SendingGroup. Chunks 13 and 56 are not included in a SG, but chunks 20 and 72 are. Thus, the algorithm looks at chunks 13 and 56. Chunk 13 is requested by 243 peers and chunk 56 is requested by 281 peers. Therefore, the algorithm chooses chunk 56 as the rarest chunk of this peer.
The second criteria is not really important for chunks, which are the rarest according to the first level, because these chunks are mostly not included in active SendingGroups. But, a SG with a very small sending rate can exist longer than a single request interval. Thus, chunks with no active SendingGroups must be preferred.
The SendingGroup Creation Algorithm limits the sending rate differently depending on the peer chosen. A new SG of a peer, which has not yet a SG assigned, always get the full upload bandwidth of the corresponding peer assigned in the first step. But, the algorithm checks that the chosen sending rate is not greater than the average download capacity calculated from the minimum and maximum available download bandwidth provided by the information base.
A new SG of a peer, which already has SGs assigned, must guarantee that also the peer with the lowest download rate can join some SGs. Thus, the sending rate is limited to the minimum available download bandwidth in the swarm. If the sending rate is not already smaller, the algorithm lowers the sending rate to half of that limit.
SGs created for these chunks always have one chunk repetition. Thus, these SGs have a longer lifetime. Therefore, more potential downloaders can profit from them.

Similar as described in 4.2.2, the FileLeader can not exactly know when the selected peer really starts sending its assigned SGs. If the FileLeader assumes that a particular SG has expired, and therefore frees the allocated resources, the SendingGroup Creation Algorithm could create a new SG for that peer too quickly. When the new SendingGroup announcement arrives at the peer, an old SendingGroup could still be active and could occupy the resources needed by the new SG. The peer must then drop the send request. Thus, the lifetime of SendingGroups maintained by the FileLeader is extended by the **sendingGroupExtraTimeout** parameter to guarantee no send request congestions. This parameter must be greater than *startSendDelay* and must also consider the maximum jitter of KeepAlive messages.

FileLeader without a fixed KeepAlive Interval
The number of active SendingGroups is limited by the fixed KeepAlive interval and the maximum number of SendingGroups in one KeepAlive message. Therefore, an additional version of the FileLeader is implemented, which adapts the KeepAlive interval depending on the number of SendingGroups in one KeepAlive message. The FileLeader is implemented as described above. But, rescheduling the timer that invokes the SendingGroup Creation Algorithm and sending of KeepAlive messages is adapted. Before a KeepAlive message is sent, the FileLeader checks

the number of newly created SendingGroups. If this is above $\frac{3}{4}$ of the maximum SGs allowed, the current KeepAlive interval is divided by two and the timer is rescheduled with the new KeepAlive interval. On the other hand, if the current KeepAlive message contains less than $\frac{1}{4}$ of the maximum SGs allowed, the KeepAlive interval is doubled. Between the two limits, the KeepAlive interval remains untouched. The keepAliveInterval parameter provided at startup is used as initial timer value and also as maximum KeepAlive interval. Thus, the KeepAlive interval will not be greater than one second although the KeepAlive message contains no SGs.

4.2.4 Implementation of dMCFTP

In this Section, the core features are extended to implement a working dMCFTP swarm. First, the two messages and their implementation are explained. Then, the implementation of peers is described. Finally, the SendingGroup Creation Algorithm used by all peers is described.

Defined Messages

In dMCFTP, The SendAnnouncement and BandwidthSignal messages are implemented. The Data message described in Section 4.2.2 is not again described here, but also used for dMCFTP. The implemented SendAnnouncement message consists of the peer ID and the information related to one SendingGroup which again consists of a multicast address, a chunk index, a sending rate and a chunk repetition. Thus, the total message size is 41 bytes.
The BandwidthSignal message only contains the download bandwidth of the originating peer and the peer ID, and therefore has a total size of 32 bytes in the implementation.
These two messages also contain a static header of 4 bytes and are not optimized in size.

Peer Implementation

Table 4.1 shows the peer configuration parameters used by dMCFTP peers. Sending and receiving of the chunks using SendingGroups is explained in Section 4.2.2. The timer that is mentioned at the end of that Section also invokes the SendingGroup Creation Algorithm, which creates new SendingGroups if enough upload capacity is free and if the maximum number of SendingGroups per peer is not reached.

BandwidthSignal messages are sent by peers every 2 minutes. The minimum and maximum download rate limits must also adapt, if the peer with the highest or lowest download bandwidth leaves the swarm. Thus, these two limits are gathered twice, but out of phase. The rate limit currently used is always created from the BandwidthSignal messages arriving at least in the last two minutes and is valid for the next two minutes.

Each peer must allocate some multicast addresses exclusively used for the SendingGroups it provides. Thus, peers maintain a queue, which stores unused multicast addresses. Then, the SendingGroup Creation Algorithm picks the first address from this queue and assigns this address to the newly created SendingGroup. When the corresponding ChunkSender is deleted, the

multicast address is again enqueued. If the queue is empty, the peer requests a new and unused multicast address from the simulator.

SendingGroup Creation Algorithm

The SendingGroup Creation Algorithm is based on history information stored for each chunk by every peer. Therefore, peers maintain some attributes for every chunk. Listing 4.3 shows those attributes. Peers can track how many times a particular chunk was sent and when this chunk was announced the last time (lines 1 – 3). How many times a chunk was sent by other peers is captured in two ways, once as an absolute counter (line 1) and once regarding the last 180 seconds (line 3). The front value of the list is the valid counter, which has counted the SendAnnouncements in the last 170 to 180 seconds. Peers also track how many times they have sent the chunk and when this occurred for the last time (lines 5 + 6).

```
1 u_int32_t                 sendExternCount;
2 double                    lastExternAnnounced;
3 std::list<u_int32_t>      sentExternalCountExtraLongPeriod;
4
5 u_int32_t                 sendCount;
6 double                    lastSent;
```

Listing 4.3: Tracked attributes per chunk

The SendingGroup Creation Algorithm distinguishes three prioritized chunk categories. The most important category contains chunks, which are never sent. The second category contains chunks, which a peer has already sent at least once, but of which the peer has never received any SendAnnouncement sent by any other peer. The last category contains chunks that are already announced by other peers.
When the SendingGroup Creation Algorithm is invoked by a peer, it first checks for chunks in the first category. These chunks are identified by the *sentCount* and the *sentExternCount* values, both containing zeros. Because these chunks must be distributed as quickly as possible, peers try to send these chunks using a high sending rate. The sending rate is set to 75 percent of the maximum observed download rate in the overall swarm. But, most of the peers have a smaller upload rate as the requested one. Therefore, these chunks are sent with the fully assigned upload bandwidth of the corresponding peer. The created SendingGroups contain no chunk repetition. At startup of each peer, all chunks are in the highest category, but this list gets rapidly smaller and after a while, each peer should have accurate history information.
When all chunks of the highest category are sent at least once, the SendingGroup Creation Algorithm looks at the second category of chunks. The peer assumes that these chunks were not received by any other peers. Within this category, the peer starts with the chunk that was last sent the longest time ago. Because these chunks also have a high priority, the SendingGroup Creation Algorithm also assigns high sending rates, but this time only the half of the maximum observed download rate in the swarm. In this category, the algorithm also assigns the full upload capacity of the peer if its upload rate is smaller than the requested sending rate. The created SendingGroups of chunks of the second category also have no chunk repetition.

The chunks in the last category are all available at least by two peers in the swarm. When a swarm already exists for a while, all chunks are owned by this category, and thus the algorithm mostly works on chunks in this category. For this category, the SendingGroup Creation Algorithm must maintain a constant sending availability of all chunks. Therefore, the algorithm uses adapted send count values provided by the *sentExternalCountExtraLongPeriod* list. The algorithm searches for chunks, which were the least sent by itself or by other peers during the last 180 seconds. If more than one such chunk exists, the chunk is selected, which was sent the longest time ago. The SendingGroup Creation Algorithm assigns small sending rates for SGs created at this level. It randomly chooses a sending rate between the half of the minimum observed download bandwidth in the swarm and the upload capacity assigned to the peer. The created SendingGroups have one chunk repetition, because correct receiving of a chunk is more important as fast transmission in this category.

The SendingGroup Creation Algorithm is controlled by some additional configuration parameters passed to each peer at simulation startup. The maximum number of active SendingGroups per peer is controlled by the **maxSendingGroupsPerPeer** parameter and is set to 5 SGs. To prevent fragmentation of the upload rate and to give the algorithm a better chance to find an appropriate sending rate for each SG, the algorithm does not create new SGs if the left over upload rate is smaller as the **minSendingRate** parameter, which is set to 35 kbps. The last parameter, **sendingGroupExtraTimeout**, is used to separate the multicast addresses of SendingGroups over time and is set to 1.5 seconds. This has also the effect that the provided upload rate of each peer is not fully used at any time. But, it makes the protocol more robust against timing errors. This parameter could be set smaller, but for fairness reasons compared to cMCFTP, the same time value is chosen.

Chapter 5

Simulation Scenarios

5.1 Overview

This Chapter describes how the three different P2P applications are used in the simulations. First, we briefly present the simulator used for the simulations. The thereafter following Section looks at the two different topologies used in the simulations and how they are build. Then, the different generated networks of the two topologies are presented with the assigned transfer rates of the network access links and the end-to-end delays between the nodes having applications attached. After this closer look at the networks and their properties, the simulated scenarios are described.

5.2 Simulation Environment

For all simulations ns-2 [32] version 2.31 is used. A small description of the simulator was already given in 2.6. All simulations use the heap scheduler of ns-2, instead of the calendar queue scheduler, which should theoretically outperform the heap structure. But simulation samples have shown that the heap structure is faster. This issue could arise because a lot of events must be rescheduled, deleted and newly inserted, while the simulation runs. The heap structure can delete arbitrary events in the worst case in $\theta(log(n))$, whereas the calendar queue needs $\theta(n)$[49].

5.3 Topologies

The simulation of the 3 different P2P applications run on two different topologies. The *normal topology* models a simple but heterogeneous Internet topology. The *overlay topology* models an overlay network by building a special multicast distribution tree.

5.3.1 Normal Topology

We call the normal topology normal, because it models the Internet. For reduced complexity, a simple backbone network with different delays and attached network trees is built. This should model the heterogeneous Internet in a simplified fashion.

Topology Generation

This topology is randomly built according to parameters that feed random number generators (RNG) or limit the randomly generated values. The basis of every topology generation is the *core router network* that was manually set up. The core router network is always the same for all different generated networks. Figure 5.1 shows the core router network with delays assigned on the links in ms used in all networks for the normal topology. From every core router, a tree of routers is randomly built. Each core router acts as root for one tree. Each tree has a depth, which is randomly chosen from a normal distribution. Also, how much child routers a parent has is randomly selected from a normal distribution. This tree building process results in different trees in depth and number of routers for each core router. Each child router is connected to its parent with a link. The delay for each link is also randomly chosen from a normal distribution. This results in a network consisting of routers. The bandwidth of links in the router network are set very high, that no congestions can occur in the router network.
Simulations running MCFTP applications using the *centralized multicast* routing protocol need a user specified Rendezvous Point. This Rendezvous Point is fixed throughout all simulations and is used for all multicast groups. The core router, which contains the Rendezvous Point (RP) is also shown in Fig. 5.1.

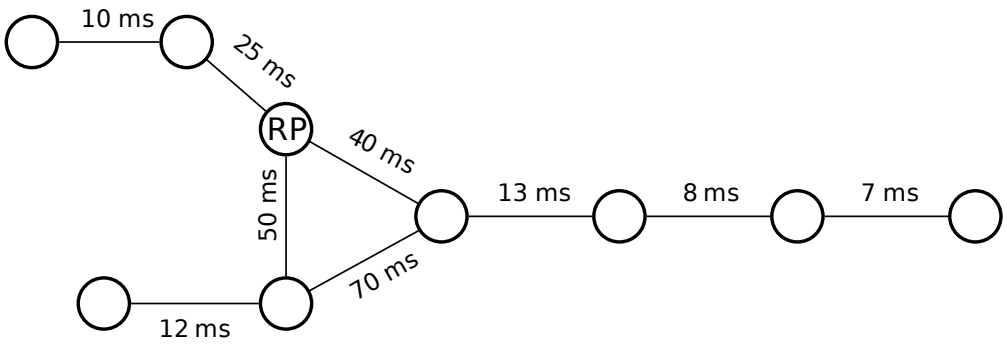


Figure 5.1: The core router network

The nodes containing the applications are then also attached randomly to the generated router network. Not all routers have nodes attached to it. Another parameter controls which routers get nodes attached and which do not. Routers, which get nodes attached are called leaf routers, although they must not be leafs of the generated trees. Then, all such leaf routers get a randomly generated number of nodes attached. The links connecting nodes with their corresponding leaf router are also set up with a random bandwidth and a random delay. But the link delays randomly

generated must be in a range specified with a minimum and a maximum delay. The bandwidths randomly generated must only be greater than a specified minimum. The bandwidth of these links are asymmetric. Today, most Internet access links, like ADSL or cablemodem, have also asymmetric bandwidths. Thus a fixed ratio is specified, which expresses how much smaller the upload bandwidth is compared to the download bandwidth. Table 5.1 shows all parameters used to generate the normal topologies, with a short description.

Parameter	**Description**
TREE_DEPTH_MEAN TREE_DEPTH_STDD	Specify the normal distribution of the RNG to generate the random tree depths.
PEER_MEAN PEER_STDD DELAY_MEAN DELAY_STDD	Specify the normal distribution of the RNG to generate the number of child routers and the delay of the corresponding links.
LEAVE_ROUTER_LEVEL	Controls which routers get nodes attached or not.
NODES_PER_LEAVE_MEAN NODES_PER_LEAVE_STDD	Specify the normal distribution of the RNG to generate the number of nodes containing applications.
NODES_LINK_DELAY_MEAN NODES_LINK_DELAY_STDD NODES_LINK_DELAY_MIN NODES_LINK_DELAY_MAX	The first two parameters specify the normal distribution of the RNG to generate random link delays connecting nodes with their leaf router. The lower two limit the generated delays.
NODES_LINK_BW_MEAN NODES_LINK_BW_STDD NODES_LINK_BW_MIN	Specify the normal distribution of the RNG to generate random link download bandwidths connecting nodes with their leaf router.
NODES_LINK_ASYM_RATIO	Specifies the ratio between available upload bandwidth and download bandwidth.

Table 5.1: Parameters used by the normal topology

5.3.2 Overlay Topology

In Chapter 2.4.1, we described the problem of the not widely deployed IP Multicast availability. Therefore, MCFTP has to run on top of an application layer multicast protocol today. Since additional simulation layers increase the simulation complexity too much, a special topology generation is used. It models a multicast distribution tree built by a multicast overlay network.

Topology Generation

The idea behind this topology generation is to build a multicast tree with respect to the forwarding capacity of each node, which can lead to congestion. Also end-to-end delays are higher compared to the normal topology. We use one shared tree for all nodes and multicast groups in the ALM Overlay Network.

Therefore, we use symmetric bandwidths as opposed to asymmetric bandwidths for the normal topology.

The overlay topology is built with two different node types. A simple data forwarding node only consists of one simple node in the simulator. A node, which also has an application attached to it, consists of two nodes in the simulator. One node is used as the forwarder of the multicast data and the other contains the application. The forwarder and the application node are connected with a link having 0 ms delay and a bandwidth set accordingly to the application running on top. If BitTorrent runs on top, the links have a limiting bandwidth assigned. Thus, the TCP-connections need no additional control of the sending rates. The two MCFTP versions use UDP-flows, which do not support congestion control and congestion avoidance mechanisms. Therefore, MCFTP applications control their maximum sending rates by themselves. The links between the forwarder and the application simulator-nodes have a very high bandwidth assigned, so that no congestion can occur on these links. Figure 5.2 shows an example of an overlay topology containing both node types.

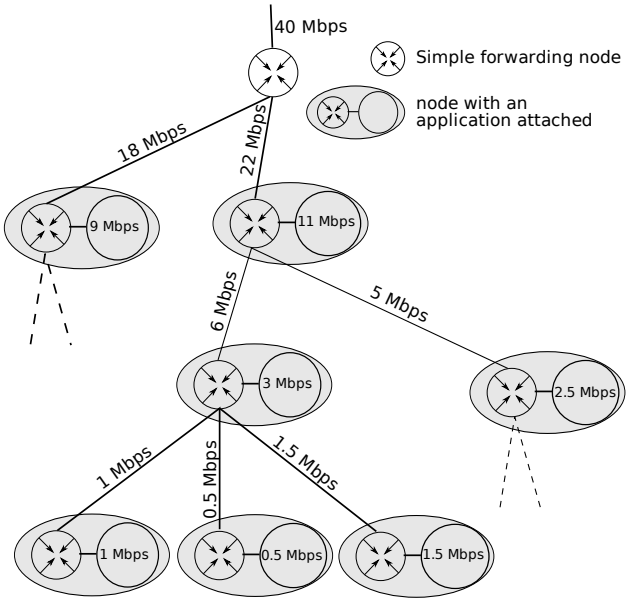


Figure 5.2: Example of an Overlay Topology

The topology generation starts at the root of the tree with a specified start bandwidth. At each node including the root node, the number of children is chosen from the RNG. If the current node is only a forwarder, the whole bandwidth assigned to the link connected to the parent node is randomly distributed among all child links. Because the root node has no parent, the initial start bandwidth is set by the user, which also indirectly controls the number of generated nodes. If the node has also an application attached to it, only half of the bandwidth of the

parent link is assigned to the child links. How much bandwidth the applications can use of the bandwidth provided by the parent link for themselves can be selected with a parameter. If it is set to use half of the bandwidth of the parent link, the bandwidth assignments are optimal and thus no congestion occurs. This guarantees Quality of Service (QoS) for all child nodes regarding bandwidth. If the applications get more bandwidth assigned, the network gets stressed and congestion occur. The applications on the leaf nodes of the tree can consume the whole bandwidth provided by the link to the parent node. Table 5.2 gives an overview of the parameters used to generate the overlay topologies.

Parameter	**Description**
ROOT_NODE_BW	The initial bandwidth at the root of the tree.
TREE_DEPTH_MEAN TREE_DEPTH_STDD TREE_MIN	Specify the normal distribution for the RNG to generate the random tree depths and the minimal generated depth value.
MAX_APPNODE_BW	Controls which nodes are forwarders only and which nodes have a client attached.
MIN_NODE_BW_HINT MIN_BW_PERCENTAGE	Limit bandwidth assignments, so that nodes do not get too low bandwidth values assigned.
PEER_MEAN PEER_STDD PEER_MIN PEER_MAX	Specify how many children a non-leaf peer has in the tree. Must be at least one and can not be more than four.
DELAY_MEAN DELAY_STDD DELAY_MIN DELAY_MAX	Control the delays assigned to the links in the tree.
NODE_BW_SCALE	Specifies how much bandwidth an application can use of its parent link bandwidth. $(0.0; 1.0]$

Table 5.2: Parameters used by the overlay topology

To use this generated tree as a multicast tree, the root of the multicast dissemination tree chosen from the multicast routing protocol must be the same as the root of the generated topology. To achieve this, the root of the multicast tree must be selectable. This can be only achieved by using a PIM-SM like protocol, where the Rendezvous Point can be placed on an arbitrary node. Thus also for the overlay topology and MCFTP applications running on it, the *centralized multicast* routing protocol provided by ns-2 is used. This allows the user to place the Rendezvous Point at any node.

5.4 Generated Networks

With the two described topology generation mechanisms, several networks are built. For the normal topology, thirteen networks with different number of nodes are generated. Nine of these

generated networks are also rebuilt with lower and higher link delays.
For the overlay topology, ten networks are generated differing in the number of nodes. These ten networks are then used once without any congestion occurrence and once with an overloaded network. This can be achieved as described in the previous Section 5.3.2.

5.4.1 Networks for the Normal Topology

With the topology generation mechanism described before, different networks are created that are then used for the simulations. Only a part of the parameters listed in Table 5.1 are also relevant for the number of nodes generated. The nodes are connected with different bandwidths and delays to the router network. But, the parameters specifying the normal distribution of the RNG used to generate the link bandwidth are fixed in all generated topologies. Also, the ratio of the upload to the download bandwidths is fixed. Therefore, the NODES_LINK_BW_MEAN parameter is set to 2000 kbps, the NODES_LINK_BW_STDD to 550 kbps, the NODES_LINK_BW_MIN to 384 kbps and the NODES_LINK_ASYM_RATIO to 0.3.

As mentioned above networks created with the normal topology generation mechanism are also simulated with different link delays. Thus, Table 5.3 presents the number of nodes generated with their corresponding parameter values and the minimal and maximal download bandwidths of the access-links generated. These topology characteristics are all independent of the delays chosen. To compare the applications using different delays, only some of the network sizes between 33 and 732 nodes are chosen. The four networks with 235, 271, 1026 and 2041 nodes are only simulated with medium delays.

The generated networks of the normal topology are characterized except the values of the delays. We present the end-to-end delays gathered from simulation results instead of pure calculations. The packets used to measure these end-to-end delays are kept very small (5 bytes), so that the transmission delay is very small and the results do not differ too much from purely calculated values.

Table 5.4 shows the mean, the standard deviation, the minimal and the maximal end-to-end delays measured in these networks, for the three different delay ranges. End-to-end delays of the networks with medium link delay are moderate and should represent normal delays in the Internet. Although the minimal end-to-end delays of each network are really low and only achievable with directly neighboring hosts, the number of such paths through the networks are small like the high delay paths. The low delay networks can appear in well connected networks like campus networks of universities or internal company networks. The networks with high delays model peers, which are placed around the world and sometimes do not have the best connections to the Internet, and therefore have a wide range of end-to-end-delays.
For a closer look at the existing end-to-end delays in these networks, we refer to appendix B, where the distribution of end-to-end delays is graphically presented for each network. Each bar in the graph shows the number of end-to-end delays falling in that particular 10 ms range.

Number of nodes	TREE_DEPTH_MEAN	TREE_DEPTH_STDD	PEER_MEAN	PEER_STDD	LEAVE_ROUTER_LEVEL	NODES_PER_LEAVE_MEAN	NODES_PER_LEAVE_STDD	minimal bandwidth	maximal bandwidth
33	0.0	0.0	2.2	1.3	1	3.5	1.0	1517	2806
69	0.4	0.4	2.2	1.3	1	5.9	2.0	384	3288
99	0.9	0.8	2.0	1.3	1	4.0	0.7	384	4144
138	0.9	0.5	2.2	1.3	2	5.9	2.0	384	4144
165	0.9	0.5	2.2	1.3	2	7	2.5	384	4144
202	1.5	0.5	2.2	1.3	1	8	2.0	384	4144
235	1.0	0.5	2.2	1.3	2	6.8	2.0	384	4144
271	1.0	0.5	2.2	1.3	2	8.3	3.0	384	4144
304	1.2	0.5	2.2	1.3	2	8.6	3.0	384	4144
511	1.7	0.8	2.2	1.3	2	8.6	3.0	384	4144
732	2.0	0.8	2.2	1.3	3	9.0	3.0	384	4144
1026	2.1	0.9	2.2	1.3	3	8.5	3.0	384	4144
2041	2.6	0.9	2.2	1.3	3	9.0	3.0	384	4144

Table 5.3: The different networks of the normal topology characterized by the chosen parameters and the minimal and maximal download bandwidths (kbps) assigned to applications

5.4.2 Networks for the Overlay Topology

Also, with the overlay topology generation mechanism, different networks are created. Analog to the table characterizing the networks of the normal topology, Table 5.5 gives an overview of the chosen parameter values for the different overlay networks. Some of the parameters are also fixed for all generated overlay networks and thus not shown in the table. These parameters are TREE_MIN set to 2, MAX_APPNODE_BW set to 8.3, MIN_BW_PERCENTAGE set to 0.23, PEER_MIN set to 1 and PEER_MAX set to 4.

Because this topology generation mechanism differs a lot from the one used for the normal topology, the generated bandwidths of the overlay topology vary more than in the normal topology. They are also differently assigned depending if the network should allow congestions or not. Therefore, each generated network shown in Table 5.5 is simulated twice, once with NODE_BW_SCALE set to 0.5 and once set to 0.75. In networks with NODE_BW_SCALE of 0.5, each node has enough bandwidth capacity to forward all data up or down the tree including the data of a potentially attached application. When NODE_BW_SCALE parameter is set to

	Low delays				Medium delays				High delays			
#Nodes	mean	stdd	min	max	mean	stdd	min	max	mean	stdd	min	max
33	40	44	2	78	80	88	3	144	152	166	22	274
69	41	45	2	79	86	95	3	160	156	171	21	278
99	41	45	1	81	92	101	2	170	160	176	21	286
138	38	43	1	81	82	93	2	169	150	166	21	287
165	37	42	1	81	81	92	2	166	148	164	18	284
202	34	38	1	80	79	88	2	153	137	150	16	281
235	X	X	X	X	88	98	2	169	X	X	X	X
271	X	X	X	X	89	98	2	171	X	X	X	X
304	35	39	1	80	79	88	2	166	137	150	16	278
511	40	45	1	83	86	94	2	171	155	171	18	306
732	40	45	1	83	86	95	2	197	159	175	15	312
1026	X	X	X	X	88	98	2	198	X	X	X	X
2041	X	X	X	X	91	98	2	184	X	X	X	X

Table 5.4: End-to-end delays between applications in networks using the normal topology

Number of nodes	37	62	101	130	164	202	236	262	310	525
ROOT_NODE_BW	67	92	155	250	258	315	430	650	621	950
TREE_DEPTH_MEAN	3.6	4.3	5.8	5.5	6.1	5.8	5.7	5.3	5.7	6.6
TREE_DEPTH_STDD	1.5	1.6	1.6	1.4	1.6	1.6	1.6	1.3	1.6	1.7
MIN_NODE_BW_HINT	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.5	2.5	2.5
PEER_MEAN	3.0	3.5	3.5	3.0	3.0	3.0	3.0	3.0	3.0	3.0
PEER_STDD	1.2	1.6	1.6	1.3	1.3	1.3	1.3	1.3	1.3	1.5

Table 5.5: The different networks of the overlay topology characterized by the chosen parameters

0.75, each link is overloaded with 25 percent of its capacity in the worst case, except the links connecting a leaf node. But, the link delays are fixed in both network types. Thus, Table 5.6 shows the two different networks with their mean, minimal and maximal bandwidth capacities of the applications generated in the networks. Also, the mean, standard deviation, minimal and maximal end-to-end delays, which are equal in both networks, are shown.

5.5 Simulation Scenarios

Almost all simulation runs look at the worst case scenario for file dissemination, where initially only one seed exists and over time the leechers arrive in the network. This is called a flash crowd effect. Other simulation scenarios look at modified flash crowd scenarios, where initially more than one seed exists. In all simulations all seeds (including leechers, which have finished their

	NODE_BW_SCALE 0.5			NODE_BW_SCALE 0.75			Delays			
#Nodes	**mean**	**min**	**max**	**mean**	**min**	**max**	**mean**	**stdd**	**min**	**max**
37	1241	138	4094	1861	207	6141	125	135	6	246
62	951	120	3996	1427	180	5995	163	171	6	297
101	1032	132	3518	1549	198	5277	166	174	6	294
130	1192	131	4149	1789	196	6223	225	234	6	398
164	1024	144	3994	1536	216	5992	239	250	6	460
202	997	108	4004	1495	162	6006	225	234	6	413
236	1138	122	4057	1707	184	6086	243	252	6	425
262	1415	134	4021	2123	202	6031	258	268	6	454
310	1237	134	4114	1856	202	6172	267	278	6	524
525	1138	128	4083	1706	193	6125	259	268	6	482

Table 5.6: Assigned application bandwidths in kbps and end-to-end delays in ms of the overlay topology networks

download during the simulation) stay active in the network until the simulation run has finished. The simulations are also run with different file sizes.
Each scenario is simulated with different independent runs. Each run differs in arrival times of leechers, the placement of the seed or seeds, the placement of the Tracker or FileLeader and the initialization of RNGs used by the applications. The different placement of the seed or seeds also implies that they have different transfer rates assigned. The Tracker in case of BitTorrent or the FileLeader in case of cMCFTP are also placed at different nodes for each run. But the Tracker and the FileLeader are always placed at the same node for each individual simulation run and scenario. If dMCFTP is simulated, this particular node that contains the Tracker or the FileLeader stays empty, because otherwise the results would not be comparable.

5.5.1 Normal Topology Scenarios

Two different scenario categories are simulated using the normal topology, the flash crowd scenarios and the scenarios with different number of seeds. All these scenarios have the same simulation start and duration behaviors. Obviously, the initial seed or seeds are started at the beginning of each simulation run. Also, the Tracker or the FileLeader must be started at the beginning of the simulations. But, for each different file size different leecher start periods and simulation durations are selected. The leechers arrive in the network uniformly and randomly distributed among a specified leecher start period. This period starts for all scenarios at the same point of time in the simulation, which is set to two seconds. This is enough time for the seeds to contact the Tracker or FileLeader. Table 5.7 shows the end time point of the leecher arrival period and the whole simulation duration. These simulation parameters are fixed in all scenarios of the normal topology.

File size	Leecher start end	Simulation termination
50 MB	1500s	3500s
100 MB	2000s	5500s
200 MB	3000s	16000s
500 MB	5000s	25000s

Table 5.7: The end point of the leecher start period and the point of time where the simulation is terminated depending on the simulated file size

Flash Crowd Scenarios

The flash crowd scenarios focus on networks with medium end-to-end delays and are simulated with different file sizes. These networks are simulated at least with a file size of 50 MB and 100 MB. The scenarios using a file size of 200 MB and 500 MB are limited to a small number of nodes, because of the simulation execution duration. Table 5.8 shows all simulated scenarios of networks using the normal topology with the number of runs in each scenario. The ”X” in the table is used to indicate that these scenarios were not simulated.

		File size			
		50MB	100MB	200MB	500MB
Number of nodes	33	20	20	20	20
	69	20	20	20	20
	99	20	20	20	20
	138	20	20	20	20
	165	20	20	20	X
	202	20	20	20	X
	235	20	20	X	X
	271	20	20	X	X
	304	20	20	X	X
	511	20	20	X	X
	732	20	20	X	X
	1026	10	10	X	X
	2041	10	10	X	X

Table 5.8: The different flash crowd scenarios of the medium end-to-end delays networks with the number of simulated runs

As mentioned before, some of the networks are simulated with lower and higher end-to-end delays between the applications. Table 5.4 shows the used networks. For each network the file size of 100 MB is chosen, and each is simulated with twenty independent simulation runs.

Scenarios with Different Number of Initial Seeds

Some of the medium end-to-end delay networks are used for these scenarios. Each of these scenarios is simulated with ten independent runs.
The first three scenario types consist of a network with 69 nodes and a file sizes of 50 MB, 100 MB and 200 MB. These three are simulated with 1, 2, 4, 6, 8, 10, 20 and 35 initial seeds. The next three scenario types consist of a network containing 99 nodes and the three different file sizes previously mentioned. These are simulated with 1, 5, 10, 20, 30, 50 and 70 initial seeds. The next scenario types are only simulated with two different file sizes, 50 MB and 100 MB. Once with a network of 202 nodes and 1, 5, 15, 30, 50, 100 and 150 initial seeds. Once with a network of 304 nodes and 1, 10, 25, 50, 100, 150 and 200 initial seeds.

5.5.2 Overlay Topology Scenarios

For the overlay topology, only flash crowd scenarios are simulated. Every network of the overlay topology shown in Table 5.6 is simulated once with a file size of 50 MB and once with a file size of 100 MB for the different scenarios. Each scenario is also simulated with twenty independent simulation runs.
Also, in the scenarios of the overlay topology, the initial seed and the Tracker or the FileLeader are started at simulation start. The leecher arrival period starts at two seconds and ends at the same time points as used for the normal topology scenarios shown in Table 5.7 depending on the file sizes used. But, the simulation durations differ depending on the file size, network type and number of nodes. These simulation durations are all bigger than the durations used for the normal topology scenarios. This is because the minimal transfer rates of the applications are smaller in the overlay topology and in half of the scenarios, congestions can occur. The simulation durations range between 5000 seconds and 7000 seconds for scenarios with a 50 MB file size. For scenarios with 100 MB file size they range between 10000 and 12000 seconds.

Chapter 6

Simulation Results

6.1 Overview

First, the download duration factor and how it is influenced by different topologies is presented. For the normal topology, the impact of different end-to-end delays and different number of initial seeds on the download duration factor is also analyzed. Second, the protocol overhead is analyzed. Then, the leecher-seed evolution and the bandwidth utilization is presented. Furthermore, we look at the overall network load. Then, the load on the FileManagementGroup is compared to the corresponding analytically derived maximums. Finally, we briefly look at a variant of cMCFTP and compare it to the original cMCFTP and also to dMCFTP.

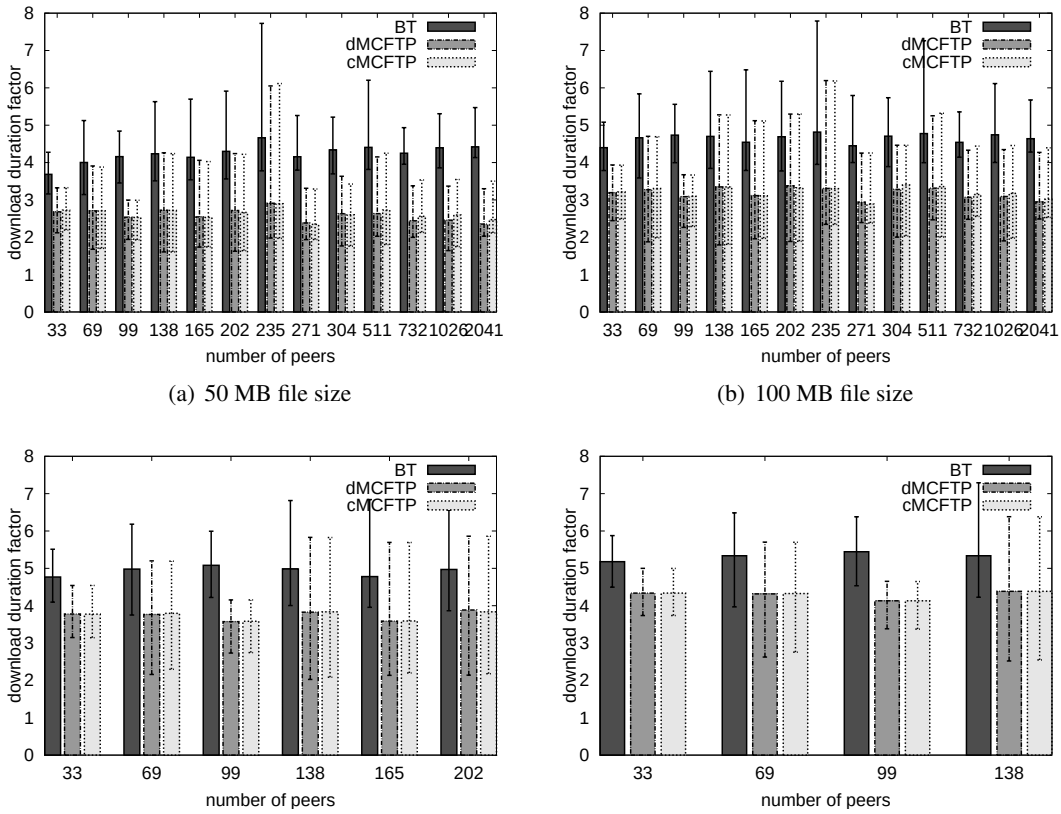
6.2 Download Duration Factor

Because applications can have different upload and download links in terms of bandwidth, different download durations are not directly comparable. Therefore, a factorial representation is chosen to represent the download speed. For each peer, the optimal download duration for the given file size and the peer's full download bandwidth is calculated. Then, the ratio between the actual and the optimal download duration is determined. A strict definition of the download duration factor is shown in Eq. 6.1. This results in a download duration factor independent of the different access links and file sizes, which expresses how much longer a peer needs to download the file compared to its optimal download duration. The smaller the download duration factor is, the faster peers downloaded the file.

$$download\ duration\ factor = \frac{actual\ download\ duration\ (seconds)}{optimal\ download\ duration\ (seconds)} \tag{6.1}$$

6.2.1 Normal Topology

In this Section, we look at the flash crowd scenarios with medium end-to-end-delays presented in Section 5.5.1. The graphs in Fig. 6.1 show the download duration factors. The upper end of a box represents the mean over all simulated download duration factors for a given scenario. The additional "errorbar" represents the minimum and maximum mean download duration factors



(a) 50 MB file size

(b) 100 MB file size

(c) 200 MB file size

(d) 500 MB file size

Figure 6.1: Mean, minimum and maximum download duration factors

achieved in one simulation run of the scenario.

First of all, both MCFTP versions are always faster than BitTorrent. Not only the mean download duration factors of the MCFTP versions are always lower, also the maximum and minimum download duration factors are lower compared to BitTorrent (BT). Otherwise the two MCFTP versions behave almost similar.

When comparing the download duration factors in relation to the number of peers, the duration factors change, but there is no trend for the increase or decrease. Therefore, this must be influenced by the underlaying generated network topology, because the development is independent of the application – MCFTP or BT. It is also similar when comparing the development in relation to the number of peers but with different file sizes. This influence of the topology on download duration factors is not negligible. When looking at the mean download duration factors of BT with 33 nodes and 235 nodes in Fig. 6.1(a), the two values differ by almost 1. The influence of the topology on the download duration factor is lower for MCFTP. For instance in

Fig. 6.1(a), the download duration factors of cMCFTP with 235 nodes and 271 nodes differ by only 0.5.
For each simulation, the initial seed is randomly placed on a node, which has different link capacities in terms of bandwidth. The maximum upload bandwidth of the initial seeds also influences the download duration factors of the leechers. Therefore, Fig. 6.2 shows reciprocals of mean, minimum and maximum upload bandwidths of the initial seeds for each scenario with a file size of 50MB. The reciprocal of the bandwidth is displayed to better show the correlation between the upload bandwidth of the initial seeds and the download duration factor. Although, all three applications run on the same networks and the initial seeds are placed at the same nodes, different runs could be eliminated for each application. Thus, Fig. 6.2 shows the mean seed upload bandwidth for all three applications. But, the differences are very small. The different impacts on BT or MCFTP performance could result from the different transport protocols, because they are also more or less influenced by the underlaying network topology. TCP in case of BT is more sensitive on topology changes than UDP in case of MCFTP.

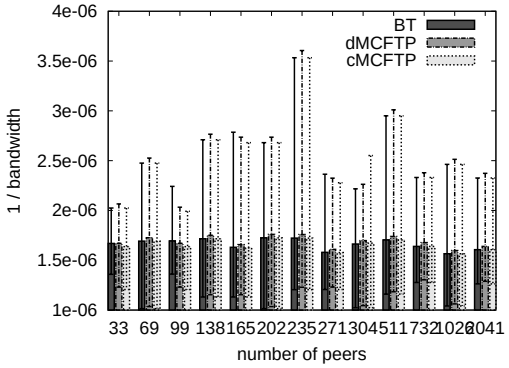
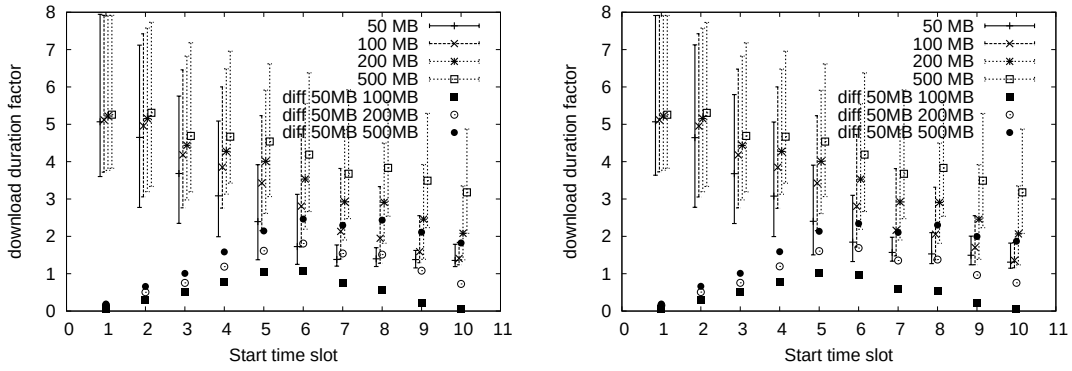


Figure 6.2: Reciprocals of mean, minimum and maximum upload bandwidths of the initial seeds

Both MCFTP versions have a similar SendingGroup Creation Algorithm. First, both algorithms try to identify high priority chunks. In cMCFTP, the FileLeader can actively determine these chunks with a good certainty. dMCFTP peers passively identify high priority chunks with more uncertainty. But, the assignment can only result in false positives and not in false negatives. Also, the bandwidth assignment and chunk repetitions for the SGs are in both MCFTP versions the same. Therefore, both MCFTP versions provide a good and equivalent mechanism to provide SGs for chunks that only one peer has. Then, both SendingGroup Creation Algorithms similarly try to provide as much as possible SGs for the unprioritized chunks. Thus, offering different SGs is almost similar in both MCFTP versions. Therefore, the download duration factors are also almost equal for both MCFTP versions.

When comparing the download duration factors in relation to the file size but with constant number of peers, the duration factors increase with larger file sizes. Thus, Fig. 6.3 shows some

examples of the development of the download duration factors with constant number of peers and increased file sizes. The leecher start time period is split in 10 slices and the mean download duration factors with the minimum and maximum factors of one peer are determined from the peers started in these slices. Additionally, the charts in Fig. 6.3 show the differences between the mean factors of different file sizes. More charts can be found in the Appendix C.



(a) dMCFTP, 33 nodes

(b) cMCFTP, 33 nodes

(c) dMCFTP, 138 nodes

(d) cMCFTP, 138 nodes

Figure 6.3: Sliced download duration factors with the differences between the different file sizes

The download duration factors differ most in the middle slices. In all cases, they do not differ much in the first slice. In the end-slices, differences decrease again. Especially, differences of the download duration factors between scenarios with a 50MB and a 100MB file size show this development. Here, the differences are again very small in the last slice. With larger file sizes, differences still decrease in the end slices, but with larger file sizes, this effect decreases.
With larger file sizes, also the number of chunks or pieces increases. Although, the mean, minimum and maximum download duration factors decrease when peers start later. The peers of the swarms with larger file sizes started in the middle of the leecher start period must wait longer until a needed chunk is announced or available than peers with smaller file sizes. This

insufficient availability and the resulting longer chunk interarrival times can also be seen in the accumulated download rates presented in Section 6.4.2 and in Appendix D.2. With larger file sizes, the graphs of the accumulated download rates oscillate more. Because scenarios with higher number of peers also show this behavior, it can not be only influenced by the number of peers in the swarm.

File size	**Minimal** (sec)	**Last peer started** (sec)
50 MB	703	1500
100 MB	1398	2000
200 MB	2796	3000
500 MB	6990	5000

Table 6.1: Minimal upload duration of the whole file in respect to the end of the leecher start period

Table 6.1 shows the minimal upload duration for the whole file by using the mean upload rate of 600 kbps, and the end of the leecher start period. With a larger file size, the ratio of the minimal upload duration to the leecher start period increases. The mean download duration factors decrease when peers are started later, because the number of offered or available chunks increases. But, with this decrease of download duration factors, the impact of waiting for missing chunks solely offered by the initial seed increases. With larger file sizes, the lesser advanced is the upload of the first copy of the file from the initial seed, when the peers of the middle- or end-slices started. Therefore, the download duration factors increase with larger file sizes. When newly started peers are closer to the point where new seeds become available, then they are less influenced by the missing chunks in the swarm. Thus, the differences of the download durations factors decrease again, when peers of the end-slices start.

	Factor difference	
Scenario	dMCFTP	cMCFTP
69 peers, 100MB–50MB	-0.0233	0.0741
69 peers, 200MB–50MB	-0.0038	0.1783
69 peers, 500MB–50MB	0.0157	0.3267
138 peers, 100MB–50MB	-0.0299	-0.0059
138 peers, 200MB–50MB	-0.0406	0.0705
138 peers, 500MB–50MB	-0.0105	0.1397
304 peers, 100MB–50MB	-0.0296	-0.0091
511 peers, 100MB–50MB	-0.0097	-0.1208

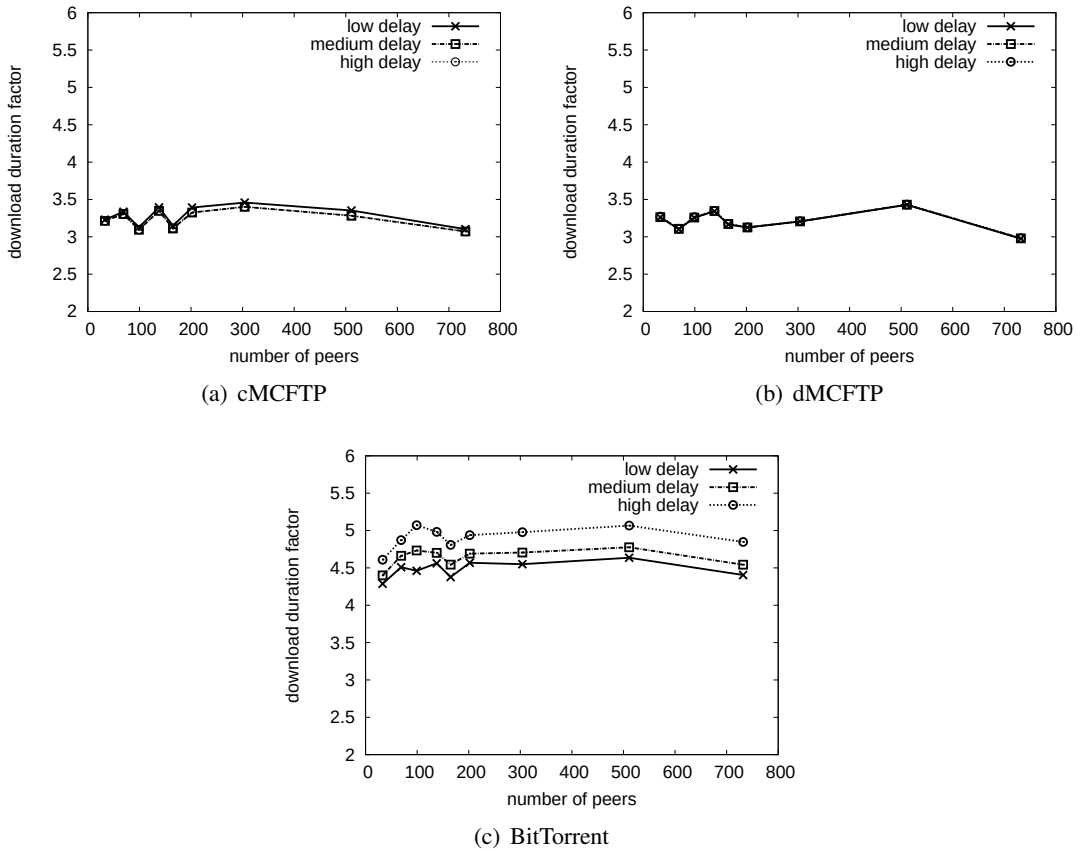
Table 6.2: Differences of the download duration factors with corrected leecher start periods

The download duration factors are influenced by the ratio between the minimal upload duration of the whole file sent by the initial seed and the leecher start period. Thus, Table 6.2 shows the

differences between the download duration factors of different file sizes, where the leecher start periods are adapted to have the same ratio between the minimal upload duration and the leecher start period as with a file size of 50 MB. Because of the simulation run durations, only some examples of MCFTP are shown. As expected, the differences become very small. These differences nevertheless increase with larger files, especially the differences of cMCFTP. Because the increase is smaller in dMCFTP than in cMCFTP and the difference is decreasing with a higher number of peers, these download duration factor differences have to be now influenced by the number of peers and the SendingGroup Creation algorithm, and not by the leecher start period.

Impact of Different End-to-End Delays

In Section 5.5.1, the scenarios with different end-to-end delays are described. The graphs in Fig. 6.4 show the impact of different delays on download duration factors for each application separately, because they are also influenced by various properties.



(a) cMCFTP

(b) dMCFTP

(c) BitTorrent

Figure 6.4: Download duration factors with different end-to-end delays

The results of cMCFTP presented in Fig. 6.4(a) have a strange behavior. Scenarios with low end-to-end delays are slower than scenarios with medium or high end-to-end delays. The other two scenarios have almost the same download duration factors.
The initial seed and the FileLeader both start at the beginning of each simulation and they immediately join the FileManagementGroup. The initial seed also sends its first FullStatus message when it started. Thus, the FileLeader has to join the FMG before the FullStatus message of the initial seed reaches the Rendezvous Point (RP). If the FullStatus message arrives earlier at the RP, then no member joined the multicast group and the message is dropped. Thus, the FileLeader is only informed of the presence of the initial seed later on. In the worst case this occurs almost two request intervals later because the first StatusRequest message could also be dropped at the RP.
Therefore, the lower end-to-end delays are, the higher the chance that the first messages do not arrive their destinations is. This increases download duration factors for all peers that start before the FL knows that there is an initial seed and only then can start to create SGs. Also, subsequently joining peers have increased download duration factors because they can not profit of peers that started earlier. With higher end-to-end delays, the FL is aware of the initial seed before other peers joined the swarm. Thus, scenarios with medium and high end-to-end delays are not influenced by this special timing conditions and have the same download duration factors.

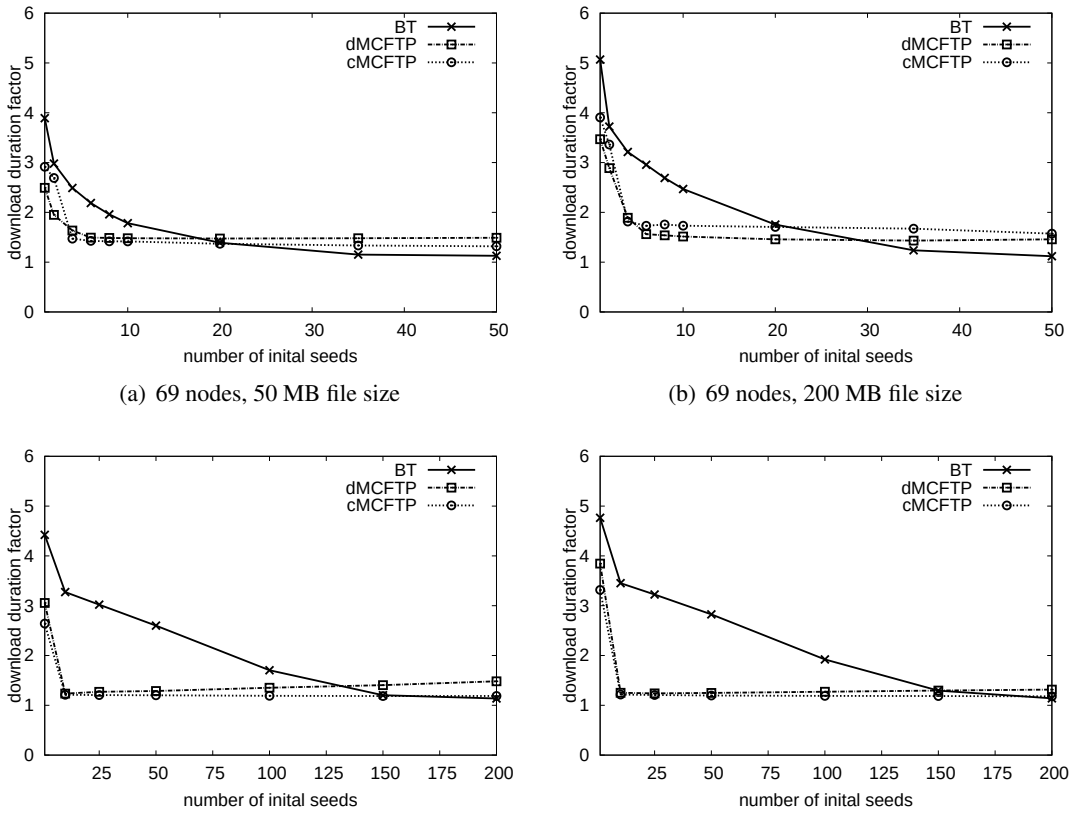
The corresponding scenarios for dMCFTP presented in Fig. 6.4(b) have almost the same download duration factors. In contrast to cMCFTP, the small FMG join delay differences influence each peer individually. But, this delays the download time for this peer by some seconds and does not delay download times for multiple peers by almost a minute. Thus, the download duration factors increase with higher delays. But, this is almost not noticeable.

The results for BitTorrent presented in Fig. 6.4(c) are highly influenced by different end-to-end delays because TCP is used as transport protocol. The maximum TCP throughput of one connection pair is limited by the receivers window size and the round-trip time. Therefore, the higher the end-to-end delays are, the higher the download duration factor is.
MCFTP in contrast uses UDP as transport protocol. UDP does not have a flow-control mechanism like TCP. Whenever congestion occurs, packets are simply dropped. Thus, MCFTP is only influenced by the timing problems previously discussed that result in almost equal download duration factors.

Impact of Different Number of Initial Seeds

The download duration factor of each peer depends also on the number of initial seeds. Intuitively, the download factors should be smaller when more peers are initial seeds. To prove the suggested behavior and to compare the behavior of the different applications, some simulation were rerun with different number of initial seeds. Figure 6.5 shows 4 examples of the scenarios mentioned in Section 5.5.1. The other scenarios are shown in Appendix E.

First, all applications show the assumed behavior. As more initial seeds exist in the swarm,



(a) 69 nodes, 50 MB file size

(b) 69 nodes, 200 MB file size

(c) 304 nodes, 50 MB file size

(d) 304 nodes, 100 MB file size

Figure 6.5: Mean download duration factors of the scenarios with different number of initial seeds

peers finish downloading the file faster. But, the behavior of BT differs from the behavior of the two MCFTP applications. BT gets smoothly faster with more peers as initial seeds. MCFTP gets rapidly faster and reaches an obvious minimal download duration factor. This minimal download duration factor of MCFTP is already reached if only 3-10% of peers in the swarm are initial seeds. BT reaches this minimal download duration factor not until more than 50% of the peers are initial seeds. Actually, dMCFTP minimally constrains itself when more initial seeds exist with small file sizes. With larger file sizes, this gets reduced or reversed, because it can nevertheless profit of its higher number of concurrent SendingGroups.

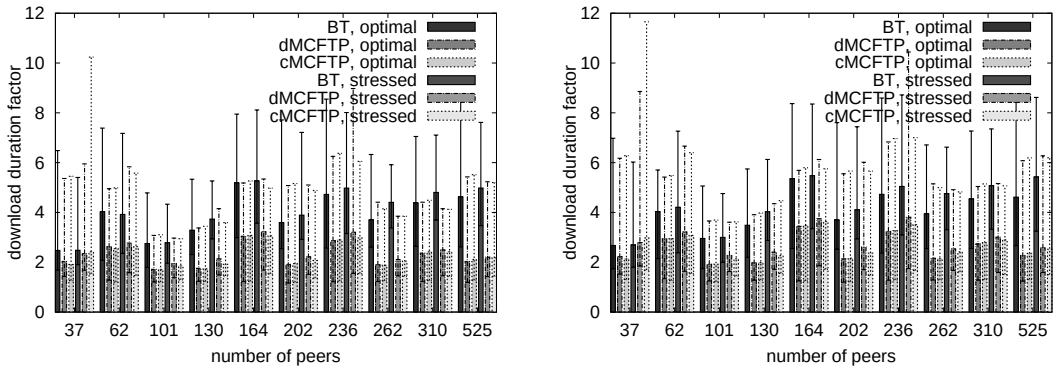
This rapid drop of download duration factors of MCFTP with higher number of initial seeds is a big advantage. In swarms that exist for a long time, it is more likely that the number of seeds ranges between 10% and 50% than between 50% and more. In this state, MCFTP is already near the optimal download duration factor, while BT is still above a download duration factor of two.
BitTorrent's minimal download duration factor is always smaller than those of MCFTP. But,

especially cMCFTP gets very close to BT's performance with higher number of peers in the swarm. And this almost optimal download duration factor of BT should be also achieved with MCFTP. With lower number of peers, the number of concurrent SGs is limited by the number of peers. Therefore, it will be hard to lower the minimal download duration factors in those cases. But, with higher number of peers, the SG Creation algorithm can be adapted to create SGs for unprioritized chunks with more repetitions. This has two advantages. First, with the longer life-times of SGs, the SG Creation Algorithm can produce more concurrent SGs, which raises the chance for a peer to find and join an appropriate SG. Second, a peer has more time to join a required SG. This results in shorter waiting times, where a peer waits for an announcement of a required SG.

Let us return to the increase of the download duration factors with larger file sizes. With more initial seeds, the minimal upload duration of the whole file drastically decreases. This should also decrease the differences between download duration factors for different file sizes. Therefore, the charts of Fig E.2 in Appendix E show these differences between download duration factors. As assumed, they decrease when more initial seeds exist.
In case of cMCFTP, the higher the number of peers is, the smaller these differences are. With 304 peers, they even vanish completely. Thus, cMCFTP is nevertheless influenced by the number of peers. But, this must come from the limited SendingGroup Creation Algorithm of cMCFTP with low number of peers. Beyond this limiting number of peers, no differences should occur.
dMCFTP gets even faster with larger file sizes, but this comes from the self-made constraint of dMCFTP with a file size of 50MB. But, when comparing the values in the charts where the graphs have the highest curvature, dMCFTP almost shows the same behavior as cMCFTP.
BT needs much longer to react on the presence of more initial seeds. But, it is not so strongly influenced by the number of peers as MCFTP, because it always reaches almost no differences at the last value of the graphs.

6.2.2 Overlay Topology

In this Section, the results of the scenarios shown in Section 5.5.2 are presented. The results mainly focus on differences of the download duration factors between the optimal forwarding tree and the stressed forwarding tree.
Actually, the topology is created to show the behavior of the two MCFTP applications when using an Application Layer Multicast protocol with an optimal and a stressed multicast forwarding tree. But nevertheless, BT is run on both overlay topologies as well.
In Section 6.4.2, we argue that BT must have equally accumulated upload rates as accumulated download rates. Thus, with the asymmetrical bandwidth capacities on the access links of the scenarios discussed in Section 6.2.1, BT is a little disadvantaged. But, in this Section, the assigned bandwidth capacities are symmetrical. Therefore, BT can theoretically have the same maximum accumulated download rates as MCFTP. Particularly, in topologies with an optimal forwarding tree, BT can be well compared to MCFTP. With a stressed forwarding tree, the download duration factors of BT are only shown for completeness.



(a) 50 MB file size

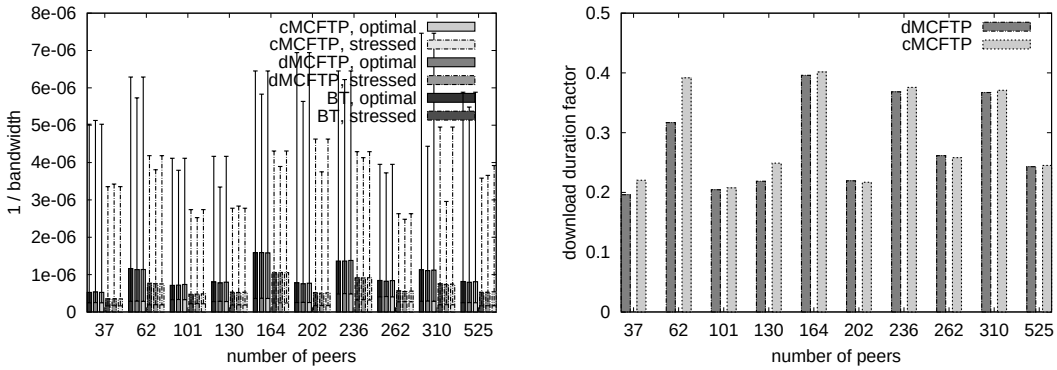
(b) 100 MB file size

Figure 6.6: Mean, minimum and maximum download duration factors of the overlay topology

Figure 6.6 shows the download duration factors for the overlay scenarios. First of all, MCFTP is always faster than BT in the optimal overlay topology. Also, the maximum and the minimum mean download duration factors of MCFTP are always smaller than those of BT. Therefore, MCFTP also beats BT with symmetrical access link bandwidths and not only with asymmetrical access link bandwidths.
On the other hand, the higher end-to-end delays of the overlay topology decrease the throughput of each connection. But, with lower end-to-end delays, BT should nevertheless be slower than MCFTP when taking Fig. 6.4(c) into account.
With the stressed overlay topology, MCFTP is almost always faster than BT. Only one main exception exists for 37 nodes in Fig. 6.6(b). But, this exception occurs by chance. Additionally in 4 cases, the maximum mean download duration factors of BT are smaller than those of MCFTP.
And again, the two MCFTP applications are almost equally fast, but not always equally influenced by the stressed overlay topology.

Along the number of nodes, the download duration factors alter more than in the scenarios of the normal topology. But, the reciprocals of the upload bandwidths of the initial seeds shown in Fig. 6.7(a) also vary more than in the scenarios of the normal topology. Thus, also in the overlay topology, the two charts of Fig.6.6 correlate with Fig. 6.7(a). And, there is also no trend for the increase or decrease of the download duration factors in relation to the number of nodes.

The download duration factors of the optimal overlay topology increase also with a larger file size. The same effect as discussed in Section 6.2.1 is also responsible for this increase. But, this increase is smaller in the overlay topology than the increase in the normal topology. In the overlay topology, the upload bandwidths of peers are higher than in the normal topology because of the symmetrical upload and download bandwidths. Therefore, the upload duration



(a) Reciprocals of mean, minimum and maximum upload bandwidths of the initial seeds in the overlay topology

(b) Differences of the download duration factors between the file sizes in the optimal overlay topology

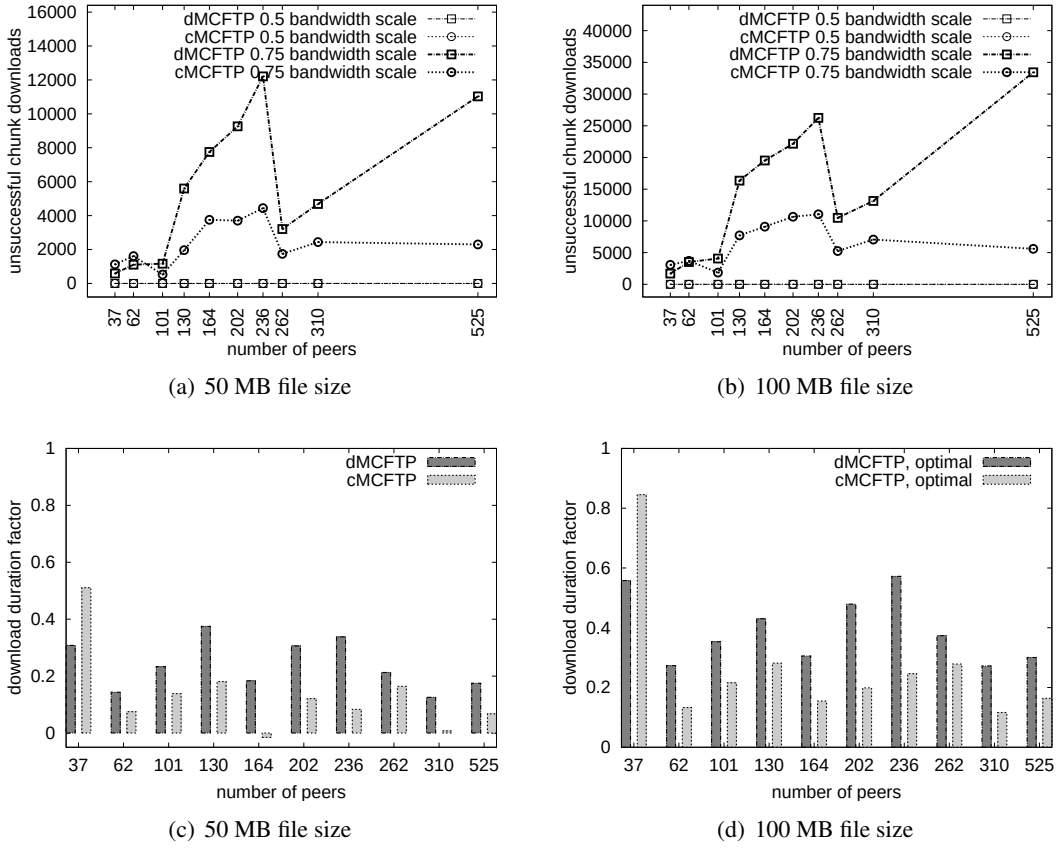
Figure 6.7: Reciprocal upload bandwidths of the initial seeds and the download duration factor differences between the file sizes

of the first copy of the file is shorter. This also decreases the increase of download duration factors with larger file sizes.
Because the mean upload bandwidths of the initial seeds varies more in the overlay topology than in the normal topology, the influence of the upload bandwidths of the initial seeds on the increase of the download duration factors with a larger file size should be visible. Thus, Fig, 6.7(b) shows the download duration factor differences for the simulations of the optimal overlay topology between a 50MB and a 100MB file size. The differences correlate with the reciprocal upload bandwidth of the initial seeds shown in Fig. 6.7(a), as they are supposed to do.

The scenarios with a stressed overlay topology provoke packet drops in the network. This leads to corrupted chunk downloads. These faulty downloaded chunks have to be discarded. Hence, Figures 6.8(a) and 6.8(b) show the number of chunks that can not be successfully downloaded. The two charts in these figures also show the unsuccessfully downloaded chunks of scenarios with an optimal topology. But, as assumed, the optimal topology provokes no packet drops.
The stressed topology is created by increasing the bandwidths of peers by 50%. Sometimes, this leads to the strange behavior of BT being able to download the file faster than in the optimal topology. But, the normal reaction on the stressed network regardless of the higher bandwidths is an increase of download duration factors.

The increase of download duration factors from the optimal overlay topology to the stressed one is shown in Figures 6.8(c) and 6.8(d) for the two MCFTP applications. This increase is not very high, but this increase nevertheless accounts for nearly 10% of download duration factors. dMCFTP is more strongly affected by the stressed overlay topology than cMCFTP. dMCFTP also shows more unsuccessfully downloaded chunks in Figures 6.8(a) and 6.8(b). With its high number of concurrent SGs in conjunction with the used multicast routing protocol, dMCFTP is

disadvantaged. Data from the sources to the Rendezvous Point (RP) is sent independently of a potential receiver. Thus, dMCFTP congests the links with unnecessary traffic towards the RP. This leads to more packet drops and to increased download duration factors.



(a) 50 MB file size

(b) 100 MB file size

(c) 50 MB file size

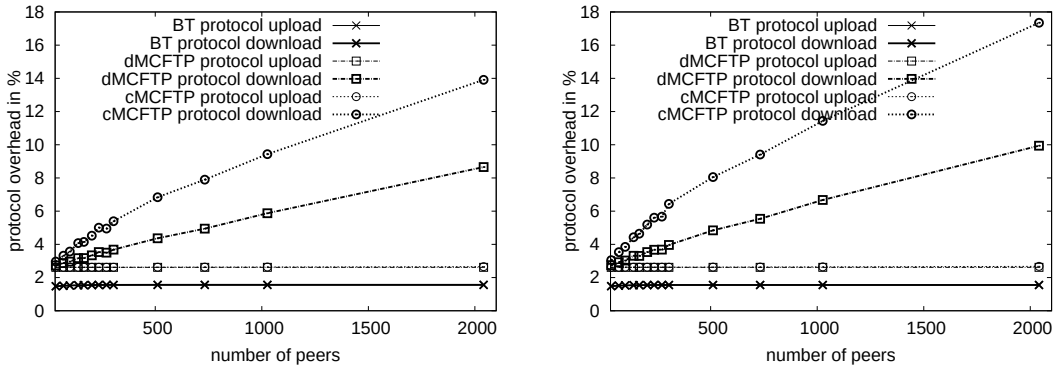
(d) 100 MB file size

Figure 6.8: The number of unsuccessfully joined SendingGroups and the corresponding download duration factor increase.

A disadvantage of MCFTP occurs only by using Application Layer Multicast protocols. As previously mentioned, BT's accumulated upload rate has to be equal to its accumulated download rate. MCFTP can have a higher accumulated download rate than accumulated upload rate. This primarily is an advantage of MCFTP when run on peers with asymmetrical access-links. But, this only works if the packets are replicated on routers and not on end-systems. If an ALM protocol is used, the accumulated upload rate has to be equal to the accumulated download rate, because end-systems also use one-to-one connections between themselves.

6.3 Protocol Overhead

The protocol overhead that is presented in this Section is derived from the scenarios of the normal topology with medium end-to-end delays. The protocol overhead of the three different applications is shown in Fig. 6.9 as a percent value for a file size of 50 MB and 100 MB. Only the effective chunk data of Data messages that are used by MCFTP count as payload data. The static message header, the chunk index, the chunk offset and the length field of Data messages count as protocol data. In case of cMCFTP, StatusRequest, FullStatus, PartialStatus and KeepAlive messages entirely count as protocol data. In case of dMCFTP, BandwidthSignal and SendAnnouncement messages also entirely count as protocol data. In case of BT, all messages that are mentioned in Table 2.2 count as protocol data, except piece message. Similar to Data messages of MCFTP, only the chunk data of piece messages count as payload data. The other fields of piece messages count as protocol data. The protocol overhead presented in this Section does not take the headers of each packet of the underlaying transport protocols into account, which would also count as protocol data. The presented protocol overhead percentages are averages over time, covering the transferred protocol and payload data from the first peer started until the last peer has finished downloading the file.
First, we individually analyze the protocol overhead for each application. Then, the protocol overhead of the different applications are compared with each other.



(a) 50 MB file size

(b) 100 MB file size

Figure 6.9: Protocol overhead in percent

The protocol upload overhead and the protocol download overhead of BT are equal. Because BT uses one-to-one connections between the peers, every uploaded protocol or payload data has to be downloaded by another single peer. Therefore, the protocol upload overhead has to be the same as the protocol download overhead.
Piece messages of BT, which are the only message that carries payload data, can contain a maximum of 1024 bytes payload data, and always contain 13 bytes of protocol data. Thus, the protocol overhead of Piece messages is 1.25%. All other messages consist of pure protocol

data, which further increase the protocol overhead. The protocol overhead measured in the simulations is around 1.53%. Therefore, the other protocol data messages are only responsible for the remaining 0.28%. Depending on the choke/unchoke and interested/uninterested alternations, the protocol overhead could be increased or decreased. But, this increase or decrease is really small, because the pure protocol messages are also very small.
The protocol overhead of BT is not influenced by the number of peers in the swarm. Also, the file size does not directly influence the protocol overhead.
In real BT networks, Piece messages consist of whole requested sub-pieces. Therefore, the overall protocol overhead is reduced to almost 0.3% protocol overhead.

The other two Multicast File Transfer Protocol applications have different protocol upload and download overhead. This results from the multicast "one-to-many" communication property.
For both MCFTP applications, the protocol overhead of Data messages sent and received via SendingGroups is always constant. One Data message that carries the maximum possible payload has 2.6% protocol overhead.
First, we look at the upload protocol overhead of one peer. cMCFTP peers only send Status messages to the FileLeader. This protocol traffic is moderate. Furthermore, as long as leechers are present in the swarm, peers get SendingGroups assigned, and therefore, continuously upload payload data. Hence, the total upload protocol overhead is close to the protocol overhead produced by the Data messages, as presented in the two charts in Fig. 6.9. For the dMCFTP application, the behavior is similar. Peers also send moderate protocol data and continuously upload payload data. Thus, also for dMCFTP, the upload protocol overhead is close to the protocol overhead of Data messages.
Second, we look at the download protocol overhead of one peer. When a peer sends protocol data to the FileManagementGroup, this protocol data is received by all other peers. In cMCFTP, the outgoing protocol traffic on the FileLeader additionally increases the protocol download. Therefore, protocol data received via the FMG is high, as shown in Section 6.6.1. But, the downloaded payload data is equal to or just a little higher than the total file size. Thus, the protocol download overhead has to be higher than the protocol upload overhead.

The upload protocol overhead of both MCFTP applications can be reduced by increasing the payload part of Data messages or by decreasing the protocol part. Increasing the payload part also increases the whole message which can lead to packet fragmentation. Thus, decreasing the protocol part is the better solution. Assuming a minimal header size, a protocol overhead of only 0.6% could be achieved for Data messages. This results in an upload protocol overhead of roughly 1%.

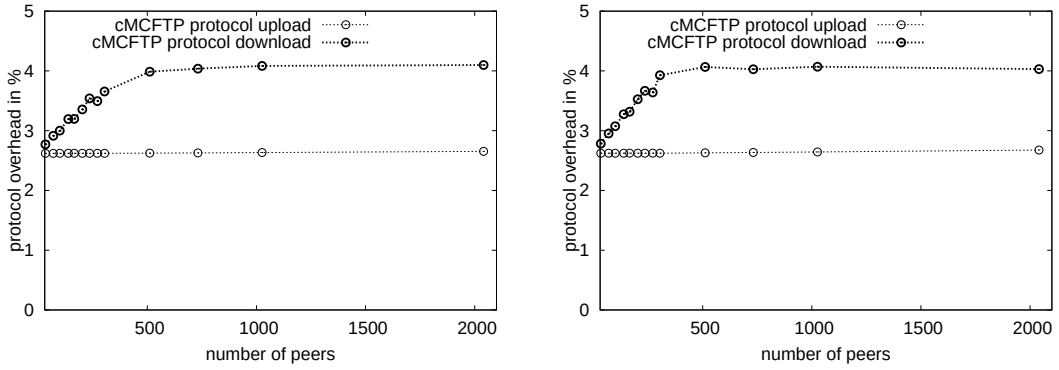
The download protocol overhead of cMCFTP is very high. It increases with the number of peers in the swarm. With more peers in the swarm, more Status messages from other peers are received by each peer. Also, the protocol data originating from the FileLeader additionally increases the download protocol overhead. But, the traffic from the FileLeader reaches its maximum with almost 300 nodes in the swarm. Therefore, received Status messages of other peers are mainly responsible for the high protocol overhead.

The download protocol overhead also increases with bigger file sizes when comparing Fig. 6.10(a) with Fig. 6.10(b). With bigger file sizes, the size of FullStatus messages is also increased. The additional PartialStatus messages should equally scale with the file size. Thus, FullStatus messages are responsible for the increase of the download protocol overhead with bigger file sizes.
To reduce the high download protocol overhead of cMCFTP, we could use the FMG splitting mechanism mentioned as solution one in Section 3.3.2. Figure 6.10 shows the protocol overhead when using the FMG splitting mechanism. The upload protocol overhead is still the same as in Fig. 6.9. But, when peers do not have to receive Status messages from other peers, the download protocol overhead is much lower, and it does not increase with higher number of peers. It still increases until 300 peers in the swarm, because the outgoing protocol traffic from the FileLeader increases until 300 peers. But, the download protocol overhead reaches its maximum at approximately 4%, because of missing Status messages. Thus, the download protocol traffic does also not increase with bigger file sizes.

The download protocol overhead of dMCFTP is also very high. It increases with the number of nodes in the swarm, which results from the same effect as described above for cMCFTP. It also increases a little with bigger file sizes, because with higher download duration factors, the last peer later finishes downloading the file. The FMG in dMCFTP can not be split as in cMCFTP to reduce the download protocol overhead. Only message sizes could be decreased and the chunk repetitions of SendingGroups could be increased with higher number of peers. Especially, a smaller SendAnnouncement message could drastically decrease the download protocol overhead.

When comparing the protocol overheads of the three different applications, BT always has the lowest protocol overhead. The protocol download overhead is almost acceptable for the MCFTP applications, but still very high compared to BT. BT only transfers protocol data when really needed. MCFTP applications send protocol data even if not needed, and peers receive protocol data they are not interested in. Therefore, these two totally different protocol message flows are responsible for the big difference in the protocol overhead.

In real swarms, the download protocol overhead of the two MCFTP versions is a little lower as shown in Fig. 6.9 . At the end of the simulations, almost all leechers become seeds, which send their FullStatus messages or SendAnnoucements to the FMG anyway. And, this traffic is received by all peers. But, the accumulated download payload is very low, because only some peers are still downloading. This leads to a very high download protocol overhead at the end of the simulations, which also increases the mean download protocol overhead. In real swarms, the ratio of the number of seeds to the number of leechers will never become that high, because nodes that finished downloading the full file will eventually leave the swarm. Therefore, the download protocol overhead should also be a little lower in real swarms.



(a) 50 MB file size

(b) 100 MB file size

Figure 6.10: Protocol overhead in percent with a FMG split

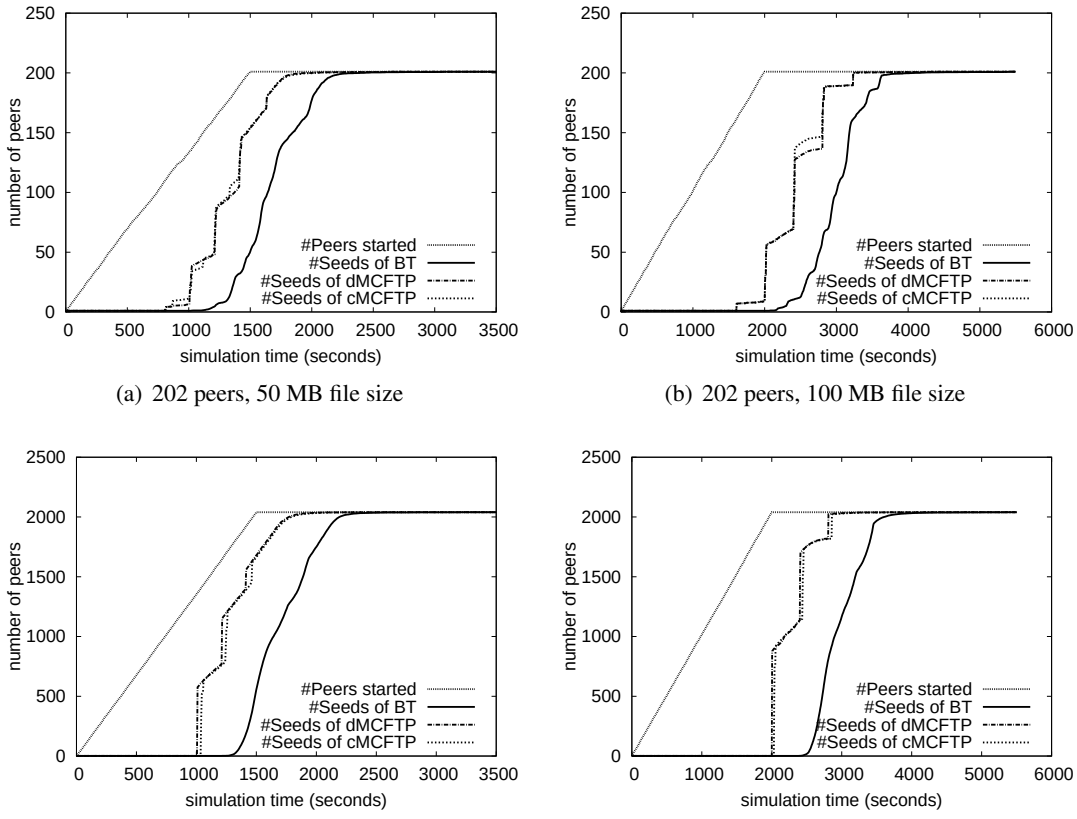
6.4 Leecher-Seed Evolution and Bandwidth Utilization

Leecher-seed evolution and bandwidth utilization are associated with each other. But, they are also interesting when looking separately at leecher-seed evolution or bandwidth utilization. The leecher-seed evolution of the two different file sharing approaches – BT and MCFTP – is different. Also, the bandwidth utilization of the two different approaches is different.
In this Section, we look at the flash crowd scenarios with medium end-to-end-delays presented in Section 5.5.1. Data used for the charts are always mean values of all simulation runs in one scenario. Because peers start uniformly distributed during the leecher start period, the number of peers started and the maximum possible accumulated upload and download rates also increase almost uniformly until all peers have started. Each peer stays active until the end of the simulation. Thus, these graphs also have their highest values until the end of the simulation.

6.4.1 Leecher-Seed Evolution

Figure 6.11 shows the leecher-seed evolution of 4 different simulation scenarios. The charts of the other scenarios can be found in Appendix D.1. Each chart shows the total number of active peers and the number of seeds for each application.
Two aspects are noticeable. First, the two MCFTP applications have almost the same seed-development, and the seed-development of BT and the two MCFTP applications are different. Second, the seed-development of BT is smooth, the seed-development of MCFTP is staircase-shaped.

The almost equal seed-development of the two MCFTP applications has the same reason as stated in Section 6.2.1 for the almost equal download duration factors.
The two MCFTP applications are faster than BT. This can also be noticed in the charts of the



(a) 202 peers, 50 MB file size

(b) 202 peers, 100 MB file size

(c) 2041 peers, 50 MB file size

(d) 2041 peers, 100 MB file size

Figure 6.11: Number of active peers and seeds

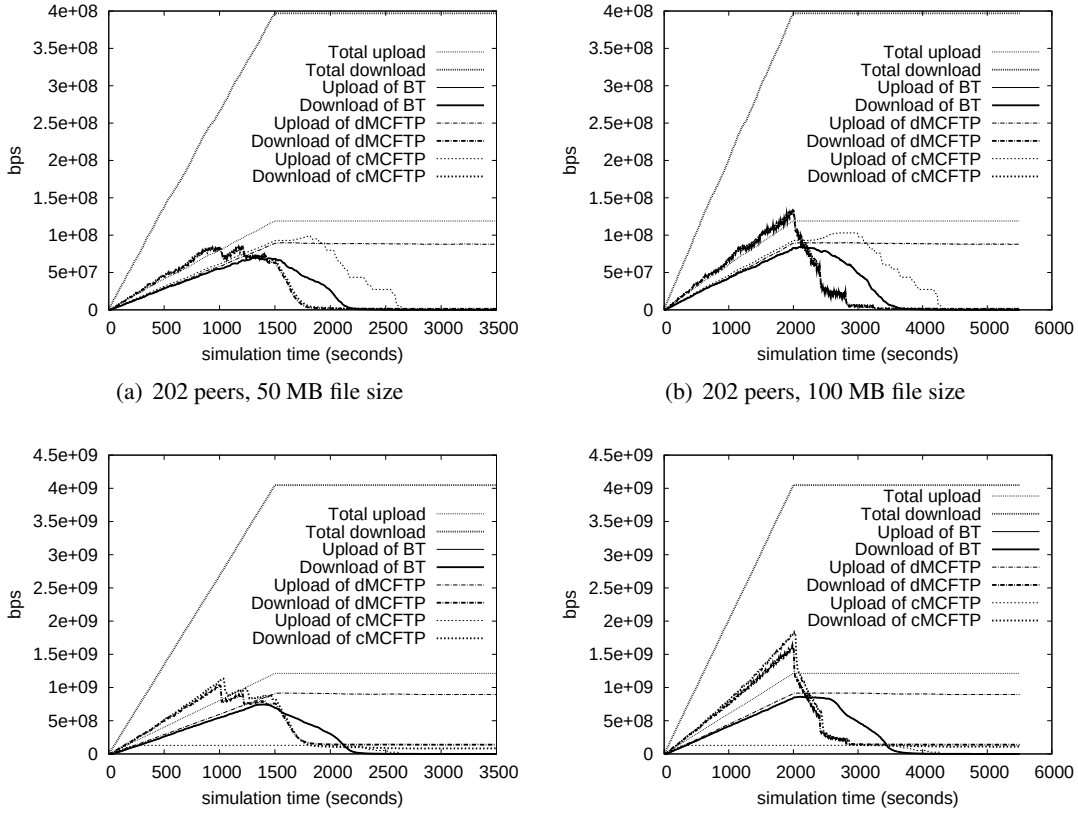
leecher-seed evolution. Thus, BT always has fewer seeds than MCFTP at the same point in time.

The staircase shaped seed-development of MCFTP comes from the multicast property. The SendingGroup Creation Algorithms for both MCFTP applications do not create SGs that have been created in a previous time interval. Therefore, a higher number of peers that are almost finished downloading the file and have the same chunk missing can accumulate. When a SG is created for that missing chunk, all these peers join that SG and finish downloading the file at the same point in time. In Figures 6.11(a) and 6.11(b) we compare the seed-development of dMCFTP and cMCFTP. SGs for this specific case are almost created at the same point in time. In Figures 6.11(c) and 6.11(d), these specific SGs are a little shifted in time. But, peers can also finish downloading the file asynchronously, which can be seen in all four charts as a smooth growth of the number of seeds. Sometimes, the accumulated number of peers differs between the two MCFTP version, because some of the peers have another chunk or only one chunk missing. But, they always catch up with each other. Figure 6.11(a) shows this behavior well.

In BT, peers do not have to wait on certain central offers. Each peer tries to download as much as it can from its connected peers. Thus, each peer finishes downloading the file independent of other peers, which results in a smooth seed-development.

6.4.2 Bandwidth Utilization

The bandwidth utilization shows the positive effect of multicast usage with MCFTP for the first time. Figure 6.12 shows the corresponding 4 charts with the accumulated upload and download rates of all peers for each application along with the maximum possible accumulated upload and download rates. The charts of the other scenarios are shown in Appendix D.2. First, each application is separately analyzed, and then they are compared to each other.



(a) 202 peers, 50 MB file size

(b) 202 peers, 100 MB file size

(c) 2041 peers, 50 MB file size

(d) 2041 peers, 100 MB file size

Figure 6.12: Bandwidth utilization

BitTorrent uses one-to-one connections between peers. This also implies that every upload on one peer results in a download on another peer. Hence, the accumulated upload rate has to be equal to the accumulated download rate. In all graphs of Fig. 6.12 and also in all other simulations, the accumulated upload and download rates are equal for BT.

The total possible accumulated transfer rates are never reached by BT. This is due to the round-trip times of the connecting network and the thereby resulting TCP-throughput behavior. The accumulated transfer rates increase until all peers have started. This can be seen in Figures 6.12(b) and 6.12(d), where all peers have started until 2000 seconds. In these two figures, the accumulated transfer rates increase until that point of time.
In Figures 6.12(a) and 6.12(c), the maximum accumulated transfer rate is reached before 1500 seconds. But, this behavior does not contradict the previous statement, because another effect decreases the accumulated transfer rates before all peers have started. When looking at Figures 6.12(a) and 6.11(a) or Figures 6.12(c) and 6.11(c) respectively, the maximum accumulated transfer rates are reached when the first peers have finished downloading the file. The more peers finished downloading the file, the lower the accumulated transfer rates are. Hence, the transfer rates decrease or stay constant before all peers have been started.
Looking at Figures 6.11(b) and 6.11(d), we see that the first peers have finished downloading the file after all peers have started. Therefore, the accumulated transfer rates approximately stay at the maximum and do not decrease until the first peers finished downloading the file.

MCFTP uses many-to-many connections between peers. Therefore, accumulated download rates can be higher than accumulated upload rates. Because of the usage of the multicast protocol, which also sends data to the Rendezvous Point although no peer has joined the particular multicast group, accumulated upload rates can also be higher than accumulated download rates. As long as accumulated download rates are higher than accumulated upload rates, the positive multicast effect on MCFTP occurs.

In dMCFTP, peers create SendingGroups by themselves and do not know if any peer is interested in that particular chunk. Thus, its accumulated upload rates increase with the number of active peers and stay at the maximum until the end of the simulations. The maximum possible accumulated upload rates are never reached, because of the SendingGroup Creation Algorithm. It does not use all available upload capacity of peers.
The accumulated download rates also constantly increase as long as new peers are started. But, similar to BT, if some peers are finished downloading the file the accumulated download rates decrease.
The accumulated download rates presented in Figures 6.12(a) and 6.12(c) do not constantly decrease after reaching their maximum. They again increase because a large part of peers did not yet start. They increase until the next higher number of peers has finished downloading the file. After reaching the point of time where all peers started, the accumulated download rates constantly decrease. The accumulated download rates presented in Figures 6.12(b) and 6.12(d) constantly increase until all peers started. Then, the accumulated download rates constantly decrease. This is because the first peers finished downloading the file when also the last peers started. Hence, the download rates can not increase again.

The accumulated download rates of cMCFTP experience a similar influence as accumulated download rates of dMCFTP. The only difference is the accumulated upload rates. The File-Leader is limited to disseminate a maximum number of newly created SGs per time interval.

This also limits the maximum number of active SGs independent of number of peers. But, with low number of peers, accumulated upload rates are limited by the number of peers. The charts in Appendix D.2 show that near the number of 304 peers, the limiting factor of accumulated upload rates switches from the number of peers to the limit of the SG Creation Algorithm.
In Figures 6.12(a) and 6.12(b), the values of accumulated upload rates show a small peak before they decrease. These peaks only occur as long as the limiting factor of accumulated upload rates is influenced by the number of peers and not by the SG Creation Algorithms. When looking also at Figures 6.11(a) and 6.11(b), we see that these peaks do not occur until almost all peers finished downloading the file. Then, the SG Creation Algorithm of cMCFTP still can not unfold its full potential, because the number of chunks provided by peers is still too small. With higher number of chunks, these peaks occur earlier and thus they are expanded.

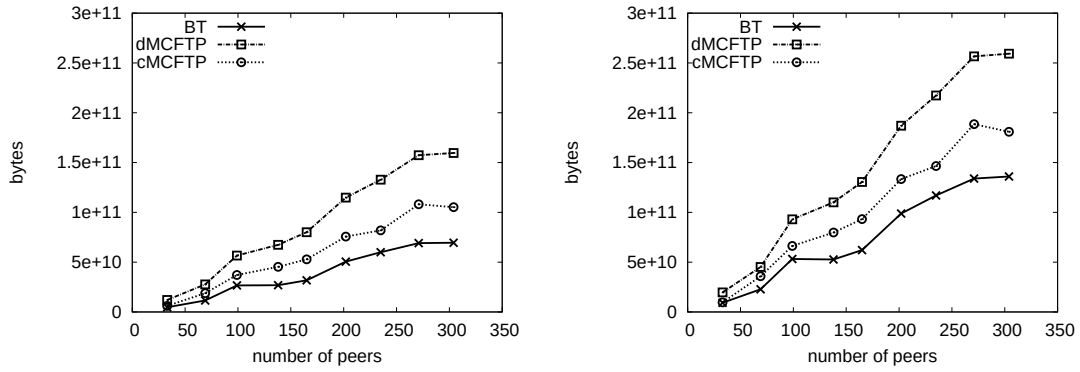
Comparing accumulated upload rates of all three applications, it stands out that they are nearly equal during the peer-start time period for up to 304 peers. But, this is not influenced by the same factors, it simply occurs by chance. For BT, the limiting factor is the normal TCP throughput behavior. For the MCFTP versions, the limiting factor is the SendingGroup Creation Algorithm. The SG Creation Algorithm is also responsible for the similar behavior of accumulated upload rates when using cMCFTP with swarms having more than 304 peers.

When comparing the accumulated download rates of cMCFTP and dMCFTP in Figures 6.12(c) and 6.12(d), download rates of cMCFTP are higher than download rates of dMCFTP. Thus, the resulting download duration factors of cMCFTP should intuitively be lower than the ones of dMCFTP, but they are not as shown in Fig. 6.1(a). When looking at Fig. 6.9, which shows the protocol overheads, cMCFTP has more protocol overhead than dMCFTP. Therefore, accumulated download rates of cMCFTP can be higher than the accumulated download rates of dMCFTP, although dMCFTP is faster.

The great advantage of MCFTP over BT is the previously explained positive effect of multicast usage with MCFTP, especially when comparing accumulated upload rates and accumulated download rates of BT and cMCFTP with higher number of peers. MCFTP can saturate the download of its peers with a small accumulated upload. BT is not even theoretically capable of doing this.
Today, most Internet access links provide lower upload rates than download rates to their users. Therefore, MCFTP can saturate the available download capacities of peers, although upload rates are smaller than download rates. BT is limited by available upload rates.

6.5 Overall Network Load

The overall network load presented in this Section is derived from the scenarios of the normal topology with medium end-to-end delays. For each simulation run, the total transferred bytes on each link that connects two routers are summed up independent of their directions. Then, the mean is built for each scenario and the overall network load is shown in Fig. 6.13.
The two charts only show scenarios until 304 nodes because of the high simulation durations



(a) 50 MB file size

(b) 100 MB file size

Figure 6.13: Accumulated transferred data on the network

when additionally trace link activities in the simulator.

The increase of the network load with more peers is influenced by the number of peers. With higher number of routers, also the number of hops between peers increases.

BT always has a smaller network load than MCFTP. But, MCFTP is disadvantaged because of two reasons.
First, the multicast routing protocol used creates Rendezvous Point (RP) rooted trees and not source rooted trees. Second, the multicast routing protocol routes data from sources to the RP although no group members joined. Therefore, the network load of MCFTP can be drastically decreased. dMCFTP is more disadvantaged than cMCFTP, due to its SendingGroup Creation Algorithm, which creates SGs although all peers have finished downloading the file. This shows some potential to reduce network load of MCFTP when using a better multicast routing protocol.

In fact, the network load of MCFTP can be decreased below the network load of BT, when using IP Multicast with an appropriate multicast routing protocol. In such a setup, SGs with no members produce no additional traffic. SGs with one member produce the same network load as when using simple one-to-one connections with BT, because the path from the source to the destination should also be the shortest path in a source-rooted multicast tree. SGs with more than one member profit of shared paths, where only one packet has to be transferred along this path instead one packet for each receiver.
When MCFTP uses ALM, it produces a higher network load than using IP Multicast. Because packet replication is performed on end-systems and not on routers, each forwarded packet traverses each access-link at least twice. Also, the shortest path from sources to the destinations is not assured with ALM, because such overlay networks use their own routing mechanisms that do not necessarily incorporate the underlaying network properties.

6.6 Load on the FileManagementGroup

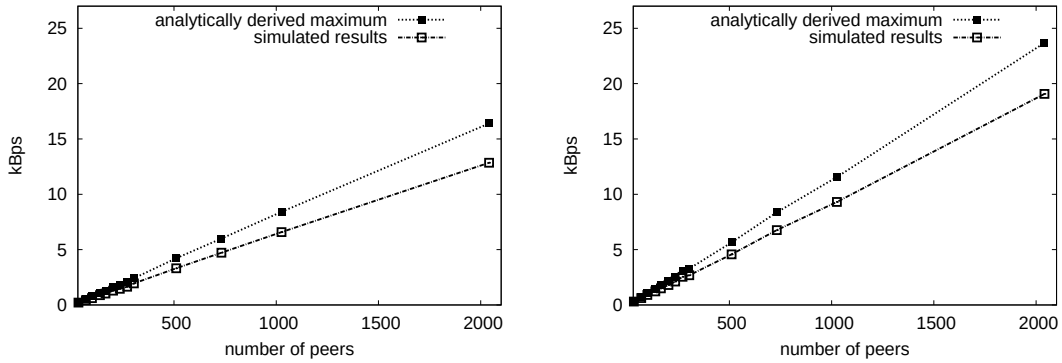
To estimate the load on the FileManagementGroup (FMG), an analytical model is presented in Sections 3.3.2 and 3.4.2. To prove the proposed equations, the actually simulated results of the normal topology with medium end-to-end delays are compared to the theoretical maximums that are derived with these equations.
This section only concentrates on incoming and outgoing traffic of the FileLeader and incoming protocol traffic of peers. The outgoing protocol traffic of peers is not presented, because it is very moderate. The outgoing protocol traffic of peers should never lead to any issues.
Although we speak about incoming and outgoing protocol traffic, only protocol traffic received from or sent to the FileManagementGroup is taken into account in this Section.
Incoming protocol traffic of peers is theoretically equal for all peers. In reality, the incoming protocol traffic is only almost equal. This results from different network delays and from the fact that data arrives in packets at discrete points of time instead of having a constant data flow as assumed in the analytical model. Therefore, the presented bandwidths gathered from the simulation results are always averages over 60 seconds.

6.6.1 Load on the FileManagementGroup of cMCFTP

The results gathered from the simulation runs are compared to the analytically derived theoretical maxima presented in Section 3.3.2. The derived download payload rates used in the equations are determined from the simulation results when the maximum incoming protocol bandwidth is measured. The number of active peers that is also required for some equations is also determined when the maximum incoming protocol bandwidth rates are measured, because at this point of time, not all peers have started.
The sizes of messages are adapted to the sizes currently used in the simulations. This is for FullStatus messages 74 bytes including the bitvector of 201 chunks, or 98 bytes including the bitvector of 400 chunks. PartialStatus messages are 60 bytes in size. The maximum size of KeepAlive messages is 1228 bytes and the size of the StatusRequest messages is 32 bytes.

Incoming Traffic on the FileLeader

The two charts in Fig. 6.14 show the analytically derived maximum incoming protocol traffic on the FileLeader using Eq. 3.3 on page 42 and the mean of the maximum incoming protocol traffic gathered from the simulation results. Figure 6.14(a) shows the analytically derived maximum and the results of the simulations with 201 chunks and a file size of 50 MB. Figure 6.14(b) shows the same but with 400 chunks and a file size of 100MB.
The analytically determined maximum is never reached by the simulation results. This has several reasons. First, a PartialStatus message is not always triggered at half of the request interval, because peers could have no new chunks available to report. Second, PartialStatus messages triggered at half of the request interval must have at least one chunk index fewer than PartialStatus messages triggered by the other criteria. The first two reasons are not taken into



(a) With 201 chunks (b) With 400 chunks

Figure 6.14: Theoretical and actual incoming bandwidth on the FileLeader

account in the equation. And finally, although the used download payload rate is an average over 60 seconds, the download payload rates vary very much.

Outgoing Traffic on the FileLeader

The outgoing traffic on the FileLeader is only influenced by the KeepAlive interval and the request interval, independent of the number of nodes or number of chunks. For all simulation runs, the request interval was set to thirty seconds and the KeepAlive interval was set to one second. In Fig. 6.15. the theoretical maximum and the current simulated results are show.

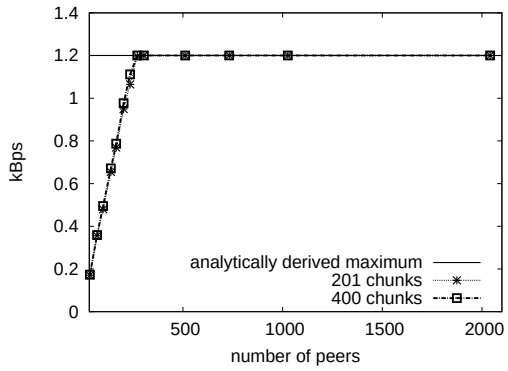


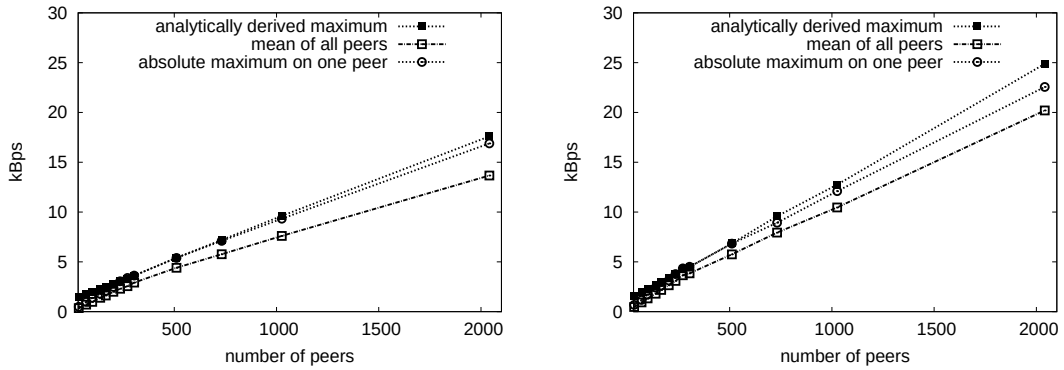
Figure 6.15: Theoretical and actual outgoing bandwidth on the FileLeader

The theoretical maximum is not reached if the number of peers is so small that the FileLeader does not find any SendingGroups it could newly assign. Thus, KeepAlive messages do not contain the maximum possible number of SendingGroups and the actual maximum outgoing traffic is below the theoretical maximum. But, with higher number of peers, the theoretical

maximum matches the simulated outgoing protocol traffic of the FileLeader.

Incoming Protocol Traffic on Peers

Figure 6.16 shows the mean of the maximum incoming protocol traffic over all peers, the absolute maximum protocol traffic received by one peer, and the analytically derived maximum incoming protocol traffic determined with Eq. 3.8 on page 46.



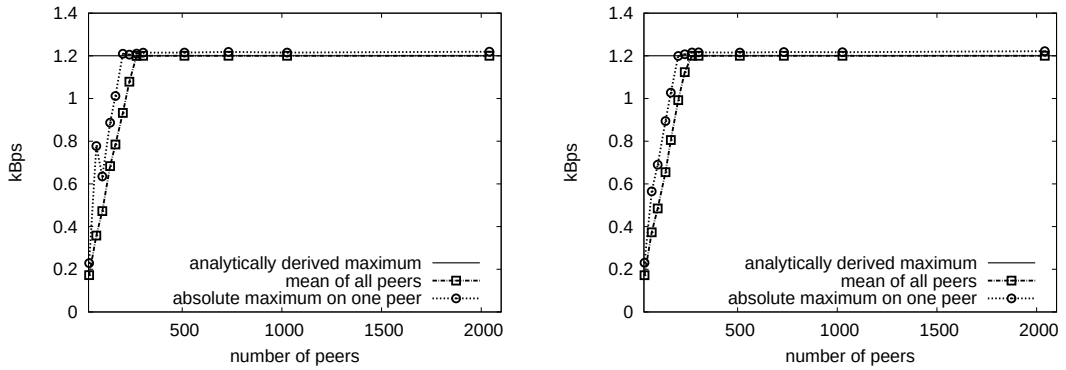
(a) With 201 chunks (b) With 400 chunks

Figure 6.16: Theoretical and actual incoming bandwidth on the peers of cMCFTP

The same statements about incoming protocol traffic on the FileLeader are also valid for incoming protocol traffic on peers. Additionally, when comparing the graphs in Fig. 6.16 with the corresponding graphs in Fig. 6.14, the incoming protocol traffic on peers is always a little higher than the incoming protocol traffic on the FileLeader, as already described in Section 3.3.2.

The incoming protocol traffic on peers is very high. This implies that peers with download bandwidths below a certain limit can not receive anything useful, because the incoming data from the FMG congest their access-links. A swarm of a popular file is quickly joined by 2000 or more peers. For example, a peer with 500 kbps download bandwidth can not really profit of cMCFTP in a popular swarm with a file that consists of 400 chunks. Such a peer has at least 32% download protocol overhead. Therefore, incoming traffic on the peers has to be drastically reduced. On one hand this can be accomplished by reducing message sizes and on the other hand by splitting the FMG.

Thus, Fig. 6.17 shows the theoretical derived maximum, the mean of the maximum incoming protocol traffic and the absolute maximum measured incoming traffic on one peer, when a split FMG is used. This decreases the incoming protocol traffic significantly. Furthermore, the incoming protocol traffic does not exceed a certain limit independent of the number of nodes and the file size. Actually, this enables peers with low download bandwidths to join cMCFTP swarms.



(a) With 201 chunks

(b) With 400 chunks

Figure 6.17: Theoretical and actual incoming bandwidth on the peers, when using a split FMG

6.6.2 Load on the FileManagementGroup of dMCFTP

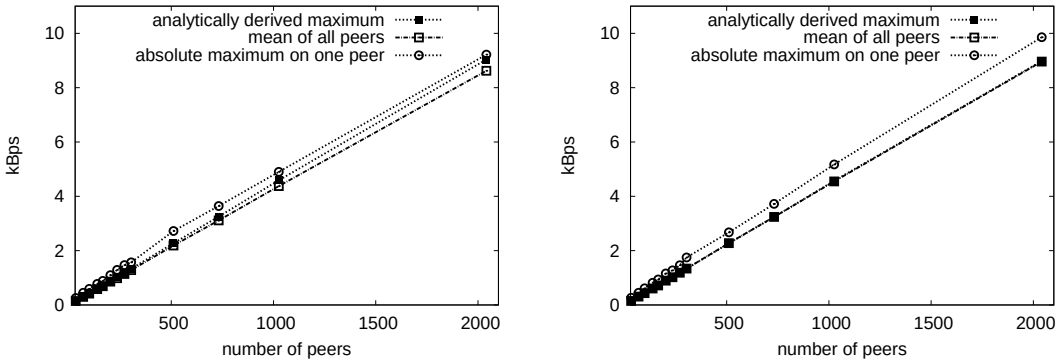
The results gathered from the simulation runs are compared to the analytically derived theoretical maxima presented in Section 3.4.2. The average upload payload rates used in the equations is determined from the simulation results when the maximum protocol bandwidth rate is measured. The number of active peers required by some equations is also determined when the maximum protocol bandwidth rates are measured.
The sizes of the messages are adapted to the sizes currently used for the simulations. This is for BandwidthSignal messages 32 bytes and for SendAnnouncement messages 41 bytes.

Incoming Protocol Traffic on Peers

The two charts in Fig. 6.18 show the analytically derived maximum incoming protocol traffic on the FileLeader using Eq. 3.12 on page 54 and the maximum incoming protocol traffic gathered from the simulation results.

The theoretical maximum is almost reached by the simulation results. This has several reasons. First, the average upload rate does not vary much. As soon as a peer has some chunks, it can immediately create new SGs if an old one expires, and thereby constantly saturates the peer's upload bandwidth. Second, the size of messages is constant and does not alter as when using cMCFTP. Finally, the number of chunk repetitions is set to 1, because newly started peers only create SGs with one repetition, and because all chunks that they have received are at least once external announced. Only the initial seed can create SGs without repetitions.

The incoming protocol traffic of dMCFTP is smaller than the incoming protocol traffic of cMCFTP, but generally still is very high. In contrast to cMCFTP, the FMG can not be split. Therefore, to decrease the incoming traffic on the peers, the size of messages has to be reduced and the number of messages that are sent per time interval has to be reduced, especially with more peers in the swarm.

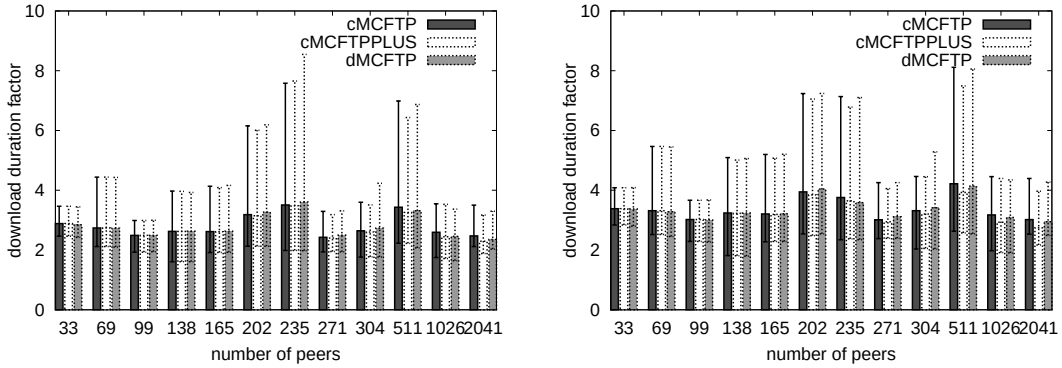


(a) with 201 chunks (b) with 400 chunks

Figure 6.18: Theoretical and actual incoming bandwidth on peers of dMCFTP

6.7 cMCFTP Without a Fixed KeepAlive Interval

In this Section, the impact of the adapted SendingGroup Creation Algorithm described in Section 4.2.3 is presented. The results that are presented in this Section are derived from the flash crowd scenarios of the normal topology with medium end-to-end delays described in Section 5.5.1.
Figure 6.19 shows the download duration factors for cMCFTP with a fixed KeepAlive interval, cMCFTP without a fixed KeepAlive interval (cMCFTPPLUS) and dMCFTP. dMCFTP is also shown because its SendingGroup Creation Algorithm is also not limited to a maximum number of SGs. All presented values are derived from 10 simulation runs without any outliers removed.



(a) 50 MB file size (b) 100 MB file size

Figure 6.19: Download duration factors of cMCFTP with and without fixed KeepAlive interval

Until swarm sizes of 271 peers, no differences can be observed between the download duration

factors of the two cMCFTP versions. Until this number of peers, the limiting factor of the maximum number of SG is not the SG Creation Algorithm, but the number of peers itself as described in Section 6.4.2. Therefore, the effect of the unlimited number of SGs can be first observed in scenarios with higher number of peers. But, the impact is really small and still, dMCFTP is sometimes faster.

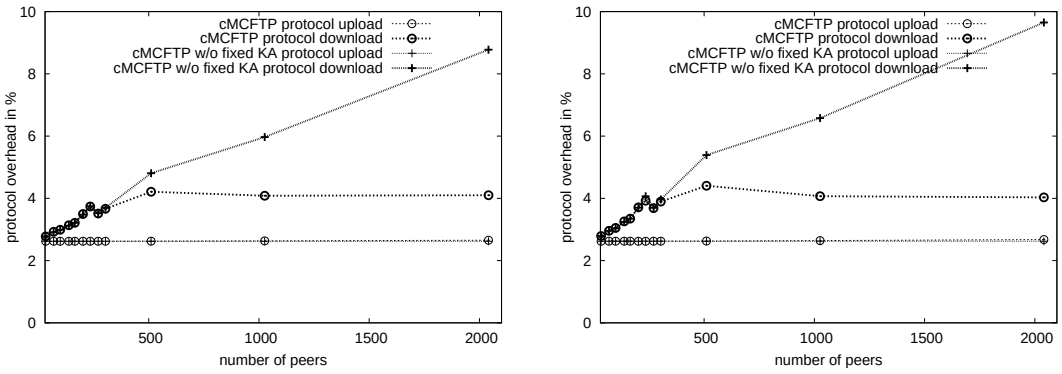
Table 6.3 shows the maximum concurrent number of SendingGroups of cMCFTP, cMCFTP-PLUS and dMCFTP. The change of the limiting factor from number of peers to the SG Creation Algorithm can also be observed in Table 6.3. Although cMCFTPPLUS has slightly more maximum concurrent SGs until 271 peers, the differences significantly increase with more than 304 peers in the swarm. Thus, the maximum number of SG in scenarios with high number of peers shows a big discrepancy between cMCFTP and cMCFTPPLUS or dMCFTP.
Because the download duration factors are not significantly different, the peers can not profit from additionally available SGs in cMCFTPPLUS and dMCFTP. This implies that a lot of SGs are redundant or not needed.

	cMCFTP		**cMCFTPPLUS**		**dMCFTP**	
#Peers	50MB	100MB	50MB	100MB	50MB	100MB
33	27	27	27	27	62	62
69	118	129	118	131	144	144
99	139	184	139	187	204	206
138	197	244	201	251	283	286
165	273	304	276	307	336	340
202	306	345	318	369	416	419
235	326	421	333	442	468	483
271	454	517	514	580	562	563
304	433	488	572	635	627	630
511	293	297	927	1088	1067	1069
1026	293	315	1999	2281	2142	2146
2041	295	317	3783	4439	4231	4242

Table 6.3: Maximum concurrent SendingGroups

This small advantage of cMCFTPPLUS has the disadvantage of a higher protocol download overhead. Thus, Fig. 6.20 shows the protocol overhead of cMCFTP with and without a fixed KeepAlive interval. To better show the impact of the additional traffic originating from the FileLeader, Fig. 6.20 shows the protocol overhead when using a split FileManagementGroup (FMG) as also presented in Fig. 6.10.

The protocol download overhead of cMCFTPPLUS rapidly grows with more peers present in the swarm which is not preferable. Because the protocol download overhead of cMCFTP with a fixed KeepAlive interval of one second already has a maximum protocol overhead of almost



(a) 50 MB file size

(b) 100 MB file size

Figure 6.20: Protocol overhead in percent of cMCFTP with and without fixed KeepAlive interval and with split FMG

4 percent with a split FMG, the solution of having smaller download duration factors must be solved by creating a better SG Creation Algorithm and not by having more concurrent SGs. Also, Table 6.3 and the corresponding observations enforce this statement.
This statement is also valid for dMCFTP, which has the highest maximum concurrent number of SGs but without being significantly faster. For dMCFTP, it is also more important to reduce the protocol download overhead – as shown in Fig. 6.9 – because there does not exist a FMG splitting mechanism as used with cMCFTP.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

We proposed the Multicast File Transfer Protocol (MCFTP) that can run either on native IP Multicast or on Application Layer Multicast. We described and implemented a hybrid (cMCFTP) and a pure (dMCFTP) P2P version of the protocol on the network simulator ns-2. To compare MCFTP, we also implemented the well established BitTorrent (BT) P2P application on the simulator.

The simulated flash-crowd scenarios with 33 to 2041 peers and different file sizes show that the two MCFTP versions are always faster than BT independent of using symmetrical or asymmetrical access-links in terms of bandwidth.
Additionally, we showed that different end-to-end delays do not influence the performance of MCFTP, contrary to the high impact on BT performance.
One of the biggest advantages of MCFTP is its rapid increase of the download performance as soon as more initial seeds are available. Furthermore, MCFTP reaches an almost optimal download performance if only 3-10% of the peers are initial seeds. BT needs more than 50% of the peers acting as initial seeds in order to catch up with MCFTP.
We also showed that MCFTP works well on top of Application Layer Multicast. Even, with a stressed multicast forwarding tree that has to transfer 25% more data than the links can actually support, the download duration only increases by 10%.

But since, coming from totally different protocol message flows, MCFTP has a very high protocol overhead compared to BT. Especially, the download protocol overhead of MCFTP is only almost acceptable, because it increases when more peers are in the swarm. Using cMCFTP, this also occurs with larger files. Furthermore, we showed that the protocol traffic itself received via the FileManagementGroup can prevent peers from joining MCFTP swarms. Therefore, we presented a splitting mechanism for the FileManagementGroup of cMCFTP, which decreases the download protocol overhead and eliminates the dependency on number of peers and file size. For dMCFTP, we could only make some proposals to decrease the download protocol overhead and to keep it independent of the number of peers. The protocol overhead is the single drawback of MCFTP, which must be well observed. Thus, we have also presented

and verified an analytical model, which can be used to determine the protocol traffic of both MCFTP versions in advance.

The adaption of cMCFTP by using an unfixed KeepAlive interval and thereby producing more concurrent SendingGroups has not lead to a significant increase of the download performance. We have come to the conclusion that a pure increase of the concurrent SendingGroups is not the solution to get a better download performance. Actually, it leads to a significant increase of the protocol overhead, which is not desirable. Also, dMCFTP, which has the highest number of concurrent SendingGroups, cannot profit from its high number of concurrent SendingGroups in a reasonable form. Thus, adapting the SendingGroup Creation Algorithms, especially by varying the chunk repetitions and sending rates, should further increase the download performance of MCFTP.

7.2 Outlook

One of the main goals is to further reduce protocol traffic and protocol overhead. For cMCFTP, this can be achieved by splitting the FileManagementGroup, but can be further decreased by shrinking protocol messages. For dMCFTP, the SendingGroup Creation Algorithm could be adapted to reduce the number of SendAnnouncements if more peers joined the swarm. Along with shrinking protocol messages, dMCFTP's protocol overhead can be reduced and would be independent of the number of peers without losing performance.

To implement fully functional applications using both MCFTP versions, some features have to be clarified and implemented.
The underlaying multicast mechanism – native or overlay – must be determined. Also, a possible DHT must be evaluated and implemented, depending on the multicast decision and the thereby resulting address management. To have cMCFTP without a single-point of failure, we would need a mechanism to negotiate a new FileLeader because the old one could leave the swarm. Furthermore, some extensions and enhancements can be integrated in MCFTP to increase security and anonymity of peers, to make the swarm more robust and to extend reliability of chunk transfers. Some proposals to improve MCFTP can be found in Appendix F.

Appendix A

Example of a Metainfo File used by BitTorrent

In this example, we inserted newline and tabulator characters for better reading and understanding. In real, this is one big line. The concatenated SHA-1 hash values are also omitted.

```
d
    8:announce
    49:http://core-tracker.depthstrike.com:9800/announce
    13:announce-list
    l
        l
            45:http://borft.student.utwente.nl:6969/announce
            53:http://core-tracker.enlist-a-distro.net:9800/announce
            49:http://core-tracker.depthstrike.com:9800/announce
            56:http://clients-tracker.enlist-a-distro.net:9800/announce
            52:http://clients-tracker.depthstrike.com:9800/announce
            35:http://www.ooodev.org:6969/announce
        e
    e
    13:creation datei1241519067e
    4:info
    d
        6:lengthi168233096e
        4:name38:OOo_3.1.0_LinuxIntel_install_de.tar.gz
        12:piece lengthi131072e
        6:pieces
        25680:<concatenated SHA-1 values of the pieces>
    e
e
```

Listing A.1: Example .torrent file for an openOffice download

Appendix B

End-to-End Delays

B.1 End-to-end delays in the normal topology

B.1.1 With medium Delays

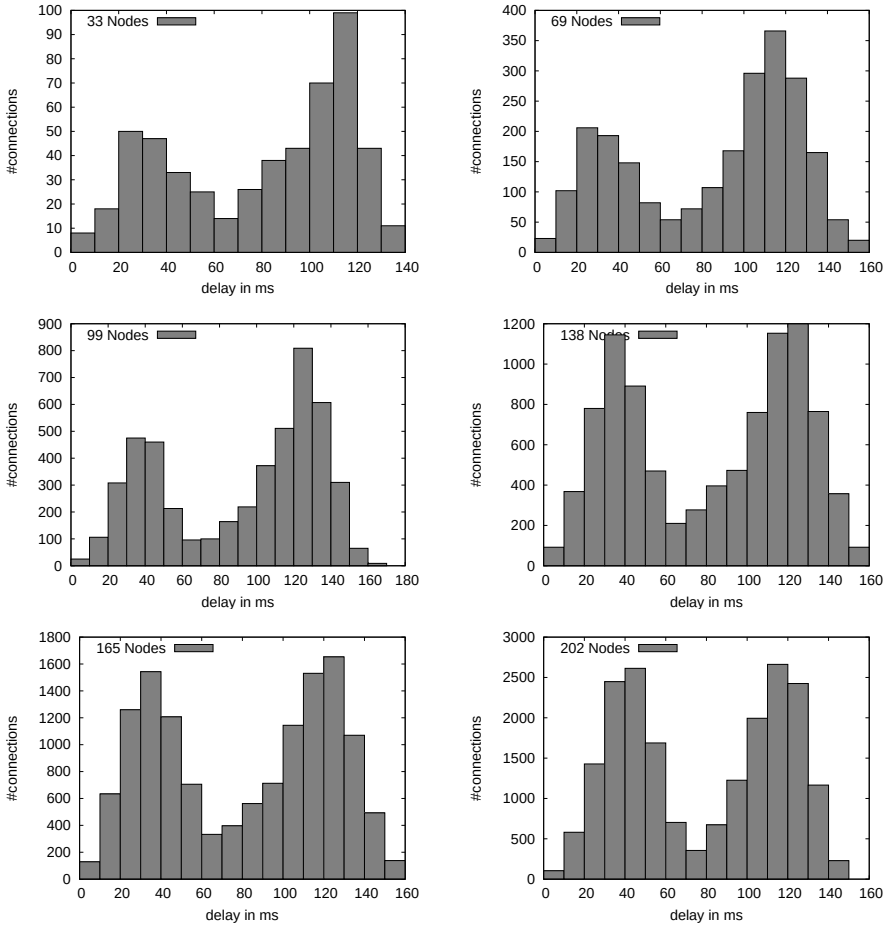


Figure B.1: End-to-end delays of the normal topology I

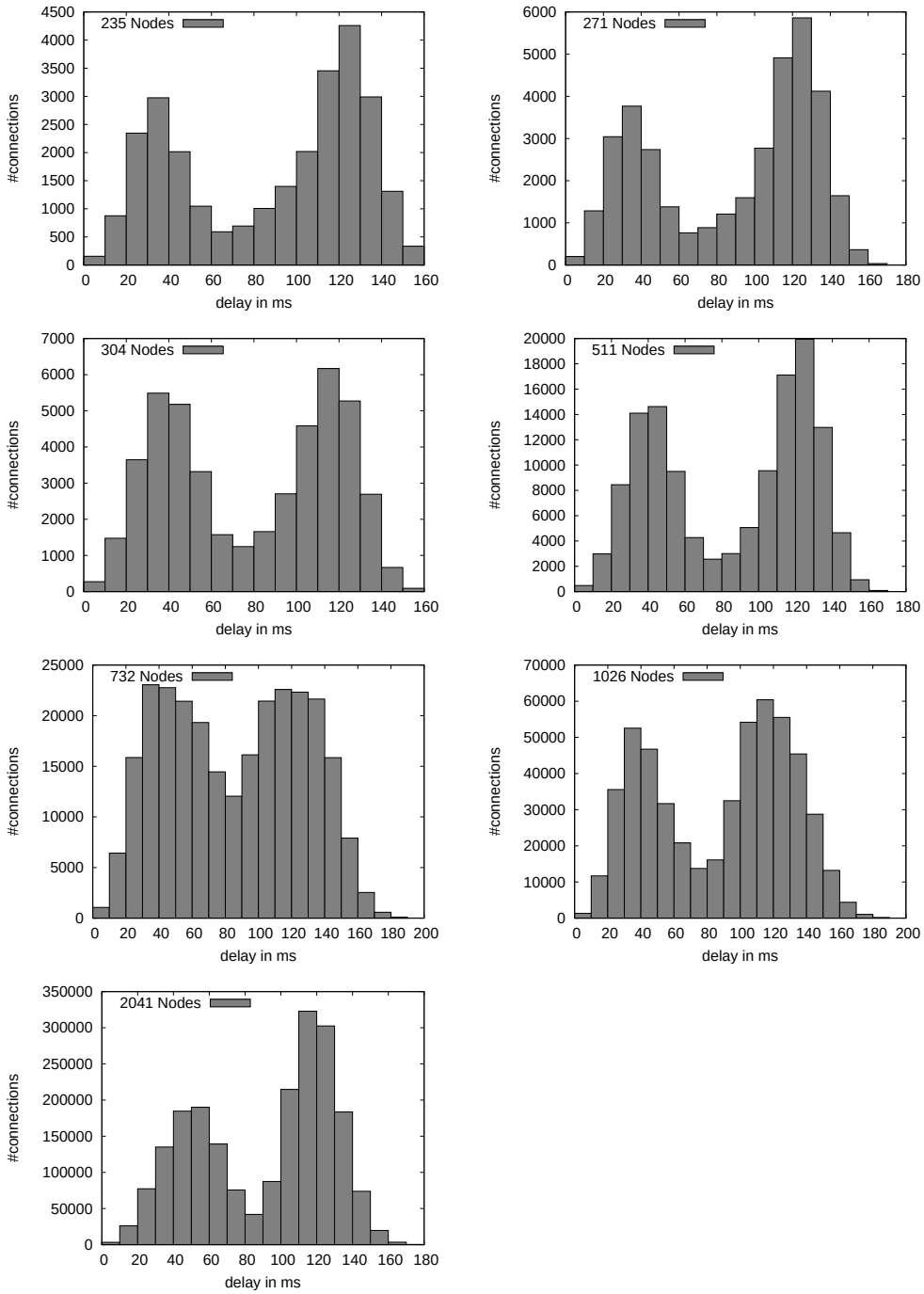


Figure B.2: End-to-end delays of the normal topology II

B.1.2 With low Delays

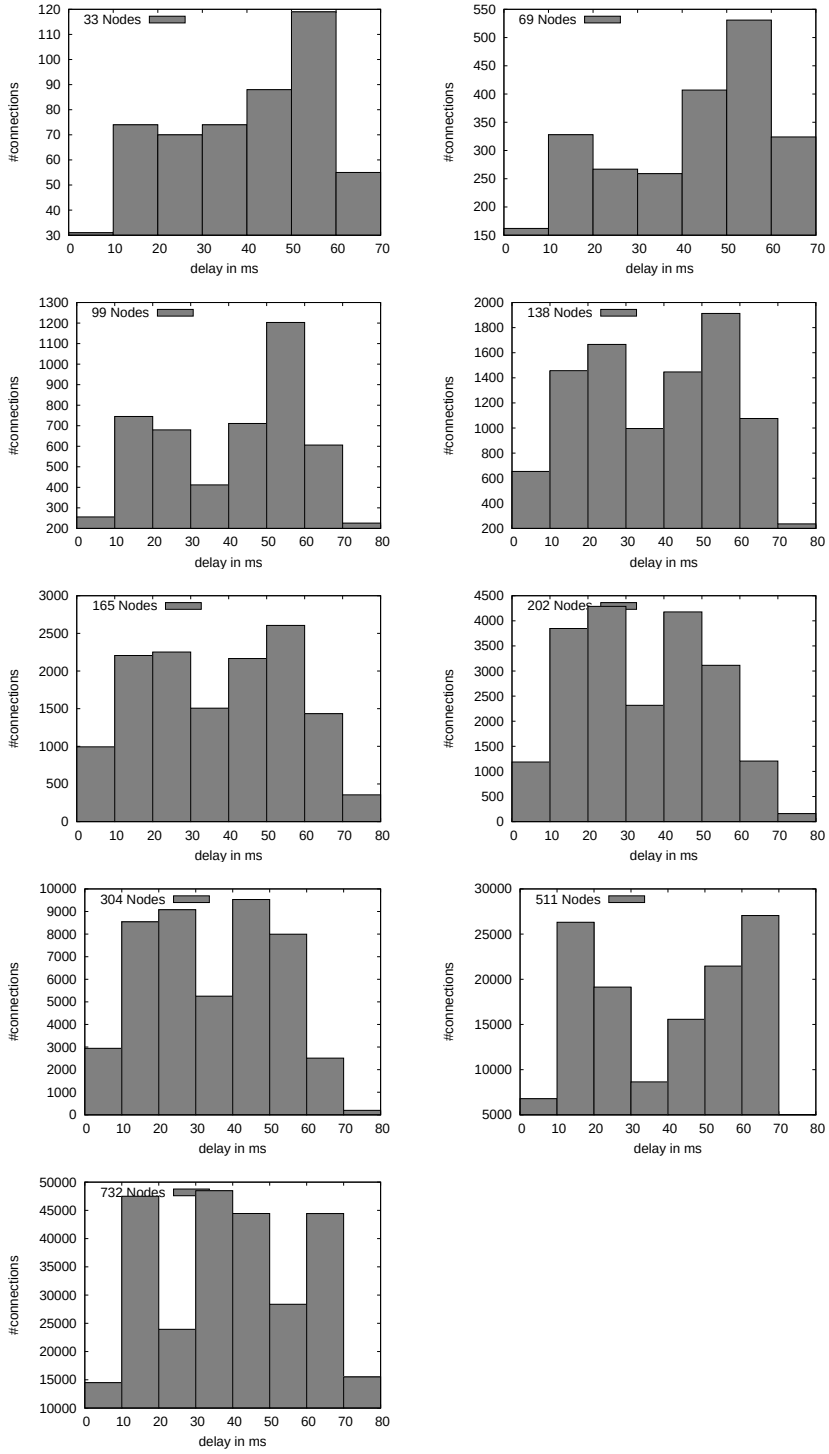


Figure B.3: End-to-end delays of the normal topology, with low delays

B.1.3 With high Delays

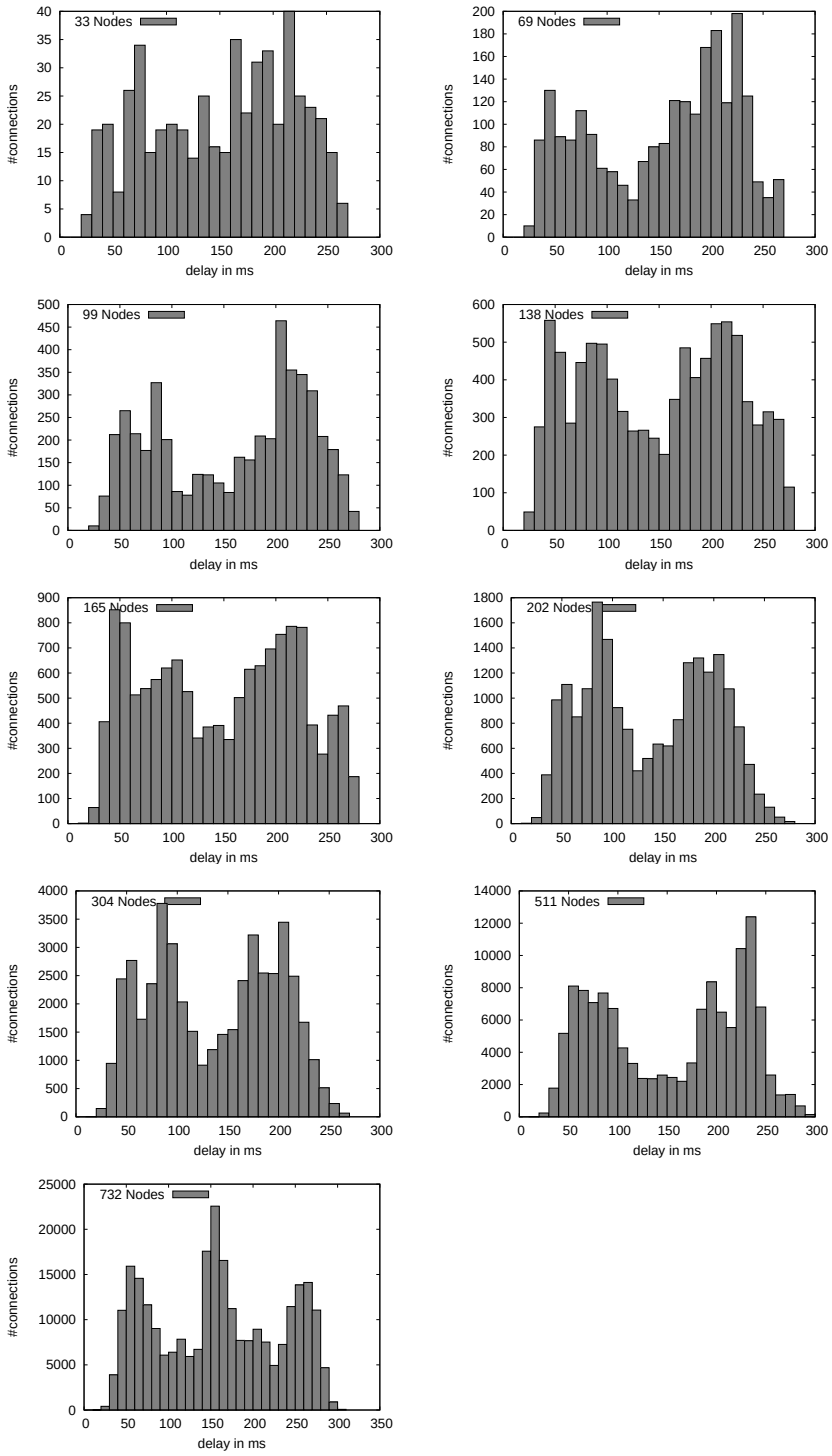


Figure B.4: End-to-end delays of the normal topology, with high delays

B.2 End-to-end delays of the overlay topology

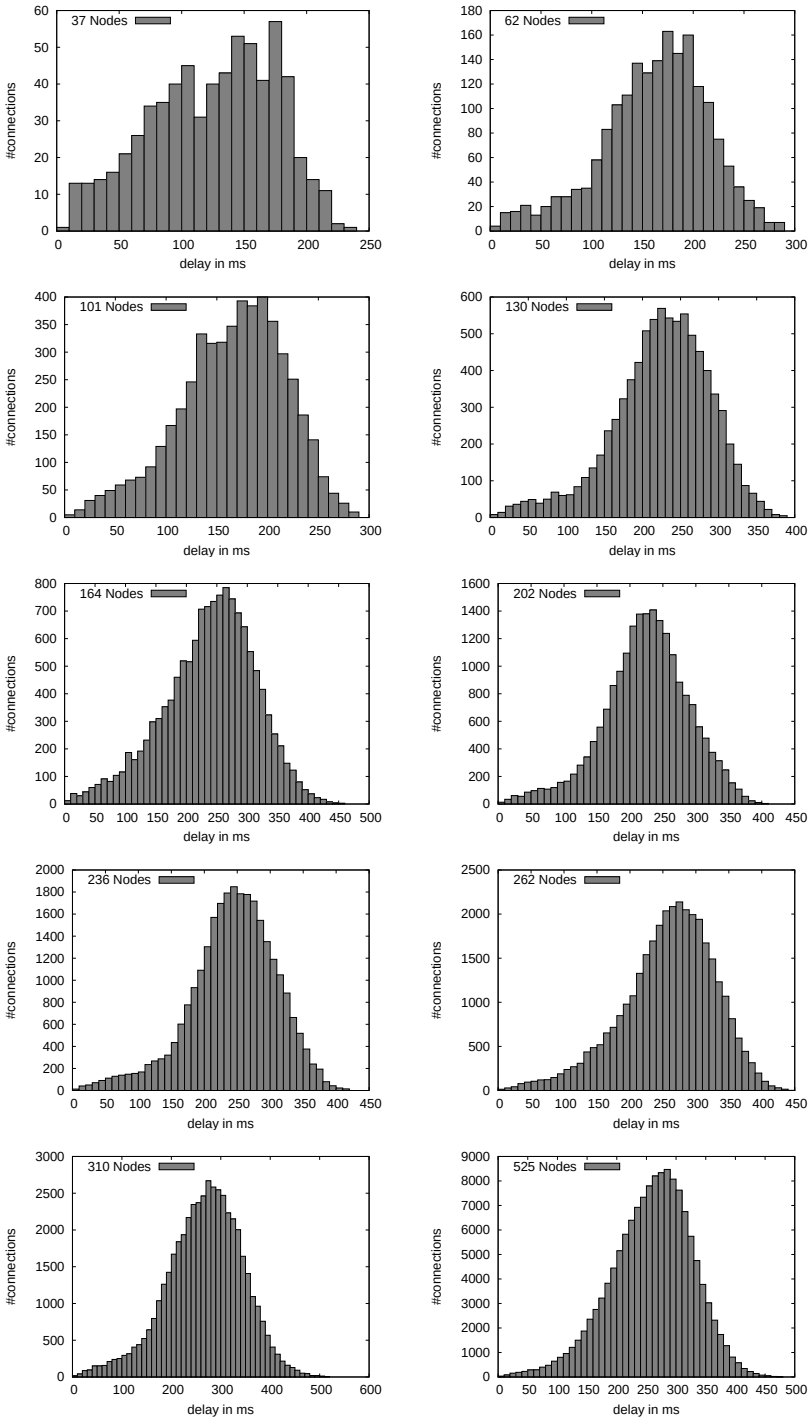


Figure B.5: End-to-end delays of the overlay topology

Appendix C

Increase of the download duration factors with larger file sizes

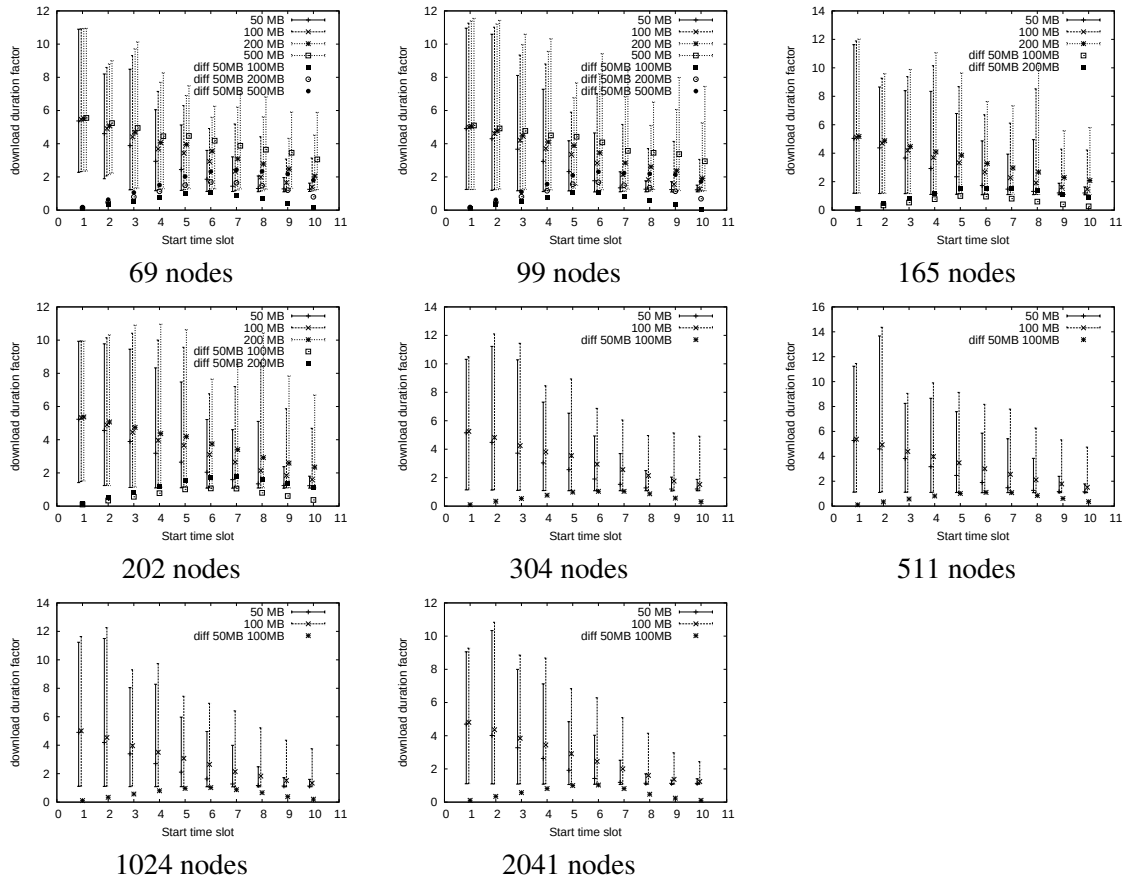


Figure C.1: Sliced download duration factors of dMCFTP with the differences between the different file sizes

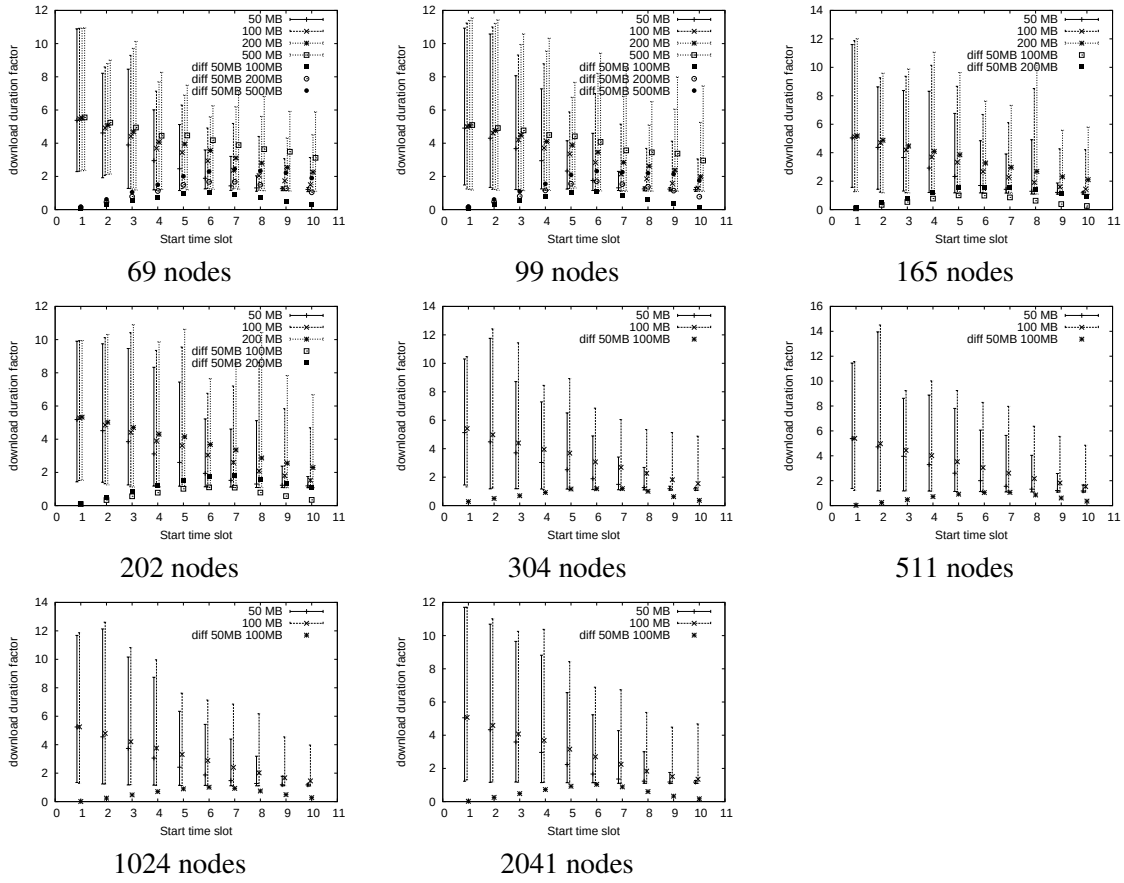


Figure C.2: Sliced download duration factors of cMCFTP with the differences between the different file sizes

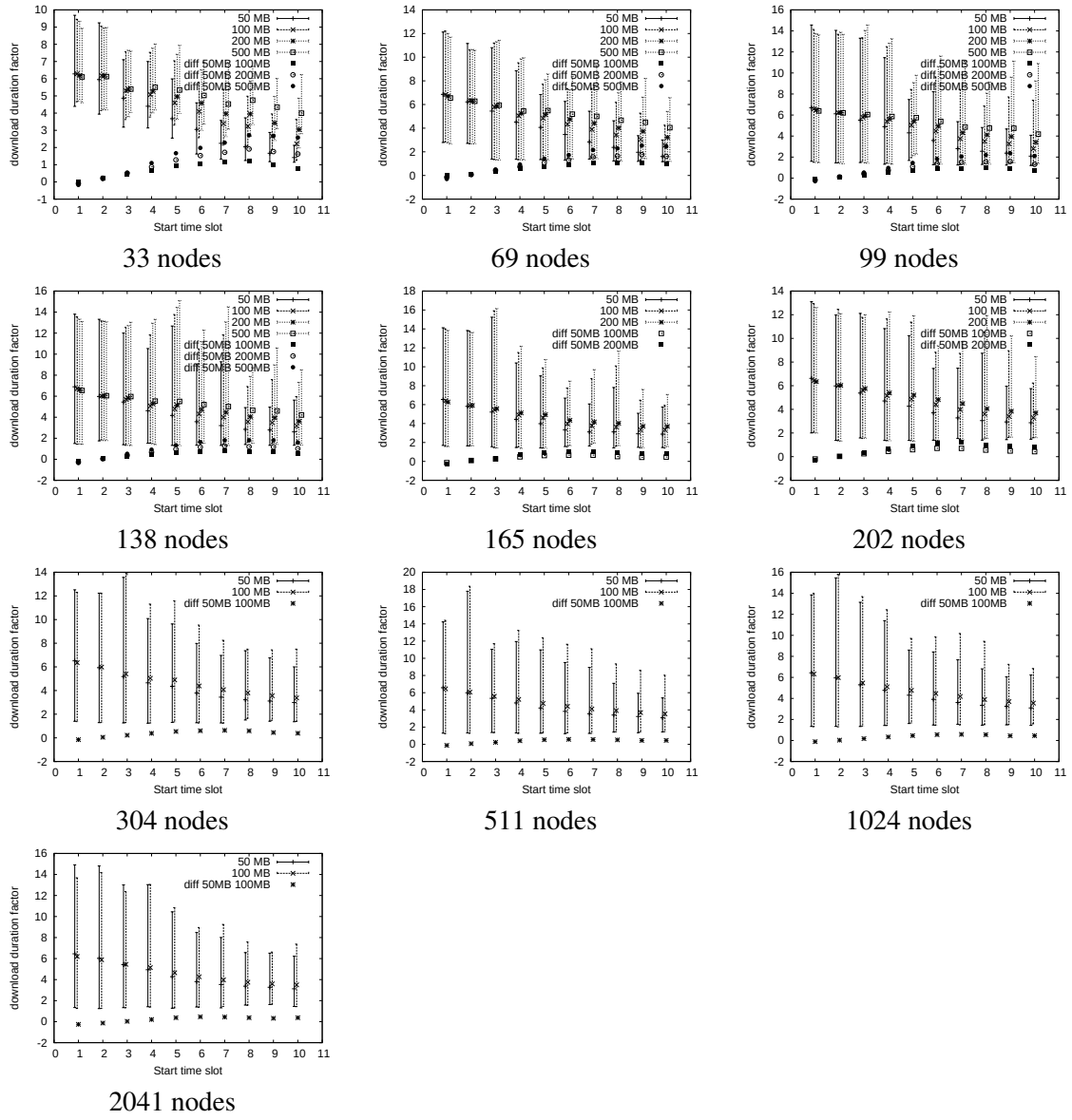
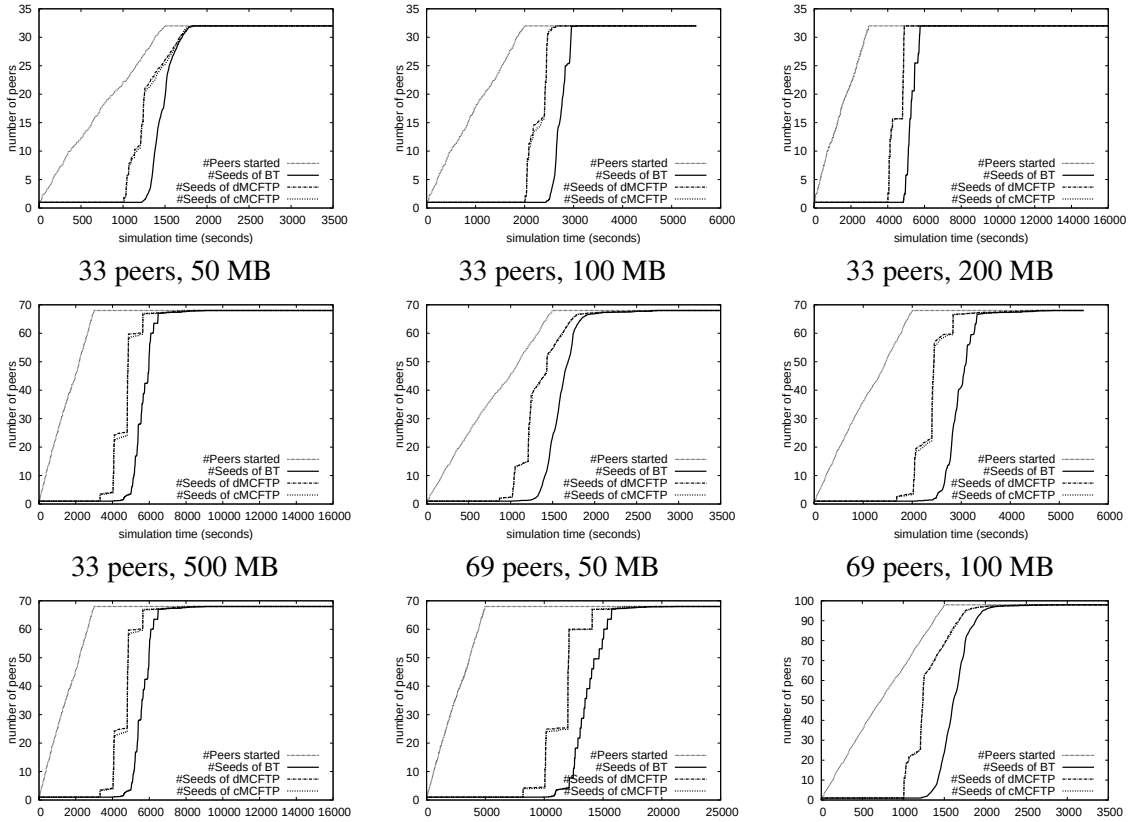


Figure C.3: Sliced download duration factors of BT with the differences between the different file sizes

Appendix D

Leecher-Seed Evolution and Bandwidth Utilization

D.1 Leecher-Seed Evolution



33 peers, 50 MB | 33 peers, 100 MB | 33 peers, 200 MB

33 peers, 500 MB | 69 peers, 50 MB | 69 peers, 100 MB

69 peers, 200 MB | 69 peers, 500 MB | 99 peers, 50 MB

Figure D.1: Leecher-Seed Evolution I

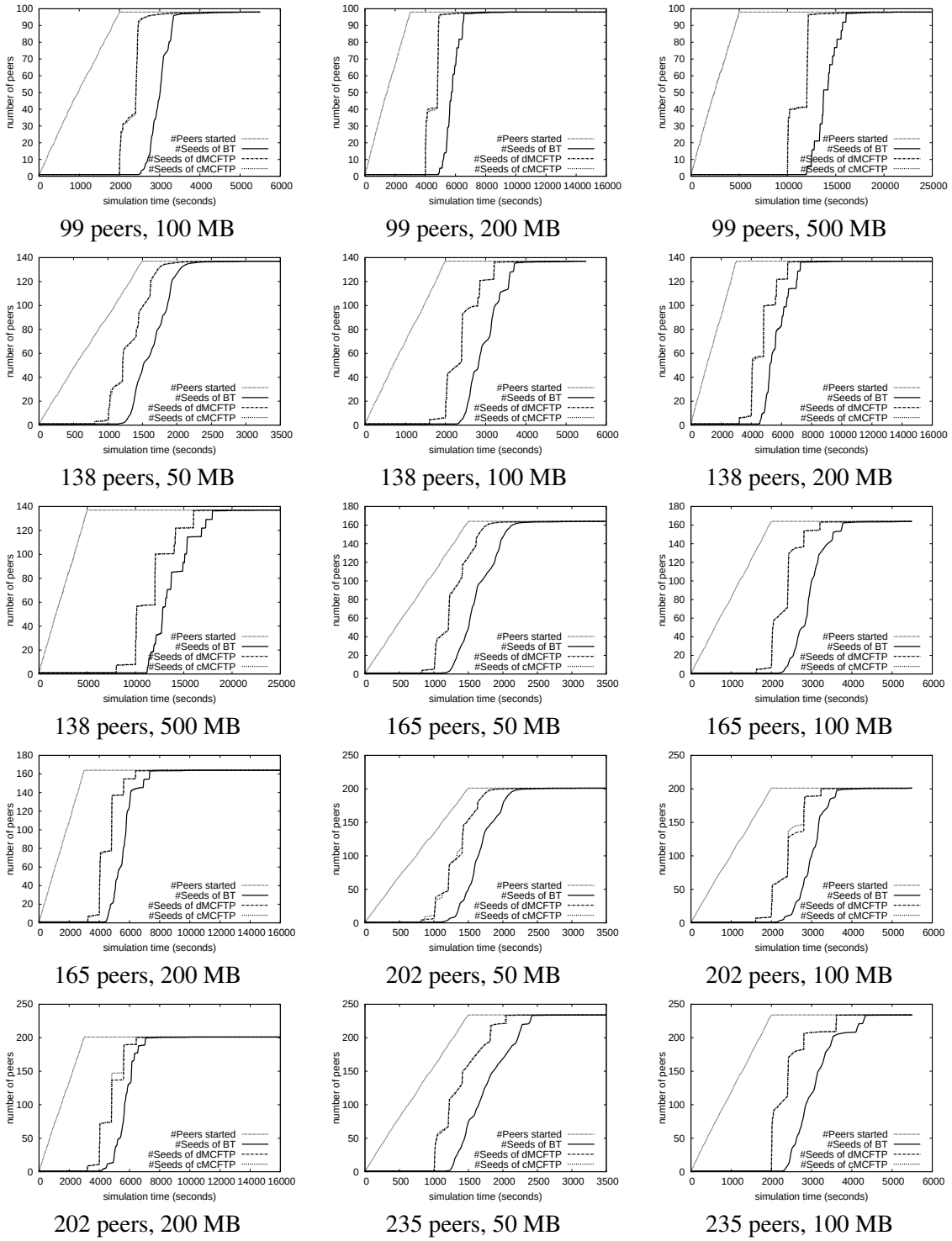


Figure D.2: Leecher-Seed Evolution II

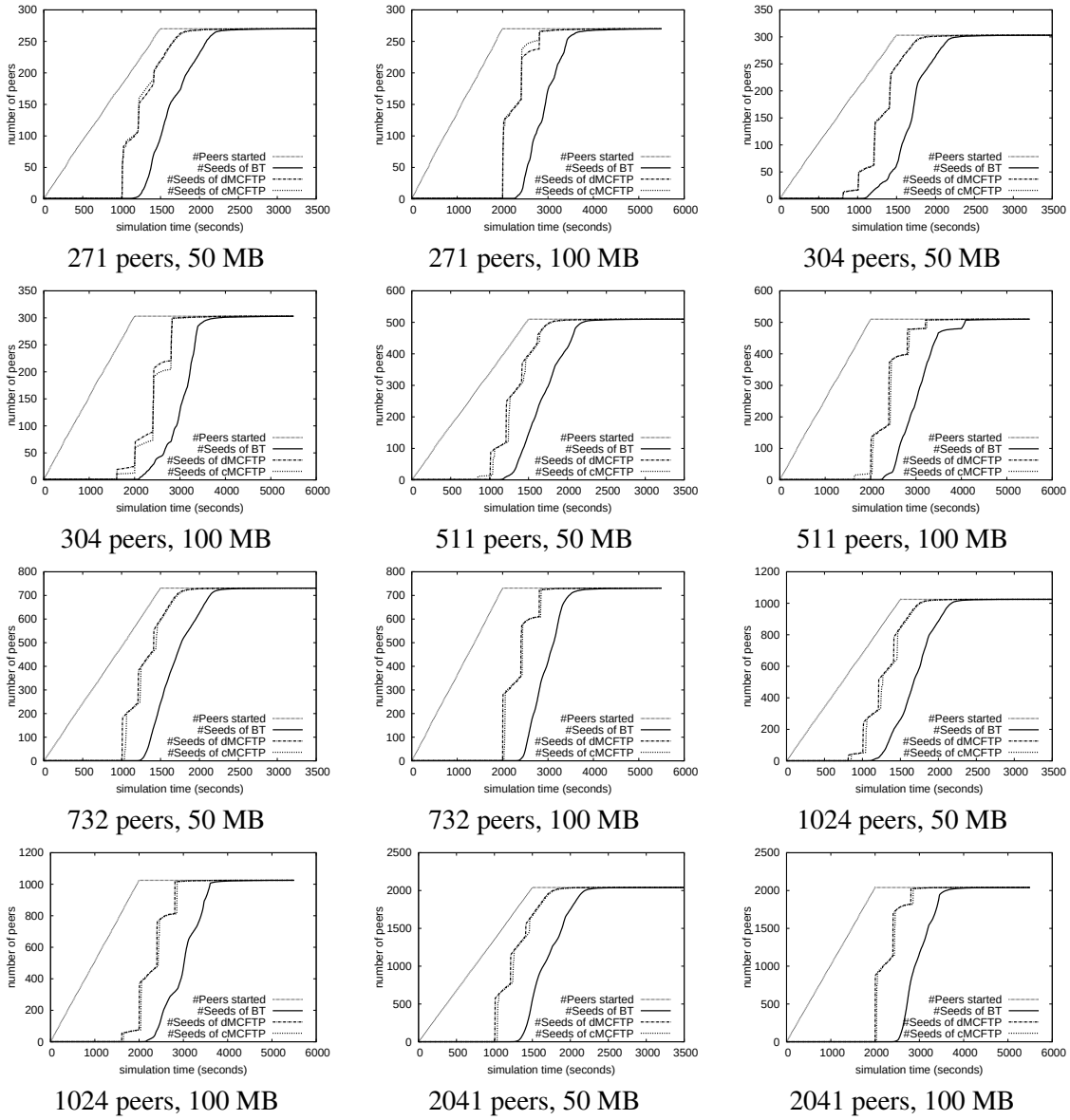


Figure D.3: Leecher-Seed Evolution III

D.2 Bandwidth Utilization

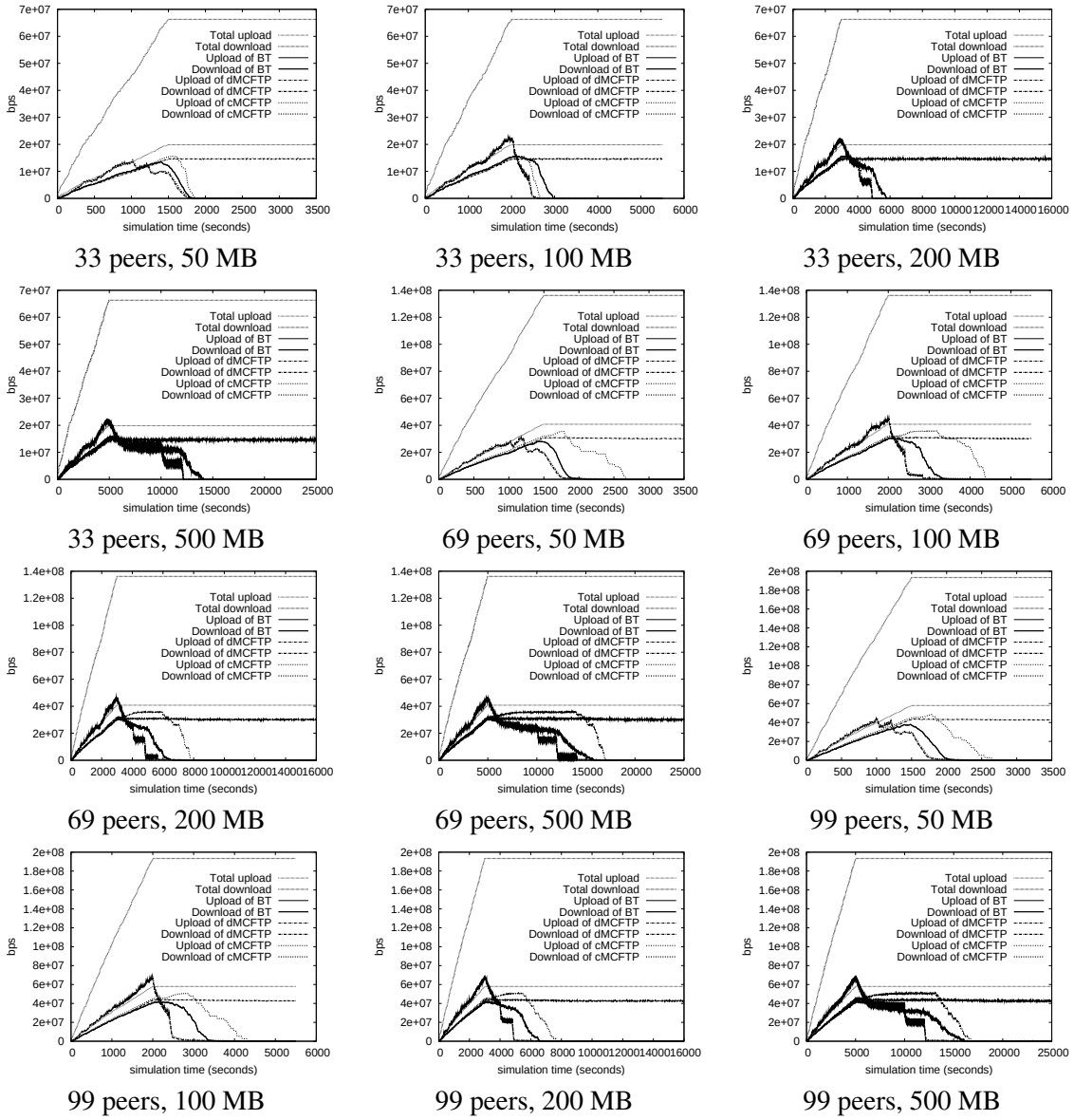
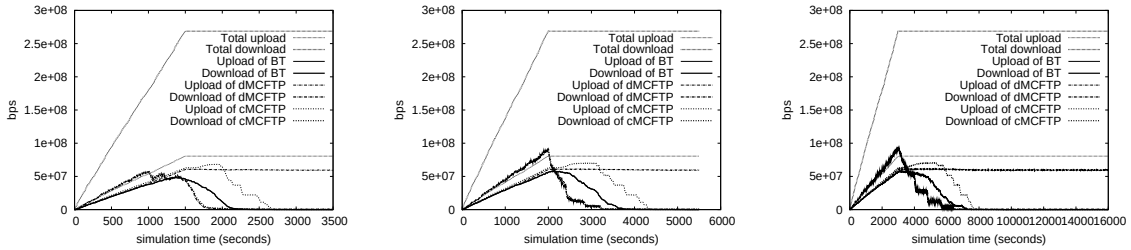
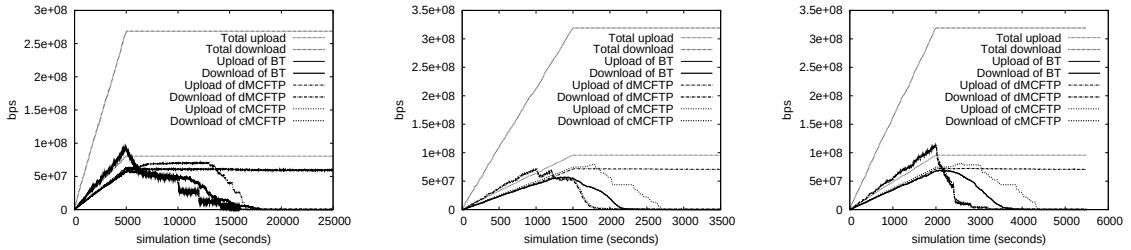


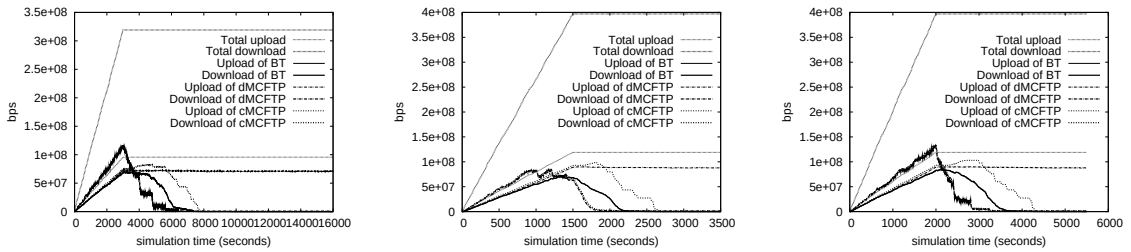
Figure D.4: Bandwidth Utilization I



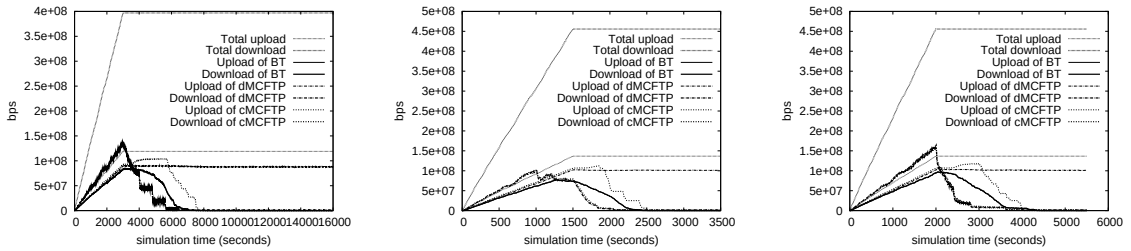
138 peers, 50 MB | 138 peers, 100 MB | 138 peers, 200 MB



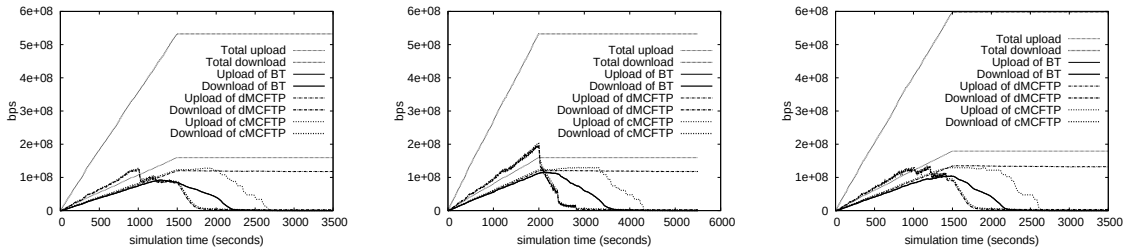
138 peers, 500 MB | 165 peers, 50 MB | 165 peers, 100 MB



165 peers, 200 MB | 202 peers, 50 MB | 202 peers, 100 MB



202 peers, 200 MB | 235 peers, 50 MB | 235 peers, 100 MB



271 peers, 50 MB | 271 peers, 100 MB | 304 peers, 50 MB

Figure D.5: Bandwidth Utilization II

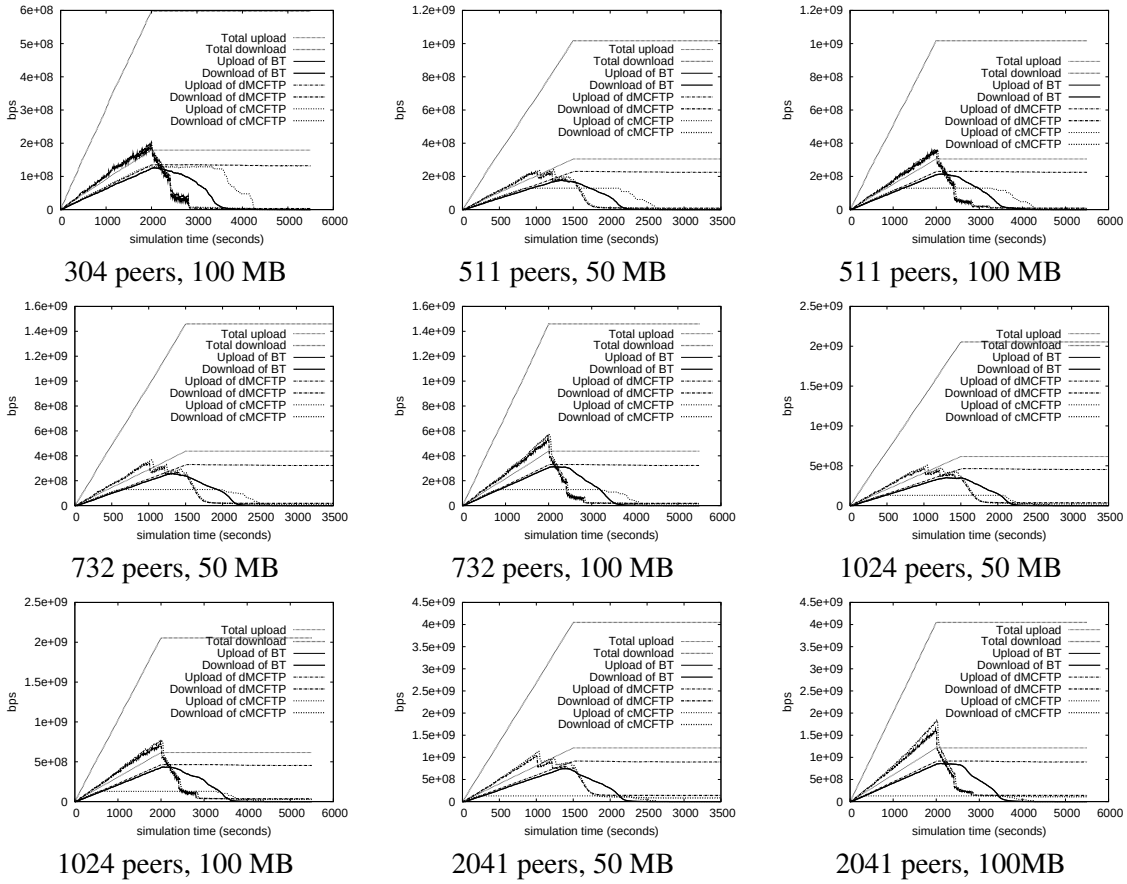


Figure D.6: Bandwidth Utilization III

Appendix E

Impact of Different Number of Initial Seeds

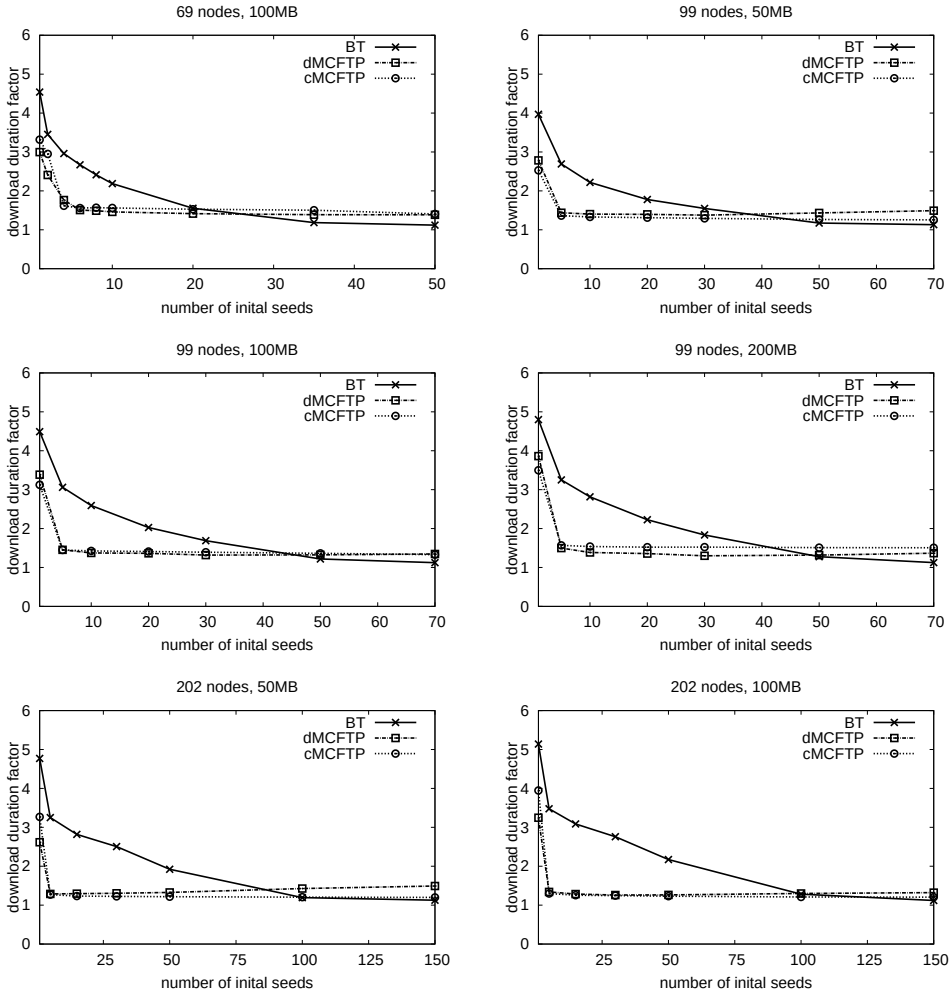


Figure E.1: More initial seeds

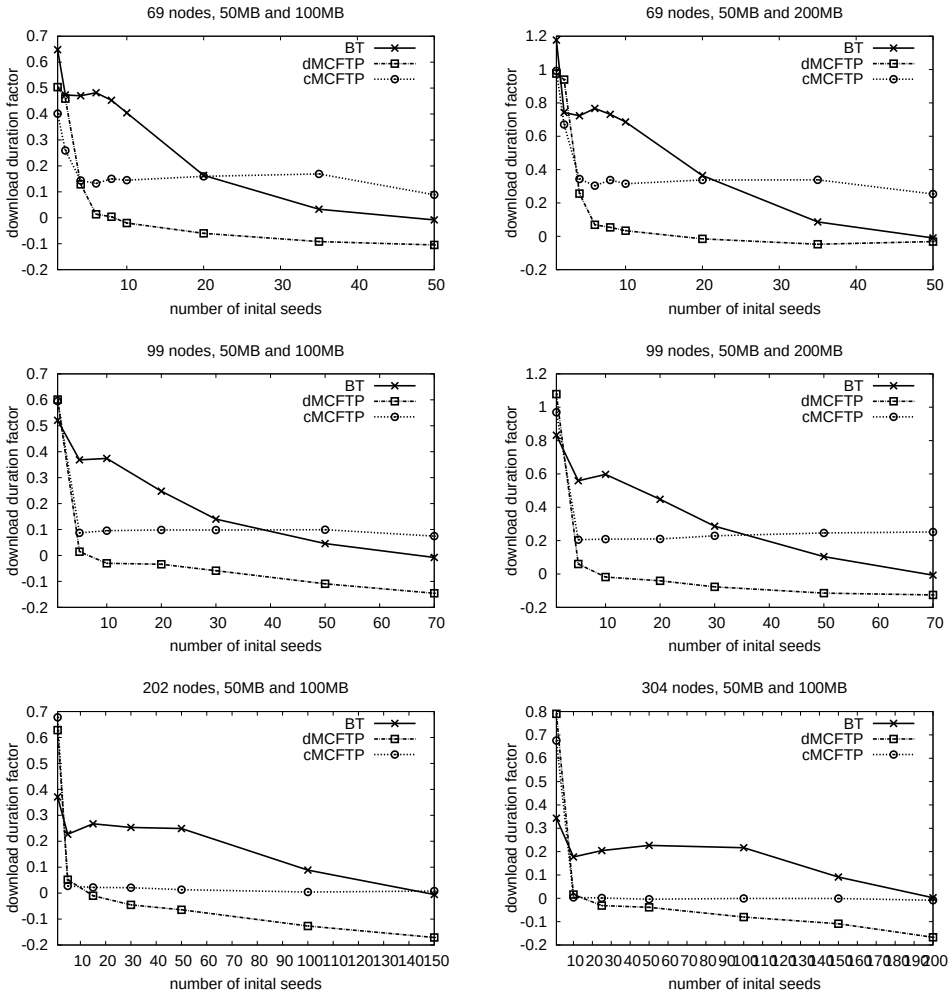


Figure E.2: Differences between the download duration factors of different file sizes

Appendix F

Extensions and Enhancements

F.1 FileLeader Negotiation

A crucial point of using cMCFTP without a single-point of failure is that the FileLeader is not only restricted to one dedicated peer. Thus, cMCFTP needs a mechanism for negotiating a peer as the single FileLeader for a given FileManagementGroup. This happens in two cases: First if no peer has joined the FileManagementGroup yet or second if more than one peer has joined the FileManagementGroup, and the current FileLeader becomes unavailable.

In the first case, when no other peer exists in the swarm, a single peer can easily detect that no active FileLeader exists. This can be detected by a missing entry in the DHT for the given file ID, thus, no FileManagementGroup already exists. So either the peer must reserve a new IP Multicast group address or it must setup a new Overlay Network for the FileManagementGroup and store the group identifier in the DHT. Then this peer becomes the active FileLeader.
If an entry still exists in the DHT, the peer can join the FileManagementGroup, but will miss any traffic in this group. Therefore the peer can declare itself as the active FileLeader. If Overlay Networks are used, the peer is not able to join the Overlay Network. Thus, it must setup a new Overlay Network and update the DHT. But for downloading the file, the peer must wait until other peers, which already have the file, join the FileManagementGroup, which does not happen often.
The most applicable scenario of the first case is, when a new file is made available. Therefore, the first peer is the seeder for the file.

In the next two Sections, we propose two complementary solutions for the second case. The first solution tries to quickly hand-over to a new FileLeader, based on information that the old FileLeader has broadcasted to its peers. The second solution must solve the problem if the first solution does not lead to a new FileLeader.

F.1.1 Preconditions and Assumptions for First Solution

A FileManagementGroup already exists, which is controlled by a single FileLeader. The FileLeader had enough time for gathering information about its peers.

F.1.2 First Solution

The current FileLeader has collected information about its peers, such as the uptime, the upload and download bandwidth, how many times the peer correctly answered on StatusRequests messages and perhaps other feedback from the peers. With all this information, the current FileLeader can decide which N_{BFL} peers are the best ones to become backup FileLeaders. The FileLeader then distributes this as a prioritized list with the StatusRequest messages to all peers. When the peers detect the unavailability of the FileLeader, all participating peers know which peer should become the new FileLeader. Thus, the first peer mentioned in the list must try to become the new FileLeader. All other peers also expect that this peer should become the new FileLeader. If this peer does not announce itself as the new FileLeader after a certain timeout, the second mentioned peer tries to become the new FileLeader, and so on. Therefore, each backup FileLeader has its time slot to announce itself as the new FileLeader. A new FileLeader sends a StatusRequest message immediately followed by the KeepAlive messages as announcements.
If none of these peers became the new FileLeader, the second solution must be used for negotiating a new FileLeader. Since the peers are not synchronized, the second or lower prioritized mentioned backup FileLeader could announce itself as the new FileLeader before the first prioritized does. Because the peers know the priority list, they accept only messages from the new FileLeader with the highest priority. Messages from other FileLeaders are ignored. The FileLeaders with lower priority must stop acting as FileLeader. Wrong FileLeaders can also inform the peers about their termination of acting as FileLeader with the below mentioned Handover message.
Thus, the StatusRequest message must be extended with a list of N_{BFL} backup FileLeaders, each entry twenty bytes long per peer (ID length).

Example:
The old FileLeader distributes with the StatusRequest messages the following list of backup FileLeaders. $N_{BFL} = 4$ and timeout is set to 5 seconds.

- Peer 5, highest priority
- Peer 18, second priority
- Peer 2, third priority
- Peer 98, lowest priority

Figure F.1 shows the different actions of the backup FileLeaders on a time scale. It starts with the last message the old FileLeader sent before it became unavailable. All peers wait five seconds, but no more messages arrived from the FileLeader. Therefore, the peers expect that peer 5 would announce itself as the new FileLeader. Unfortunately peer 5 becomes also unavailable, because the user has shut down the application. Ten seconds after receiving the last message peer 18 should announce itself, but the other peers receive no announcement of it. Thus peer 2 announces itself as the new FileLeader. A short period after the announcement

of peer 2, the announcement of peer 18 is received by the peers. Peer 18 has received the last message of the old FileLeader very late, and the propagation of its announcement takes also a long time. Therefore peer 18 becomes the new FileLeader, because of its higher priority in the list. Peer 2 has also received the announcement of peer 18 and can inform the peers with a Handover message, that it aborts its service as FileLeader. At the point where peer 98 has its slot to announce itself as FileLeader, the new FileLeader is already determined. Thus, peer 98 does nothing because it has also received the two announcements and the Handover message.

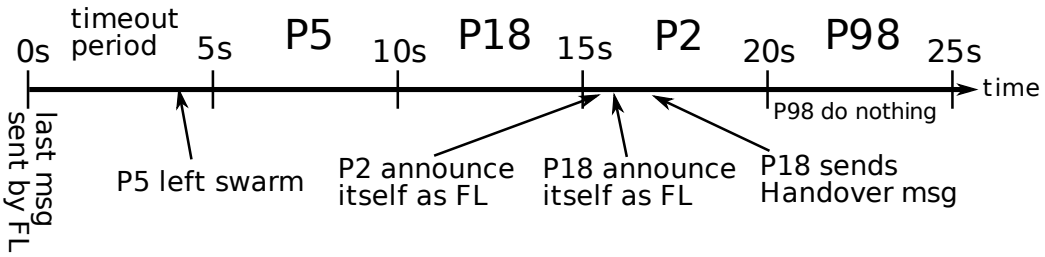


Figure F.1: FileLeader negotiation with the first solution

F.1.3 Preconditions and Assumptions for Second Solution

The first solution does not lead to a new FileLeader, but there are still peers connected. All timeouts are exhausted.

F.1.4 Second Solution

The peers announce with a Negotiation messages a single value calculated from their parameters. This value must form a metric, which reflects the capability of each peer to act as the FileLeader. A Negotiation message only contains the peer ID and the calculated value. To not overload the entire FileManagementGroup, the peers must be able to calculate a threshold on the basis of the current number of peers in the network, which was distributed by the old FileLeader. A peer only participates in this negotiation when its own value is higher. All peers listen to these messages. The peer with the highest value must become the new FileLeader. If two peers have the same value, the peer with the smallest peer ID wins. If this peer does not declare itself as the new FileLeader, the next best peer tries to become the new FileLeader after a certain timeout. There must be no further negotiation because all peers know which one is the winner of the negotiation. If a peer tries to become the FileLeader with a smaller value, this peer will be ignored.

F.1.5 Handover Message

If a client application, which acts as FileLeader, is gracefully terminated, it must announce this to its peers. This handover mechanism works analog to the FileLeader negotiation mechanism provided in the first solution of the previous section, except that the FileLeader explicitly

initiates the handover to the first backup FileLeader. It announces its leave with two Handover messages. The new FileLeader must then react within the normal timeout conditions. This could immediately lead to an new FileLeader. In the worst case, the departure of the old FileLeader is detected when the timeouts are expired, because both of the Handover messages could have been dropped in the network.

The other usage of the Handover message is if a peer tries to become the new FileLeader because perhaps some network problems could have occurred. Also other peers make the same but wrong decision and accept the new FileLeader. But after a while, the new FileLeader and the other peers detect that the old FileLeader is still working. Thus the wrong FileLeader must send a Handover message. This message announces to all peers that the wrong FileLeader will abort its service as FileLeader.
Such a Handover message is really simple. It contains only the peer ID of the old or wrong FileLeader and the peer ID of the new or correct one.

F.1.6 Winner takes it all

A last possibility to resolve the problem of having two or more FileLeaders for one FileManagementGroup is that the peers count, which peers regard which FileLeader as the right one. If a peer detects that most of the other peers regard another FileLeader as the correct one instead of the one it has chosen, it can assume that it has chosen the wrong one.

Therefore, the peers must report their selected FileLeader to the other peers. This can be simply done by extending the FullStatus message with an additional field containing the peer ID of the FileLeader. All peers also receive FullStatus messages from other peers. Then, a peer extracts only the FileLeader field from the message and sums up the occurrence of the different reported FileLeader IDs. Because of some intended FileLeader changes, the counters for every potential FileLeader must be set to zero after a specific time interval. It is best to choose here also the already known backoff interval.

F.2 Security and Anonymity Enhancements

Every peer can join the FileManagementGroup or one of its partition. Thus in cMCFTP, after one request interval every peer can grab the whole information and exactly knows which peer has which chunks of the file. In dMCFTP, if a peer stays longer in the FileManagementGroup, it can also make a history of which peer sends when which chunk. This is sometimes not appreciated. This problem can not be completely solved and only with reservations.
Today anonymity in P2P networks is also more demanded, specially in file sharing networks. Therefore, this Section looks also at the possibility of anonymity in MCFTP.

F.2.1 Security enhancement for cMCFTP

To make cMCFTP more trustful and secure, public key cryptography [57] can be used to encrypt and sign messages. Each peer generates a random public and private key pair. The random peer IDs are replaced with the SHA-1 fingerprints of the public key. Then, the FileLeader can distribute its public key with its StatusRequest messages. With this public key, the peers can encrypt their Status messages, hence only the current FileLeader have all information of the swarm. In contrast, the FileLeader can sign its KeepAlive and Handover messages with its private key. By signing the Handover messages, a handover to a new FileLeader can not be initiated by a malicious peer.

This security enhancement makes it harder for a malicious peer to gather information about the overall swarm. To achieve this, the peer has to first try to become the FileLeader. When the peers also includes the unencrypted fingerprint of the FileLeader's public key in the Status messages, newly connected peers can verify that they have received the correct public key of the FileLeader. By signing the KeepAlive messages, a malicious peer can also not confuse the participating peers by announcing own SendingGroups. The same principle is used if the current FileLeader becomes unavailable. The peers can verify the public key received from the new FileLeader with the expected fingerprint in the backup FileLeader list. Only if no new FileLeader can be determined from this list, the new FileLeader negotiated can not be verified.

The main disadvantage is the additional computation overhead produced by message encryption, decryption and signing. Mainly on the FileLeader, the decryption is a big task and can not be ignored. Also the backup FileLeaders can not gather information of the peers anymore, because the new FileLeader must first distribute its public key. Thus, a quick handover is not possible anymore. The new FileLeader has at least to wait a full request interval until it has all information.

This security enhancement does not completely secure the network and does not guarantee a correctly working protocol. On one hand, a peer can try to become the FileLeader and then distributes valid KeepAlive messages with invalid SendingGroups. On the other hand such a FileLeader can also work correctly to only gather information about all peers. Also these extensions can not prevent that peers do not corporate.

F.2.2 Security enhancement for dMCFTP

dMCFTP needs no security enhancements like cMCFTP does. The use of public key cryptography with the fingerprint of the public key as peer ID can not be used, because peers should not send their ID. Signing messages also yields to nothing and can not be realized. Peers should not distinguish between peers, although the source address of received packets could be used to distinguish between different peers. The messages peers send must be received and processed by all peers. Therefore, encryption can be also never used.

F.2.3 Anonymity Enhancements for cMCFTP

With native IP Multicast, no anonymity can be guaranteed. Each sender puts one of its individual IP addresses corresponding to the outgoing interface in the source address field of the IP header. Thus, a peer can join the FileManagementGroup and listen to the Full- and PartialStatus messages and has the whole information like the FileLeader has within a request interval. If the protocol is secured with the security enhancements mentioned above, it can use the KeepAlive messages to gather information about all peers. Then, it can join for a short time a SendingGroup to associate peer IDs to IP addresses. But this needs a long time. If the peer achieves to become the FileLeader, it has the whole information within a request interval. An additional problem is that a peer can not even detect if a peer is gathering such information.

Therefore, anonymity can only be achieved by using Application Layer Multicast. In the last years, a lot of research has been done about anonymous communication. But research on anonymity in Application Layer Multicast has only been investigated in recent years.

In [58], the authors propose an overlay multicast protocol, which provides mutual anonymity for senders and receivers. The idea behind this overlay is that not all nodes needs anonymity. These non-anonymous nodes form a mesh and establish source rooted trees with a shortest path spanning tree like Narada [59]. The nodes, which want to connect anonymously search for not saturated but already connected nodes and connect to them with a unicast mutual or initiator anonymity protocol.

The drawbacks of such an anonymous overlay multicast is the high latency and the additional computation costs of encryption and decryption. Also the scalability to thousands of nodes, the latency of forming a new multicast tree and how much the degree of anonymity is decrease by the static peer ID while the peer is connected still would have to be evaluated. Thus, more research has to be done until a useful mutual anonymity overlay multicast protocol can be used for cMCFTP.

F.2.4 Anonymity Enhancements for dMCFTP

When running dMCFTP using native IP Multicast, the same problem exists as for cMCFTP. Every peer can analyze the IP header of incoming packets and create a history of when which peer has sent which chunk. When such a peer stays long enough in the swarm, it can create an almost accurate overview of the entire swarm.

Having no peer IDs and the very trivial message flow in combination with a mutual anonymous Overlay Multicast protocol like [58] guarantees a totally anonymous participation of every peer. If the sender and the receiver of a multicast group are mutually anonymous, the SendAnnouncement messages can not be assigned to a specific peer. This is also valid for SendingGroups. Therefore, receivers can not identify a certain peer as the sender of a specific SendingGroup.

F.3 Self Healing Extensions

The download rate of a peer is set by the user. As mentioned before, this is not only used for letting enough bandwidth free for other Internet use, but it is also used by the peer to know how many SendingGroups it can join. If this value is set too high and the peer joins too many SendingGroups, which the underlaying network could not serve, packet drops occur. In the worst case, no chunk could be correctly downloaded. Thus, a peer could automatically lower its user defined download rate. It lowers the download rate in small steps until it can correctly download the chunks of its joined SendingGroups. If the bandwidth is reduced, the peer tries to restore the download rate back to its initial rate. The download rate is not only lowered internally, the peer also reports the currently reduced bandwidth to the FileLeader with the Full- and PartialStatus messages. Then, the FileLeader can respond on such download rate reduction. Especially when all peers of a whole network segment lower their bandwidth, the FileLeader must offer SendingGroups with lower sending rates.

Normally, it can not be determined, where in the network packets are dropped. This could happen directly on the access link of the sender, somewhere in the underlaying network or on the access link of the receiver. Therefore, when a peer must lower its download bandwidth, it is also lowering its upload bandwidth accordingly. If some SendingGroups exists, which have a higher total of upload as newly reported, these SendingGroups should be finished first. The newly assigned SendingGroups from the FileLeader will then have a total sending rate, which will fit the lowered upload bandwidth.

This behavior has also a very nice side effect. If packet drops do not occur at the last hop which means it is not only a faulty set parameter of the user or a simple overload on the access link, but a bottleneck deeper in the underlaying network, all affected peers behind this bottleneck also lower their download bandwidth. This lowering is done until the overall traffic through the bottleneck does not lead to anymore packet drops. Thus, the chunks can be again correctly downloaded and not only fragments of them. Although the overall traffic through this bottleneck is really small, it is better to receive some whole chunks instead of none at all. Because of the multicast protocol, the overall traffic through the bottleneck is also affected by the different numbers of joined SendingGroups, hence lowering and rising the download rate needs to be done gently. In the worst case, all peers join disjunct groups and they can not profit of the multicast advantages. But it can not be worse than peers connected using unicast protocols, such as TCP used by BitTorrent does. With these self healing extensions, the Multicast File Transfer Protocol can react on simple or bigger network problems.

F.4 Extending MCFTP using Erasure Codes

Because of the unreliability of most multicast protocols, some packets can be dropped. This can occur because of short traffic peaks or other short congestion situations in the network. Thus, a mechanism should be made available to recover from these rare packet drops without discarding the current chunk and joining again a SendingGroup for the same chunk later.

Forward Error Correction (FEC) is a technique to detect and correct errors in data communication or data storage. Two well-know exponents of FEC codes are the Reed-Salamon code [60] and the simple Hamming-Code [61]. As an example, Reed-Salomon codes are used in storage systems like CDs, DVDs, Blue-ray disks or RAID6, in data transmission technologies like DSL and WIMAX, and in broadcasting systems like DVB.

But packet-based delivery systems, such as multicast, mainly deal with erasures and not errors. In IP Multicast, UDP packets correctly arrive at the application or are dropped because the CRC check fails, or more frequently because of congestions in the underlaying network. Erasure codes can handle such packet drops by adding redundant information to the packets. Erasure codes are special FEC codes.

The idea behind erasure codes is that a data object is split into several *k* blocks. These blocks are encoded which produces *n* blocks of encoded data. A potential receiver then needs at least *k* encoded blocks to decode the original data object. Therefore, of *n* sent encoded blocks, *n - k* blocks can be dropped.

There exists some work on making multicast more reliable by using FEC codes [62, 63]. Thus, cMCFTP and dMCFTP can directly use such an improved multicast version. Or they could implement their own erasure code mechanism.

The advantage of using erasure codes is that simple packet drops can be recovered. Also, if some peers join a SendingGroup too late, because of a high propagation delay of the KeepAlive messages, they have a chance to download the whole chunk anyway. Also the repetition of chunks in SendingGroups could then be made smaller or even removed, which results in more different chunks available for downloading at the same time.
The main disadvantages are the increase of data transferred by a factor of $\frac{n}{k}$, and the additional computation costs of encoding and decoding the blocks, which is not insignificant.

Bibliography

[1] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM New York, NY, USA, 2001, pp. 149–160.

[2] "Multicast status page provided by AmericaFree.TV," 2008. [Online]. Available: http://www.multicasttech.com/status/

[3] M. Hosseini, D. Ahmed, S. Shirmohammadi, and N. Georganas, "A survey of application-layer multicast protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 58–74, 2007.

[4] *The ns Manual*, The VINT Project, Jan. 2008. [Online]. Available: http://www.isi.edu/nsnam/ns/ns-documentation.html

[5] M. Mysore and G. Varghese, "FTP-M: An FTP-like Multicast File Transfer Application," University of California at San Diego, La Jolla, CA, USA, Tech. Rep., 2001. [Online]. Available: http://www.cs.ucsd.edu/Dienst/UI/2.0/Describe/ncstrl.ucsd_cse/CS2001-0684

[6] D. Kostic, A. Rodriguez, J. R. Albrecht, and A. Vahdat, "Bullet: high bandwidth data dissemination using an overlay mesh," in *SOSP*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 282–297. [Online]. Available: http://dblp.uni-trier.de/db/conf/sosp/sosp2003.html#KosticRAV03

[7] R. Sherwood and R. Braud, "Slurpie: A Cooperative Bulk Data Transfer Protocol," in *INFOCOM*, 2004. [Online]. Available: http://dblp.uni-trier.de/db/conf/infocom/infocom2004.html#SherwoodB04

[8] R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," *Peer-to-Peer Computing*, pp. 101–102, 2001.

[9] D. Bertsekas and R. Gallagher, *Data Networks*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1991.

[10] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of Napster and Gnutella hosts," *Multimedia Syst.*, vol. 9, no. 2, pp. 170–184, 2003. [Online]. Available: http://dblp.uni-trier.de/db/journals/mms/mms9.html#SaroiuGG03

[11] S. Baset and H. Schulzrinne, "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol," in *INFOCOM*. IEEE, 2006. [Online]. Available: http://dblp.uni-trier.de/db/conf/infocom/infocom2006.html#BasetS06

[12] S. Dustdar, H. Gall, and M. Hauswirth, *Software-Architekturen für verteilte Systeme*. Springer, 2003. [Online]. Available: http://lsirpeople.epfl.ch/hauswirth/SA-VS/

[13] K. Aberer, L. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth, "The essence of P2P: a reference architecture for overlay networks," *Fifth IEEE International Conference on Peer-to-Peer Computing*, pp. 11–20, 2005.

[14] "The gnutella protocol specification v0.4," document Revision 1.2. [Online]. Available: http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf

[15] T. Klingberg and R. Manfredi, "Gnutella v0.6," June 2002. [Online]. Available: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html

[16] S. Fahmy and M. Kwon, "Characterizing overlay multicast networks and their costs," *IEEE/ACM Trans. Netw.*, vol. 15, no. 2, pp. 373–386, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/ton/ton15.html#FahmyK07

[17] S. E. Deering, "Host Extensions for IP Multicasting," Internet RFC 1112, August 1989.

[18] B. Quinn and K. Almeroth, "RFC3170: IP Multicast Applications: Challenges and Solutions," *RFC Editor United States*, 2001.

[19] H. Holbrook and B. Cain, "RFC4607: Source-specific multicast for ip," *IETF Request for Comments (RFC)*, vol. 4607, 2006.

[20] Z. Albanna, K. Almeroth, D. Meyer, and M. Schipper, "RFC3171: IANA Guidelines for IPv4 Multicast Address Assignments," *RFC Editor United States*, 2001.

[21] M. Handley, C. Perkins, and E. Whelan, "RFC2974: Session Announcement Protocol," *RFC Editor United States*, 2000.

[22] M. Handley, V. Jacobson, and C. Perkins, "RFC 4566: SDP: Session Description Protocol," *Request for Comments, IETF*, 2006.

[23] I. RFC4291, "IP Version 6 Addressing Architecture," *R. Hinden, S. Deering*, 2006.

[24] B. Haberman, "RFC3307: Allocation Guidelines for IPv6 Multicast Addresses," *RFC Editor United States*, 2002.

[25] B. Haberman and D. Thaler, "RFC3306: Unicast-Prefix-based IPv6 Multicast Addresses," *Internet RFCs*, 2002.

[26] P. Savola and B. Haberman, "RFC 3956: Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address," RFC 3956, November 2004, Tech. Rep., 2004.

[27] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "RFC3376: Internet Group Management Protocol, Version 3," *RFC Editor United States*, 2002.

[28] R. Vida and L. Costa, "RFC3810: Multicast Listener Discovery Version 2 (MLDv2) for IPv6," *Request for Comments, IETF*, 2004.

[29] H. Eriksson, "Mbone: The Multicast Backbone," *Commun. ACM*, vol. 37, no. 8, pp. 54–60, 1994. [Online]. Available: http://dblp.uni-trier.de/db/journals/cacm/cacm37.html#Eriksson94

[30] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized publish-subscribe infrastructure," *IEEE JSAC*, vol. 20, no. 8, 2002.

[31] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, pp. 329–350, 2001.

[32] "The Network simulator ns-2." [Online]. Available: http://www.isi.edu/nsnam/ns/

[33] J. B. Postel and J. K. Reynolds, "File Transfer Protocol (FTP)," Internet RFC 959, October 1985.

[34] R. Ghosh, R. Ghosh, G. Varghese, and G. Varghese, "Congestion control in multicast transport protocols," Tech. Rep., 1998.

[35] K. Miller, K. Robertson, A. Tweedly, and M. White, "StarBurst Multicast File Transfer Protocol (MFTP) Specification," Apr. 1998.

[36] K.-W. Lee, S. Ha, and V. Bharghavan, "IRMA: A Reliable Multicast Architecture for the Internet," in *INFOCOM*, 1999, pp. 1274–1281. [Online]. Available: http://dblp.uni-trier.de/db/conf/infocom/infocom1999-3.html#LeeHB99

[37] L. Cherkasova and J. Lee, "FastReplica: Efficient Large File Distribution Within Content Delivery Networks," in *USENIX Symposium on Internet Technologies and Systems*, 2003. [Online]. Available: http://dblp.uni-trier.de/db/conf/usits/usits2003.html#CherkasovaL03

[38] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *SIGCOMM*, 2000, pp. 43–56. [Online]. Available: http://dblp.uni-trier.de/db/conf/sigcomm/sigcomm2000.html#FloydHPW00

[39] D. Kostic, A. Rodriguez, J. R. Albrecht, A. Bhirud, and A. Vahdat, "Using Random Subsets to Build Scalable Network Services," in *USENIX Symposium on Internet Technologies and Systems*, 2003. [Online]. Available: http://dblp.uni-trier.de/db/conf/usits/usits2003.html#KosticRABV03

[40] Bram Cohen, "Incentives Build Robustness in BitTorrent," *In Proc. 1st Workshop on Economics of Peer-to-Peer Systems*, 22 May 2003.

[41] "bittorrent.org." [Online]. Available: http://bittorrent.org/

[42] "Another BitTorrent protocol specification." [Online]. Available: http://wiki.theory.org/BitTorrentSpecification

[43] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," *Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002*, 2002.

[44] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," Institute of Standards and Technology (NIST), Aug. 2002.

[45] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in P2P systems," *IEEE Internet Computing*, vol. 8, no. 3, pp. 19–26, 2004.

[46] D. Li, X. Lu, B. Wang, J. Su, J. Cao, K. Chan, and H. Leong, "Delay-Bounded range queries in DHT-based peer-to-peer systems," in *Proc. of the 26th IEEE Intï¿½l Conf. on Distributed Computing Systems (ICDCS). New York: IEEE Computer Society*, 2006.

[47] M. Handley and D. Estrin, "The internet multicast address allocation architecture," *Internet Draft*, 2000.

[48] A. Norberg, "libtorrent 0.10," 2006. [Online]. Available: http://sourceforge.net/projects/libtorrent/

[49] A. Binzenhöfer, T. Hoßfeld, G. Kunzmann, and K. Eger, "Efficient Simulation of Large-Scale P2P Networks: Compact Data Structures," in *PDP*. IEEE Computer Society, 2007, pp. 467–474. [Online]. Available: http://dblp.uni-trier.de/db/conf/pdp/pdp2007.html#BinzenhoferHKE07

[50] P. Savola, "RFC 5110: Overview of the Internet Multicast Routing Architecture," *Request for Comments, IETF*, Jan. 2008.

[51] D. Waitzman, C. Partridge, and S. Deering, "RFC1075: Distance Vector Multicast Routing Protocol," *RFC Editor United States*, 1988.

[52] J. Moy, "RFC1584: Multicast Extensions to OSPF," *RFC Editor United States*, 1994.

[53] A. Adams, J. Nicholas, and W. Siadak, "RFC 3973-Protocol Independent Multicast-Dense Mode (PIM-DM): Protocol Specification," 2005.

[54] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas, "RFC 4601: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification (Revised)," RFC 4601 (Proposed Standard), Aug. 2006.[Online]. Available: http://www. ietf. org/rfc/rfc4601. txt, Tech. Rep., Aug. 2006.

[55] T. Bates, Y. Rekhter, R. Chandra, and D. Katz, "RFC2858: Multiprotocol Extensions for BGP-4," *RFC Editor United States*, 2000.

[56] B. Fenner and D. Meyer, "RFC3618: Multicast Source Discovery Protocol (MSDP)," *Internet RFCs*, 2003.

[57] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, Oct. 1996. [Online]. Available: http://www.cacr.math.uwaterloo.ca/hac/

[58] L. Xiao, Y. Liu, W. Gu, D. Xuan, and X. Liu, "Mutual anonymous overlay multicast," *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1205–1216, 2006. [Online]. Available: http://dblp.uni-trier.de/db/journals/jpdc/jpdc66.html#XiaoLGXL06

[59] Y. Chu, S. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1456–1471, 2002.

[60] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, pp. 300–304, 1960.

[61] R. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[62] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "RFC3453: The use of forward error correction (FEC) in reliable multicast," *Internet RFCs*, 2002.

[63] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 2, pp. 24–36, 1997.