Implementation and Evaluation of the Multicast File Transfer Protocol (MCFTP)

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Alican Gecyasar
Dezember, 2010

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

Abstract

Large amounts of data are exchanged and shared over the Internet every day. This is traditionally performed using client-server communication. Big files such as movies and music are nowadays often shared using Peer-to-Peer (P2P) networks like BitTorrent. In this thesis, we introduce an implementation of Multicast File Transfer Protocol (MCFTP), which is a data dissemination approach based on native IP Multicast and Application Level Multicast (ALM). MCFTP aims to save valuable bandwidth resources without increasing download time. MCFTP has no single point of failure and can serve all its downloaders efficiently. We compare MCFTP with BitTorrent, a leading state of the art data dissemination protocol. In this thesis, we present an implementation of MCFTP and show that using our implementation, data can be disseminated in a much more resource-conserving way than when using BitTorrent. Although the presented algorithms are not fully optimized yet, our implementation of MCFTP is normally faster and more resource-conversing regarding data dissemination for every downloader, compared to BitTorrent.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank to Prof. Dr. Torsten Braun who gave me the opportunity to carry out my master thesis in his research group named 'Computer Networks and Distributed Systems'. Furthermore, I would like to express my gratitude and thanks to my tutor Marc Brogle for his effort in exchanging his constructive and interesting views. He gave valuable feedback and advices. He was always available for exchanging ideas and supported me in my approaches. Special thanks go to Dragan Milic and Markus Anwander who were as well available for exchanging ideas and helped me out, when the servers caused problems. Additionally I would like to thank Sandro De Zanet for giving me different kind of input, that motivated me to optimize my results.

Chapter 1

Introduction

1.1 Motivation

The use of the Internet has changed immensely since it was first available. In a time where only text and small pictures had to be exchanged, it was doubtless the best solution to have one server providing data and many clients downloading data from that server. But, one might need to reconsider the client server communication paradigm, as the kind of data exchanged has changed to rather big files like movies or music albums. Not only the amount of data has grown, but also the amount of Internet users has increased dramatically. This means that the same data is often requested multiple times over and over again by different users. Using the traditional client server communication, the server is responding to each of the clients one after another. All work is done by the server and it has to have appropriate bandwidth available, to serve all users requesting this data. There have been many different approaches opposed to traditional client server communication.

1.2 Peer-to-Peer Networks

Peer-to-Peer(P2P) networks seem to be the most successful protocols to disseminate large amount of data. BitTorrent is a widely spread and often used P2P protocol. Nevertheless, there are different issues with BitTorrent. The BitTorrent tracker could be considered as single point of failure. If the BitTorrent tracker fails, data exchange will fail. Newer implementations of BitTorrent use a trackerless version to overcome this issue.

1.3 Multicast Paradigm

Another approach to disseminate data efficiently is the multicast paradigm. It offers a possibility to send data simultaneously to a group of receivers. An implementation of the multicast paradigm for the Internet is IP Multicast. The multicast functionality has to be enabled at the routers inside an IP Multicast network. IP Multicast has been available for over 20 years. Unfortunately, IP Multicast was not being used widely in the Internet for a long time. This was most likely due to the missing support by Internet Service Providers (ISP). ISPs have billing

challenges and new security issues when confronted with IP Multicast. Nowadays Internet TV applications like zattoo[1] and wilmaa[2] use native IP Multicast. IP Multicast can be used in an environment where all routers and service providers between end nodes support it. But, this is not the case for the entire Internet. To overcome these limitations, Application Level Multicast (ALM) approaches were developed, which shift the multicast functionally from the network to the end systems.

1.4 Contributions

In this thesis, we introduce Multicast File Transfer Protocol (MCFTP) a data dissemination approach based on native IP Multicast and Application Level Multicast. MCFTP focuses on reducing overall bandwidth consumption for all downloaders by using the multicast paradigm. Evaluations of MCFTP on a simulation basis have already shown good results in [3].
With this thesis, we define and implement MCFTP from scratch such that it can compete with BitTorrent and still be resource-conserving. The implementation consider native IP Multicast as well as Application Level Multicast. We use Scribe/Pastry for the MCFTP-ALM implementation. The ALM framework can be replaced by any other ALM framework with little effort. To experiment with different settings, two different approaches of MCFTP are implemented. The first approach is called centralized Multicast File Transfer Protocol (cMCFTP). It has one file leader node, which is responsible for coordinating all data dissemination. The other approach is called decentralized Multicast File Transfer Protocol (dMCFTP) and does not require a file leader node. All nodes are equal and each of them is responsible for efficient data dissemination. The decentralized approach is inspired by the trackerless version of BitTorrent. These two MCFTP approaches are evaluated separately. The most important part of our implementation is the possibility to adapt dissemination strategies. Furthermore, tools are provided to extend and optimize dissemination strategies for further work. We compare MCFTP to the most popular protocol, which is used to exchange data, such as movies, music and more. Peer-to-Peer (P2P) is surely the most used and most spread protocol to disseminate data. We use BitTorrent, a very well known P2P protocol to compare its performance with MCFTP. We focus especially on scenarios with one seeder, but also evaluate scenarios with more than one seeder.
With this implementation we proof the concept of MCFTP. Although it is only a prototype version of MCFTP, we are able to evaluate different dissemination strategies. Special care is given to code interfaces, such that this implementation can be used for future experiments by easily adapting dissemination strategies or the underlying ALM framework.

1.5 Thesis Outline

Chapter 2 gives an overview of already existing mechanisms and approaches. One of the main approaches discussed in detail is the BitTorrent protocol. Another focus is on Scribe and Pastry, as we use these protocols for our underlying Application Level Multicast framework. The main chapter of this thesis is Chapter 3, where all information about MCFTP is provided. MCFTP is presented with all its elements and variations. Potential problems are pointed out and solutions

are provided on how to avoid them. In Chapter 4, details about our specific MCFTP implementation are depicted. Furthermore, details on how MCFTP messages are constructed are also described. In Chapter 5, the results and evaluations of the comparison between BitTorrent and MCFTP are presented. Details about the used environment and problems encountered during evaluations can be found in this chapter as well. Chapter 6 sums up evaluations and points out various ideas for future work.

Chapter 2

Related Work

Efficient data distribution has been quite an important research topic for a while. There are several approaches to distribute data over a network. In this Chapter, different approaches are explained and underlined with examples starting with the traditional unicast architecture model in Section 2.1, followed by IP Multicast shown in Section 2.2. P2P networks are presented in Section 2.3. Finally Application Level Multicast is introduced in Section 2.4.

2.1 Traditional Client-Server Unicast Communication

A traditional unicast architecture model is based on roles, with a server and a client. The server is keeping data and as the name implies, it serves the data whenever it is requested by a client. The bandwidth limitations are given either by the maximum available download rate by the clients or the maximum available upload rate by the server.

$$Bandwidth_{Limit} = min(max(Client_{Downloadrate}), max(Server_{Uploadrate}))$$

In a traditional unicast architecture model, there is one server containing the data and several clients requesting data from this server. Assuming there are multiple clients requesting the same data at the same time from the server, which might occur actually very often if the server contains popular data, then the server can either serve all the clients at the same time or one after the other, or a combination of both. In either case, the server will send out the same data as often as the amount of clients requesting the data. If the server serves all the clients at the same time, the bandwidth limit will decrease dramatically. As an example, all clients could have the same available download rate of 500kBps and the server's upload rate could be 1000kBps. In a scenario with up to two clients, this case would work optimal. The clients could download at their full rate of 500kBps. But imagine 20 clients downloading at the same time from the server. This would lead to a bandwidth limit of 50kBps for each client. Thus, to have a correct bandwidth limit, we should not forget the amount of clients. In a homogeneous network with the same sort of clients and bandwidth, this leads to the following bandwidth limit for each client:

$$Bandwidth_{LimitEach} = \\ min(max(Client_{Downloadrate}), max(Server_{Uploadrate})/Amount_{Clients})$$

For an inhomogeneous network the bandwidth limit calculation gets rather confusing, simply because there are different kinds of variations and strategies. For example, the server could serve all the clients as it does in a homogeneous network or the server could prefer to serve only clients with a higher bandwidth rate. In client-server communication, all file transmissions start from the server to each client served. The server is always the source of all transmissions. Given a scenario like in Fig. 2.1 with one sender and four receivers, the server starts a new transmission to each of the clients interested in the file. The server sends the same data to the first router four times. The first router forwards the same data twice to the second router.

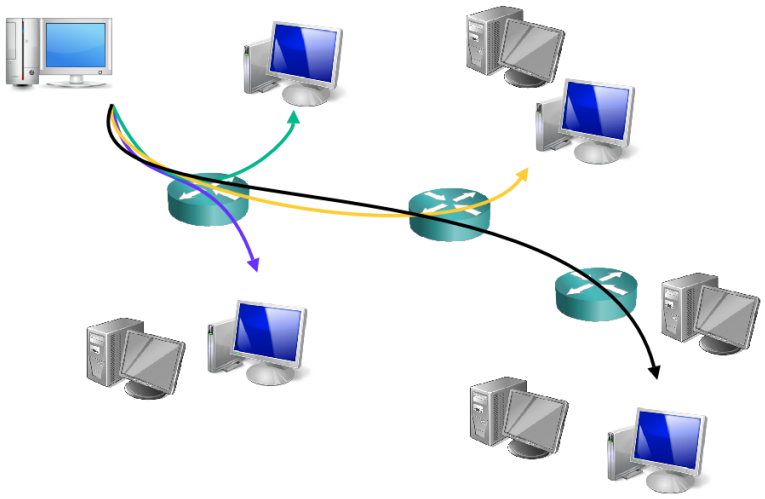


Figure 2.1: Client-Server communication: A node sends data to four clients, one after the other or in parallel

These transmissions are redundant. There are a total of eleven transmissions including the routers. A further downside of the traditional unicast approach is, that it does not take the location of the clients into account. If there is only one server, which has the needed files, all clients have to get the files from this server, there is no alternative. If the server is for example located in California, USA, but some of the clients are located in Europe, they would each have to get the file by themselves. Another disadvantage is that the server is a single point of failure. If the server crashes, goes down for maintenance or simply is not reachable, no one can download any data from the server. All these examples show, that there is a lot of potential to optimize the traditional unicast file distribution approach.

2.2 IP Multicast

IP Multicast communication [4] [5] [6] gives the possibility to address a group of nodes at once by sending IP packets to an IP Multicast address and not directly to a receiver's IP address. Nodes interested form a multicast group by joining the IP Multicast address. An IPv4 IP Multicast address is in the range of 224.0.0.0 to 239.255.255.255. There is no need to be joined in a multicast group, in order to send data packets to a multicast group. In any case, whether the

sender is joined to the group or not, the sender has no information about any node joined to the multicast group nor about the size of the group. Any node can join and leave a multicast group at will. More than one multicast group can be joined at a time. The roles of server and receiver are not clearly distributed, since every node could send data to the IP Multicast address at any time. There can be more than one sender per multicast group. Figure 2.2 is the same scenario as presented in Fig. 2.1. This time, IP Multicast is being used to transmit a file. The source sends the file only once and the data is being replicated at the routers. Therefore, there is only one transmission between the source and the first router and also between the first and second router. In total, there are 7 transmissions.

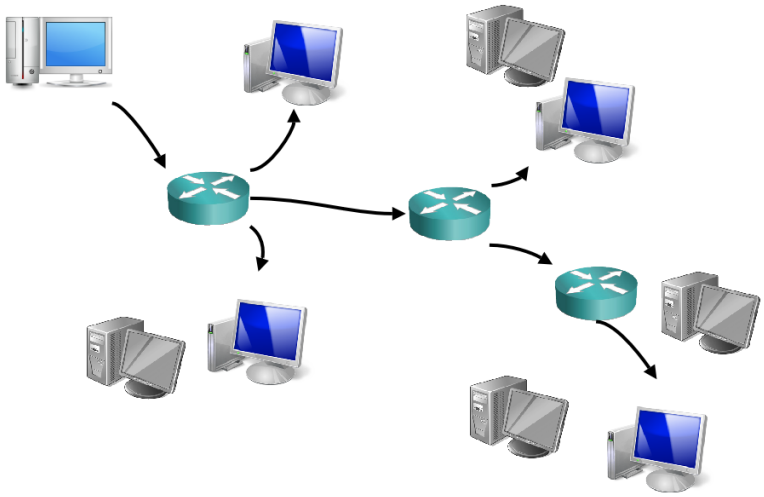


Figure 2.2: Multicast: A node sends data only once but is received by four clients in parallel

Local area networks support IP Multicasting over Ethernet for immediate neighbors in the same subnetwork, no special hardware is needed. But as soon as switches or routers are in between networks, the hardware has to support IP Multicasting. Especially to use IP Multicast over the Internet, all the routers and all the switches on the way to the own ISP have to support IP Multicast. This though limits to use multicast within one ISP's network. To use IP Multicast all over the Internet, all the ISPs and backbones should be using IP Multicast supporting hardware. However, this is not (yet) the case due to several reasons. For an ISP it could be unclear how to bill the traffic usage. But there are different approaches in order to use the advantages of multicast communication over the Internet without IP Multicast support. Some of them will be described in Section 2.4.

2.3 Peer-to-Peer Networks

Peer-to-Peer (P2P) networks [7] [8] have become a very popular way of distributing and sharing data between nodes. The advantages compared to traditional client-server networking are shown in Fig. 2.3 and 2.4. The client-server network is built using a star topology. Every client has only one connection, which is a connection to the server. If the server goes down for maintenance

or is not reachable due to any kind of reason, this topology can not continue sharing data. The server is the only node using its upload-bandwidth. Whereas in a P2P network, there are multiple connections between nodes. Since a file is broken into file pieces, the nodes can share the pieces they have with other nodes. A node does not have to get all the pieces from the same node, and therefore has more flexibility to fetch the whole file. When one particular node is not reachable by the network, the rest of the nodes still can continue to share data among each other. Chances are that all nodes are using their upload-bandwidth. In a P2P network, data is often parted into chunks and distributed partially. As soon as a node has a few chunks, it can start to act like a server and distribute the chunks it has. This mechanism is reducing the payload from the normal server and distributing the bandwidth load to the P2P nodes in the network.

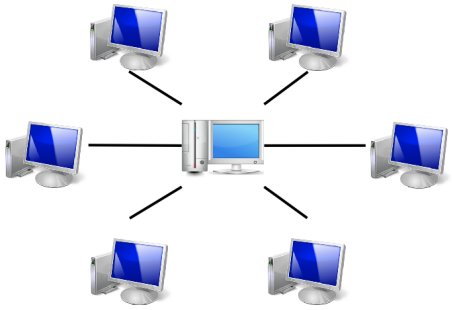


Figure 2.3: Client-Server network

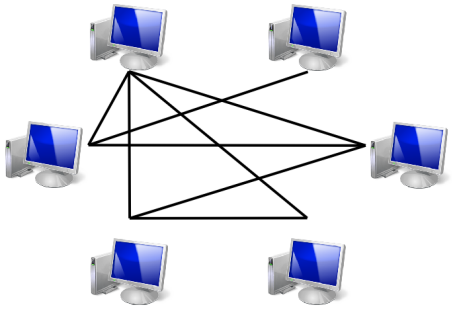


Figure 2.4: Peer-to-Peer network

In this Section, two popular P2P protocols will be introduced and discussed: BitTorrent and Pastry.

2.3.1 BitTorrent

The BitTorrent protocol was first introduced in July 2001 by Bram Cohen [9][10]. Its purpose is to distribute the upload cost and duties among the downloaders. The protocol is based on a torrent file, which includes various information about the data being downloaded and shared. A torrent file must be fetched prior to entering the P2P network. The main information stored in a torrent file are:

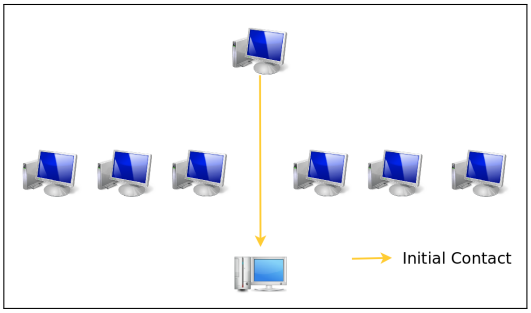
- File length in bytes
- File name
- Amount of pieces
- Concatenated SHA1 [11] hash value of each piece
- Tracker URL

In the BitTorrent protocol, a file is divided into pieces, which are distributed inside the P2P network. Each piece is a small part of the actual file. To manage the piece distribution inside the P2P network, a tracker is needed. The tracker is a server, which keeps information about the

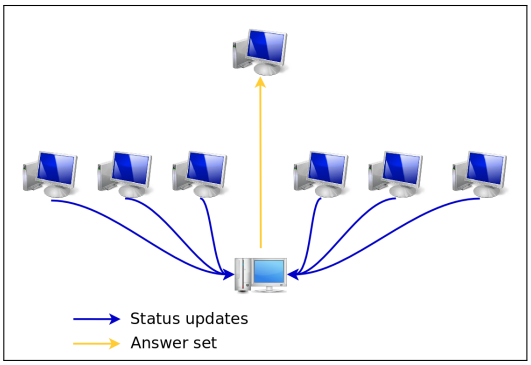
status of all clients inside the P2P network. Thus, the URL of the tracker is an important piece of information, which has to be included in the torrent file.

BitTorrent Swarm Establishment

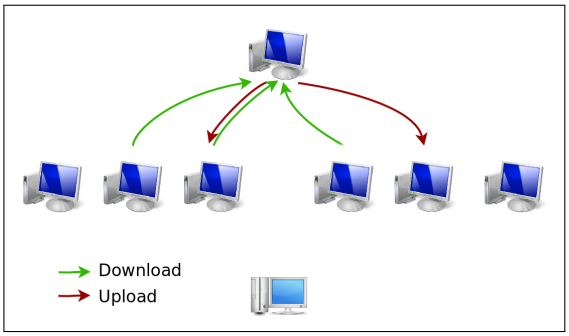
For an illustrated overview of the BitTorrent protocol see Fig. 2.5. When a new node is interested in a file, it downloads the torrent file and opens it with a BitTorrent client. The client then contacts the BitTorrent tracker defined in the torrent file. In this example, there exists already a P2P network, and therefore the tracker already knows about the existing P2P network. The tracker knows about all the nodes and their status of already downloaded file pieces. It returns a set of nodes to the new joined node, telling which other nodes in the P2P network to contact next.



(a) New node contacts the BitTorrent tracker



(b) BitTorrent tracker receives status updates of the P2P network and returns a set of peers to the new node



(c) New node becomes part of the P2P network downloading and uploading pieces

Figure 2.5: An overview of how BitTorrent works

Seeder and Leecher

In BitTorrent terms, there is a leecher and a seeder. A node who has the complete file is considered a seeder. Compared to the traditional client-server scenario, a seeder would be like a server.

A leecher on the other hand is a node, which does not have the complete file. Nevertheless, it might have already some pieces of the file, but not all of them. Leechers could be considered as clients in the traditional client-server scenario.

Torrent-File

For a file to be shared, a seeder has to create a torrent file. A torrent tracker has to be declared with a full URL or hostname and a port to which it will be listening to. The announced tracker has to be reachable at any time while the file is being shared, otherwise data can not be distributed. During creation, the file is parted into pieces and for each of the pieces a SHA1 hash is calculated and put into the torrent file. This hash will help identify which pieces already have been downloaded and which are still missing.
After it has been created, the torrent file has to be distributed among the nodes, which are interested in the original file. The torrent file is rarely larger than 20kB and can be distributed using traditional client-server communication.

File Sharing in BitTorrent

Once the torrent file is created and distributed, and the torrent tracker is up and running, sharing the file is possible. Interested nodes (called ”peers” in BitTorrent terms) contact the tracker through a simple protocol and let it know, which file they are interested in and which pieces they have already. This exchange of information is done periodically by each peer inside the P2P network. The torrent tracker in return sends to each peer a list of other peers interested in the same file. The standard algorithm suggests to include only a random subset of all peers inside this list. This information sent by the tracker will then be used by the peers to connect to each other and build the P2P network. When two peers connect with each other, they exchange information about their needs concerning the pieces of the file. There is a chance that two peers connect but have nothing to share with each other. It is also possible that a peer has already reached its maximum upload bandwidth limit and therefore is not able to share data with further peers. There is also a mechanism called *choking*, which disallows another peer to download any pieces. This mechanism will be discussed later. Thus, not all peers which are connected to another, do exchange pieces of a file.
As soon as a peer finishes downloading one piece, it creates the SHA1 hash of that piece and cross-checks it with the SHA1 hash in the torrent file. If they match, the peer can announce the new information to the connected peers. When all the pieces are downloaded, a leecher turns into a seeder.
There are two main algorithms for BitTorrent, the piece selection algorithm and the choking algorithm. Both of them are subject to be adapted and changed by implementations. We would like to discuss the main algorithms for each and give a little insight to them.

Piece Selection

One of the main keys to a good performance in BitTorrent is the piece selection algorithm. It is used to define the order of pieces being shared in the P2P network. The main problem that could

arise when a bad algorithm is used, is that not all of the nodes would finish downloading after a certain time. When thinking about a piece selection algorithm, it is important to remember that a node (especially a seeder) might not stay for a long time in the network and that there might be different bandwidth available for each of the nodes. There are three piece selection algorithms proposed in [9].

Rarest First

This algorithm is aware of the risk that nodes can disconnect from the P2P network at any time. Therefore, the main goal of this algorithm is to replicate the rare pieces as soon as possible, before the node with the rare piece would disconnect from the P2P network. The rarest piece within the set of peers connected to a node is selected and downloaded, if there is enough bandwidth available. Pieces, which are more common are postponed for later download.

Random First

An issue with the ”Rarest First” algorithm is that rare pieces are mostly present on only one node. The node with a rare piece could possibly not deliver the rare piece to all of the nodes in the P2P network with an accurate bandwidth. At the beginning of a P2P network, no leecher has any piece, and therefore can not upload any data. The idea behind the ”Random First” algorithm is that a leecher should have something to offer to other nodes as soon as possible. This is considered as more important than to focus on the rarest piece at the beginning. According to [9] the ”Random First” algorithm should turn into the ”Rarest First” algorithm as soon as the first piece is downloaded from the P2P network.

Endgame Mode

BitTorrent also has a strategy to speed up the finishing period of a piece download. A piece is divided into subpieces and a request is based on a subpiece. When a node has requested all of the missing sub-pieces, it also sends out a request for the same sub-pieces to all of the nodes it has an open connection to. As soon as a sub-piece is fetched, a cancel message is being sent to avoid redundant data transfers. This strategy is mainly introduced to avoid a delay when finishing the download of a piece which could happen with slow transfer rates. Bram Cohen ensures in [9]: ”In practice, not much bandwidth is wasted this way, since the endgame period is very short, and the end of a file is always downloaded quickly.”

Choking

BitTorrent nodes decide by themselves who they want to share data with. There is no global instance telling which nodes should send which pieces to whom. Nodes will try to download pieces from any other node they have an open connection to. But a node can restrict its upload to certain nodes. When a node is not willing to upload data to another node, then this is referred as ’choking’. The ’choke’ mechanism is introduced by BitTorrent to gain better performance for the whole P2P network. By default, four nodes are always unchoked. The algorithm to select

which nodes to unchoke is based on a variant of tit-for-tat. That means, a node is more willing to upload data to another node of which it is currently downloading data from.

Trackerless BitTorrent

The original BitTorrent protocol as proposed by [9] is not mentioning a trackerless version of BitTorrent. The idea of having a decentralized tracking came up with later approaches. The most important purpose of a BitTorrent tracker is to provide nodes with information of other nodes in the BitTorrent swarm. The most common problem with the tracker is, that it is a single point of failure. If the tracker fails, the whole BitTorrent swarm fails. There have been solutions having multiple trackers, but the most used solution is to apply a trackerless version of BitTorrent having a distributed hash table(DHT) to share information about the BitTorrent swarm. Most of the current BitTorrent clients use Kademlia [12] as their DHT implementation. The DHT implements the BitTorrent tracker functionalities. DHT is still used to provide nodes with information of other nodes in the BitTorrent swarm. In [13], there is another approach to replace the BitTorrent tracker completely by using entry points which perform random node selection without tracking all nodes in the swarm. Instead they perform multiple perpetual random walks. The bias of these walks determine the distribution from which nodes are randomly sampled. This approach omits tracker functionalities completely. In trackerless approaches of BitTorrent there are no differences between nodes. All nodes are the same and each of them has the same responsibility to disseminate data. We took this idea and used it for our decentralized approaches (See 3.2.2).

2.3.2 Pastry P2P Network

Pastry is an implementation of a self-organizing P2P network proposed by Antony Rowstron and Peter Druschel in November 2001 [14]. Each node in Pastry uses a unique random node ID when joining a Pastry network. A node ID is 128-bit long and is uniformly distributed in a circular space in the range of 0 to $2^{128} - 1$. By default, any ID or key in Pastry is represented as a sequence of 32 hex. When a Pastry node receives a message and its destination represented as a numeric key, it will route the message to the Pastry node, which is numerically closest to the given destination address.

Pastry Node Setup

Every Pastry node maintains a neighborhood set, a leaf set and a routing table. The neighborhood set is referred to as M and keeps track of nodes, which are close in terms of locality. To measure the locality distance, different variables may be considered, such as hop count or round trip time (RTT) delays. The neighborhood set is mainly used to maintain the routing table and is not used for routing messages. The leaf set, referred to as L, contains nodes, which are numerically close by ID and it is mainly used for routing. Half of the entries in the leaf set have a larger node ID and the other half has a smaller node ID than the current node. The neighborhood set and leaf set both have 16 or 32 entries each in a default Pastry configuration. The routing table has 16 columns by default, one for each hex value. The rows in the routing table indicate the amount of matching prefixes of a node's ID to the current node's ID, starting with zero. For the sake of

simplicity, our example is using a quaternary numeral system instead of a hexadecimal numeral system and node IDs are limited to a size of only 10 bit. For the given node with ID 21023, the routing table might look like shown in Table 2.1.

Routing Table of node: 21023

	0	1	2	3
0	**0**2321	**1**3202	**2**1023	**3**2110
1	*2-***0**112	*2-***1**023	*2-***2**331	*2-***3**012
2	*21-***0**23	*21-***1**32	*21-***2**02	*21-***3**20
3	*210-***0**2	*210-***1**2	*210-***2**3	*210-***3**0
4	*2102-***0**			*2102-***3**

Table 2.1: Pastry: Example of a routing table in a quaternary system

Row *n* means that there are *n* common prefix matches for the given node's ID compared to the current node's ID and that digit *n+1* of the node's ID, is the same as the column value. For any entry in the routing table, there might be multiple candidates fulfilling the described requirements. But only the closest node according to the neighborhood set is chosen and put into the routing table. In case of no such node in the Pastry network, the field in the routing table is left empty. The routing table is not the prime lookup table when routing a message.

Routing in Pastry

Every message has a key, which is considered as the destination address. Thus, every node will try to forward the message to the node with the same ID as the key of the message. This is done by first checking if the message key is in the range between the lowest and highest node ID inside the leaf set. If so, the message will be forwarded to the node, which has the numerically closest node ID to the message key. If the message key is outside the mentioned range, then the routing table is consulted. The only criteria, which has to be fulfilled, is that the next node ID should have at least one more common prefix digit match than the current node's ID. Assuming that the node from the example before does not have an entry in its leaf set for the message key 21213, then a lookup in the routing table is required. The current node ID is 21023. Since the first two digits of the current node ID match with the message key, row number 2 of the routing table has to be considered. Next, the node with at least one more common prefix digit has to be found. This is done by looking at the columns of the routing table. Accordingly, the node with ID 21202 would be selected as the next node in our example. If there is no matching entry in the routing table or the node is not reachable, then the message has to be routed to a node, which has at least the same amount of matching prefixes and its node ID is numerically closer to the messages key then the current node's ID. This is considered as a special case though. The authors of Pastry state: "The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network." [14]. Figure 2.6 is an example of how the message with the key 21213 would be routed in the given Pastry ring.

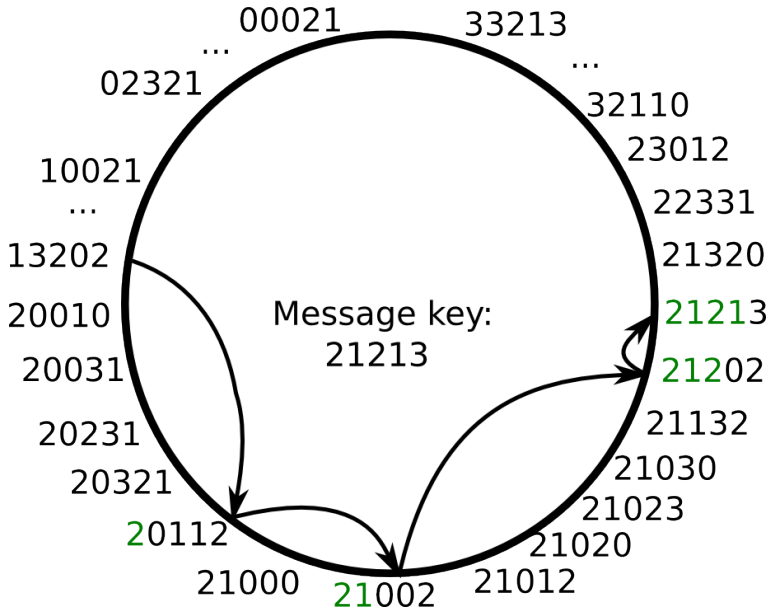


Figure 2.6: Pastry: Example of routing a message

A message sent in the Pastry network is routed to the node with the same first digit as the message key. In the example, this is digit 2. Next, the message is routed to the node with one more common prefix digit. That would be 21. This routing mechanism is continued until the message key and the node ID match all digits.

2.4 Application Level Multicast / Overlay Multicast

As described in Section 2.2, IP Multicast has an efficient solution for many to many communication. But, as also discussed in Section 2.2, there is no Internet-wide support for IP Multicast as of today. There have been different approaches to support a multicast communication over the Internet. These approaches are mostly referred to as Application Level Multicast or Overlay Multicast [15] [16] [17]. In this Section, three Application Level Multicast Models will be introduced. Basically, the idea is to move the multicasting functionalities to the end systems. Therefore, the multicasting trees are built directly between the end systems, using a P2P network. These models are not as efficient as IP Multicast, but are rather thought of as an alternative solution to enable multicast over the Internet. Figure 2.7 shows an example of Application Level Multicast compared to IP Multicast. In the IP Multicast example, there is no physical link used twice to distribute data. Routers are replicating data and send it to subscribed nodes. In the Application Level Multicast example, data is replicated at end nodes, and therefore some of the physical links are used more than only once.

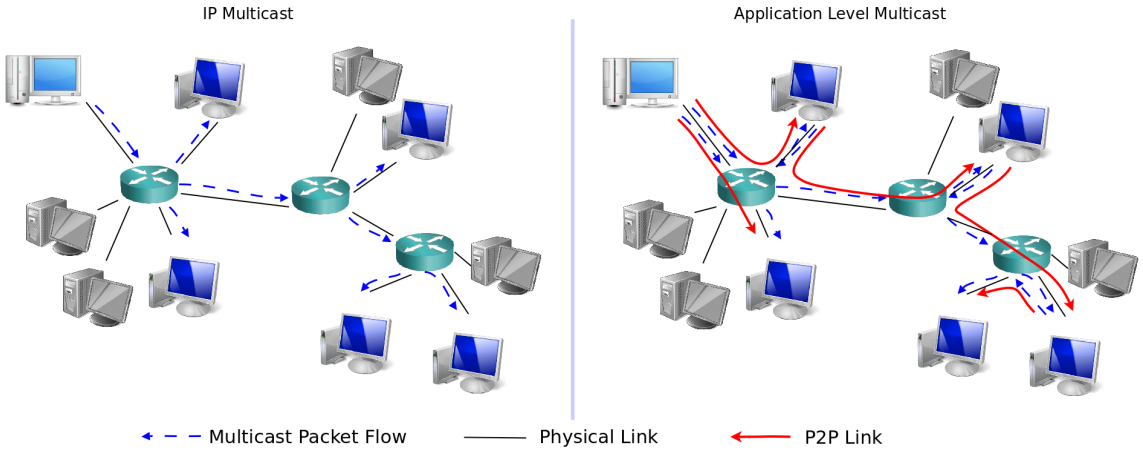


Figure 2.7: Application Level Multicast compared to IP Multicast

2.4.1 Slurpie

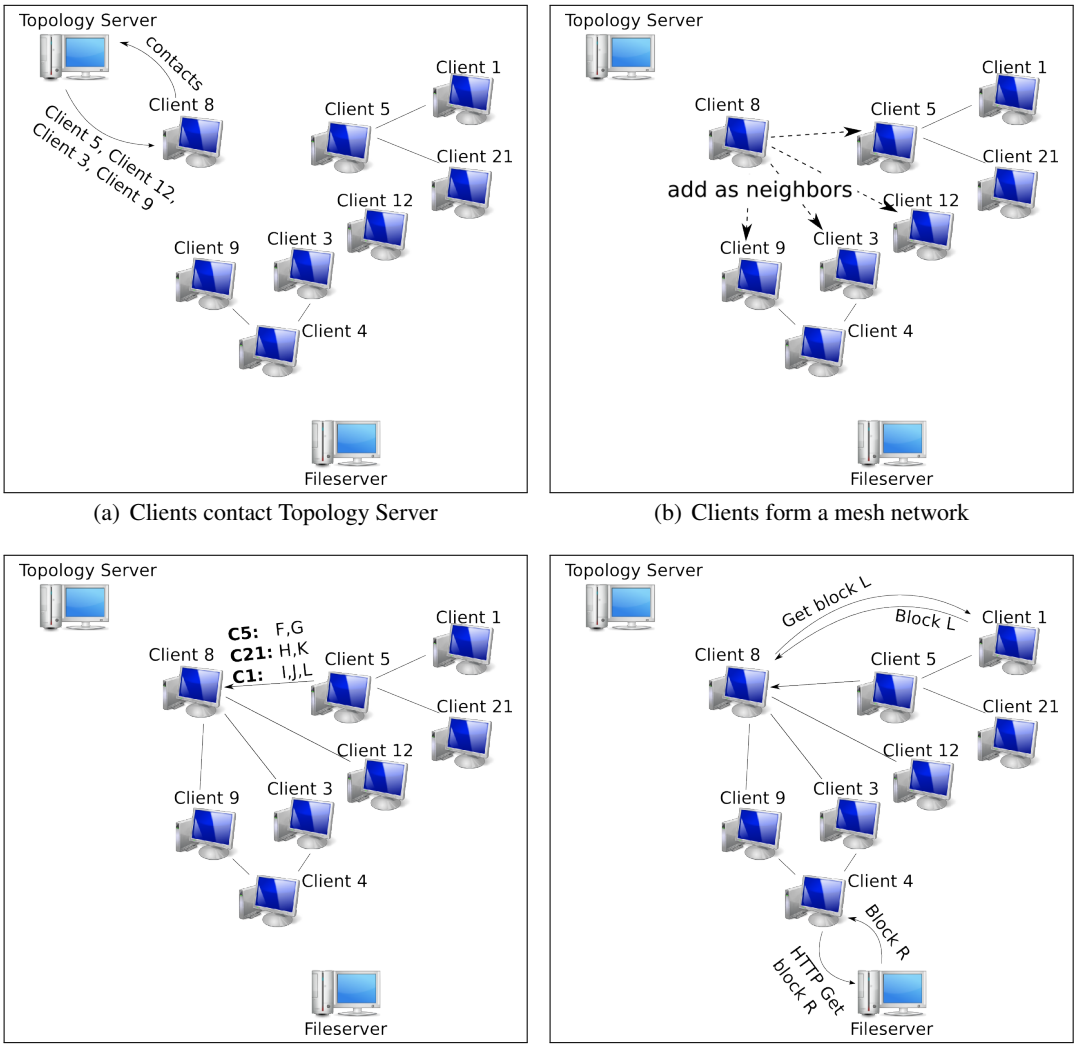
Slurpie was developed at the University of Maryland in 2004 [18]. Slurpie has the goal to reduce client download times and to reduce load on servers. To achieve this goal, Slurpie builds an overlay network between downloading clients, which is referred to as a mesh network. Instead of getting all data from the server, clients try to download data from each other. Only if this is not possible, the server is contacted to download missing data.

Implementation

Every client downloading the same file is first contacting a topology server as shown in Fig. 2.8(a). The topology server is returning a list of some other clients, which have contacted the topology server before. In the example, client 8 receives a list from the topology server with clients 5, 12, 3 and 9. Furthermore, client 8 adds all clients of the newly received list as its neighbors and the clients form a mesh network as shown in Fig. 2.8(b). In the mesh network, progress updates are propagated periodically to let other clients know which client has which blocks. In Fig. 2.8(c) client 5 is propagating information about blocks F to L to client 8. Each client stores information about N other clients. Data blocks can now be requested by client 8 as shown in Fig. 2.8(d). If a client needs a block, which non of his neighbors can propagate to him, the client would make a HTTP or FTP [19] request to a file-server as client 4 is doing in Fig. 2.8(d).

2.4.2 Scribe

Scribe [20] is an Application Level Multicast infrastructure built on top of Pastry (presented in Section 2.3.2). A Pastry network has to exist prior the usage of a Scribe infrastructure. Scribe's effort is to bring multicast functionality to a Pastry network. Any node inside a Scribe infrastructure can create a multicast group.



(a) Clients contact Topology Server

(b) Clients form a mesh network

(c) Clients exchange information

(d) File server used only when needed

Figure 2.8: An overview of how Slurpie works

Any other node is able to join a multicast group as well to send a message to all of the multicast group members. The main methods introduced with Scribe's application programming interface (API) are: create, deliver, join, forward, leave and multicast. A scribe multicast group is also referred as topic.

Creating a Multicast Group

The create method is used to create a new multicast group for a given group ID as Fig. 2.9 shows. To create a multicast group, Scribe creates a new Pastry message containing the keyword "CREATE" and the group ID as the message key. Pastry delivers the message to the Pastry

node with the numerically closest node ID for the given group ID. Pastry passes the content of the message to the Scribe method called deliver, which then points the Scribe node to be the rendezvous node for the given group ID.

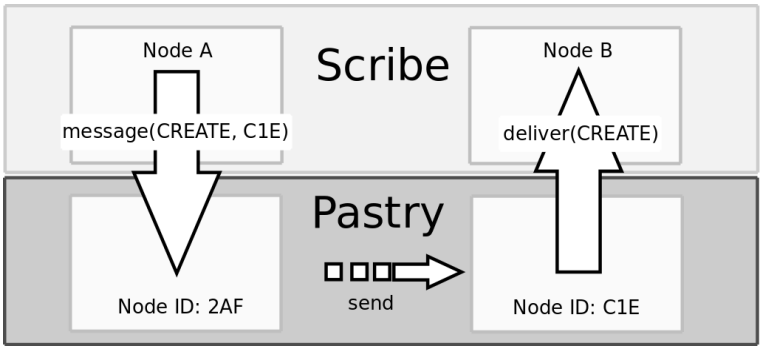
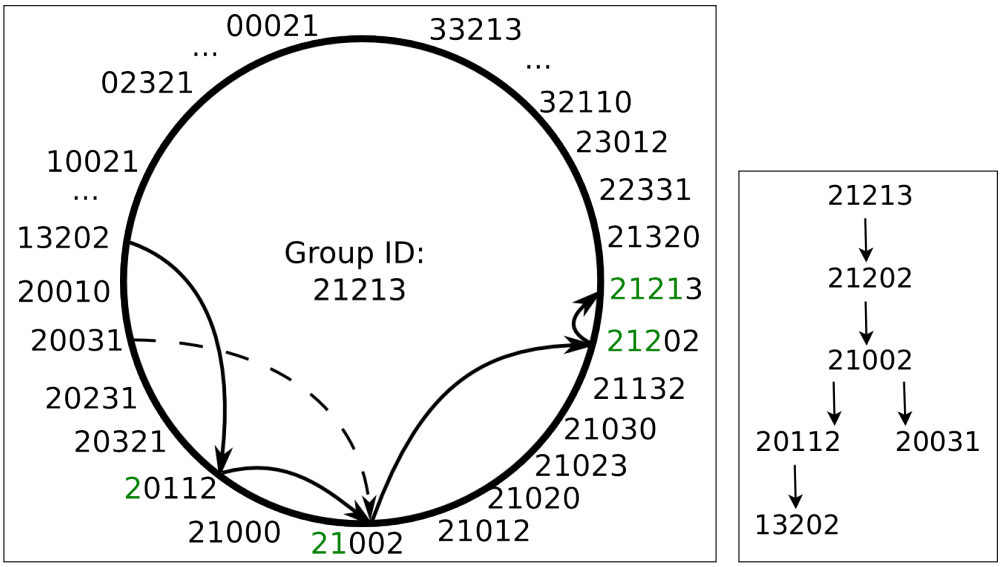


Figure 2.9: Creating a multicast group in Scribe

Joining and Leaving a Multicast Group

As soon as a rendezvous node is determined, Scribe nodes can join to the given group, by creating a Pastry message with the group ID as the message key and the content ”JOIN”. While Pastry is routing the message towards the rendezvous node, each of the intermediate Pastry nodes call the forward method of Scribe. The forward method is used to build a multicast tree rooted at the rendezvous point. When the forward method is called, the current Scribe node adds the preceding node to its children table and sends back the message to Pastry to continue delivering the original message. If Scribe has the preceding node already in its children table, the original message will be discarded. Figure 2.10(a) depicts an example of the same scenario as in Section 2.3.2. In this example, all children tables are empty and the multicast group with ID 21213 has just been created. Node 13202 is the first one to send out a join message towards the rendezvous node 21213. The standard Pastry routing is applied while each of the intermediate nodes is calling the forward method of Scribe. Since all children tables are empty, each of the intermediate nodes will add the preceding node to their children table and forward the original message using Pastry. Node 20031 is willing to join the same multicast group, and therefore sends a join message towards the rendezvous point. Node 21002 will receive the original message, add node 20031 to its children table and forward the original message to node 21202. Node 21202 already having node 20031 in its children table will not add any new information to its children table and it will not forward the original message further. Thus, the multicast tree as shown in Fig. 2.10(b) will be built after the first two join messages are sent for the given multicast group with ID 21213. Due to Pastry’s routing mechanism, as described in Section 2.3.2 there will not be loops in the multicast tree. When a node wants to leave a multicast group, it first checks the children table, if there are any other nodes, which are in its tree branch. If the node wanting to leave is a leaf node of the multicast tree, then Scribe creates a Pastry message with the content ”LEAVE” and the group ID as the message key. It also stores that it left the group. If there are other nodes in the children table, the node wanting to leave only

stores that it left the group and does not create a Pastry message. When a Scribe node receives a ”LEAVE” message, it confirms, that there are no other entries in its children table than the one who just left the group before it forwards the message. The ”LEAVE” message will not be forwarded when there are still other nodes in the children table.



(a) Two nodes send a join message (b) Resulting multicast tree

Figure 2.10: Scribe using children tables to create a multicast tree

Let us assume that node 20031 is leaving the multicast group 21213 from the example before. After Node 21002 receives the ”LEAVE” message, it checks its children table and sees that there is still another node with ID 20112 subscribed. Therefore, node 21002 is not forwarding the ”LEAVE” message. Further, let us also assume that node 13202 is leaving the multicast group after a while. Node 20112 receives the message and confirms that there are no other entries in its children table. Node 20112 forwards the ”LEAVE” message, since its children table is empty at this time. Nodes 21002, 21202 and 21213 will do the same recursively.

Message Transfer inside a Multicast Group

When a node wants to send data to a multicast group, it first creates a Pastry message with the keyword ”MULTICAST” and the group ID as the message key. After routing the message, the rendezvous node returns its IP address. This initial message is sent to cache the IP address of the rendezvous node in order to avoid repeated routing through the Pastry network. To send the actual data as a multicast message, Scribe creates a new message sent directly to the IP address of the rendezvous node. If the rendezvous node changes or is not reachable anymore, a new routing through Pastry is initiated to gather another IP address to send the multicast message to. Once the rendezvous node receives the multicast message, it disseminates the message to all its leaf nodes along the multicast tree. Therefore, any Scribe node receiving the multicast message has joined the multicast group, is used as a forwarder, or both.

Summary and Conclusion

In this chapter, we presented why there is a need to restructure the way big files are exchanged. Traditional client-server communication is inefficient for one-to-many communication when the data exchanged is large and/or the amount of participants is increasing. IP Multicast is efficient for group based one-to-many and many-to-many communication. But, IP Multicast is not available over the Internet, mostly because of ISP restrictions. Hence, Application Level Multicast could be used on top of P2P instead. We presented various P2P networks and Application Level Multicast approaches such as Slurpie, BitTorrent, Scribe and Pastry. Our MCFTP implementation uses Scribe and Pastry as underlying ALM. We compare the performance of MCFTP with BitTorrent.

Chapter 3

Multicast File Transfer Protocol - MCFTP

The Multicast File Transfer Protocol (MCFTP) is designed to disseminate data efficiently using multicast mechanisms. It supports not only native IP Multicast (see Section 2.2) but also Application Level Multicast (see Section 2.4). MCFTP consists of nodes that send messages to different multicast groups. All components of MCFTP will be introduced in this chapter, starting with a short overview of MCFTP presented in Section 3.1, followed by the different MCFTP variations shown in Section 3.2. The nodes are presented in Section 3.3, and the messages in Section 3.4 and the multicast groups in Section 3.5 respectively. Finally, the different strategies for the two approaches of MCFTP are depicted in Section 3.6 and an example is shown in Section 3.7.

3.1 Overview

The main goal of MCFTP is to distribute files to every node inside the MCFTP swarm as efficiently as possible. MCFTP divides the whole file into chunks. The dissemination of a file is done by disseminating these chunks. We used a predefined configuration file with a size less than 1kB which has to be gathered prior entering the MCFTP swarm. Basically the configuration file tells how to connect to the MCFTP swarm by predefining the Multicast address and port. When a node starts MCFTP, it will join to a predefined multicast group (file management group) for each file being shared. This group is only used to communicate among nodes, i.e., to exchange information and announcements. No data of the file will be sent on this group. For each chunk, there will be a new multicast group (sending group). As soon as a regular node, which is interested in downloading and/or sharing the file, has joined the file management group, it will start sending status messages periodically. Depending on the chosen approach, either regular nodes or a file leader node will create keep alive messages, which contain information about which multicast group a specific chunk will be sent to. When a keep alive message has been received, a regular MCFTP node will decide whether it will join to a multicast group to gather the announced chunk or whether it has to send the announced chunk to the specified multicast group. A sender creates a chunk message and starts sending it to the multicast group without joining it, while a node who wants to receive data will only join to the multicast group. When a node has received the whole chunk message, it will leave the multicast group and create a new

status message announcing an updated status about the missing chunks. This cycle is repeated until the file is distributed to every node inside the MCFTP swarm.

3.2 MCFTP Modes

MCFTP can run in two different modes. One uses native IP Multicast and the other uses Application Level Multicast. Those modes are influencing the MCFTP algorithm and will be discussed in this section. Also, there are different approaches concerning the behavior and the set up of MCFTP. Two of them will be depicted further in this section.

3.2.1 Network Modes

When a MCFTP node is started, it will join to a predefined multicast group. Depending to the running mode, this will either be an IP Multicast address in case of the native IP Multicast mode or it will be a bootstrap MCFTP node's IP address in case of the Application Level Multicast mode. The choice of the mode has an influence on the addressing and creation of multicast groups. It has no effect though on the handling of messages exchanged in the MCFTP swarm. If native IP Multicast is supported by the underlying network for all MCFTP nodes, it is recommended to use the native IP multicast mode of MCFTP. The reasons why to prefer native IP Multicast over Application Level Multicast were discussed in Section 2.2. If there is no native IP Multicast support for all the MCFTP nodes, MCFTP can still be used with the Application Level Multicast mode.

Native IP Multicast MCFTP

For the sake of simplicity, the native IP Multicast mode in MCFTP will be abbreviated with MCFTP-IPMC. In MCFTP-IPMC, all communication between MCFTP nodes is done using the IP Multicast paradigm. Thus, all MCFTP nodes have to be able to communicate to each other over native IP Multicast. MCFTP-IPMC is straight forward and does not have any specific setup, except the usage of native IP Multicast.

Application Level Multicast MCFTP

The Application Level Multicast mode in MCFTP is abbreviated with MCFTP-ALM. In this mode, an underlying ALM framework is used. MCFTP-ALM will act the same way as it would with MCFTP-IPMC, but using IP addresses of other nodes instead of IP Multicast addresses. When joining a multicast group or sending to a multicast group, MCFTP will pass the MCFTP message with the IP address and port of the multicast group to the underlying ALM framework, which will take care of the connections. Figure 3.1 is an adaption of the example used in Section 2.4.2, which shows how MCFTP-ALM mode would look like in case of using Scribe / Pastry as the Application Level Multicast framework. MCFTP-ALM nodes do not have to care about the handling of messages on the ALM level.

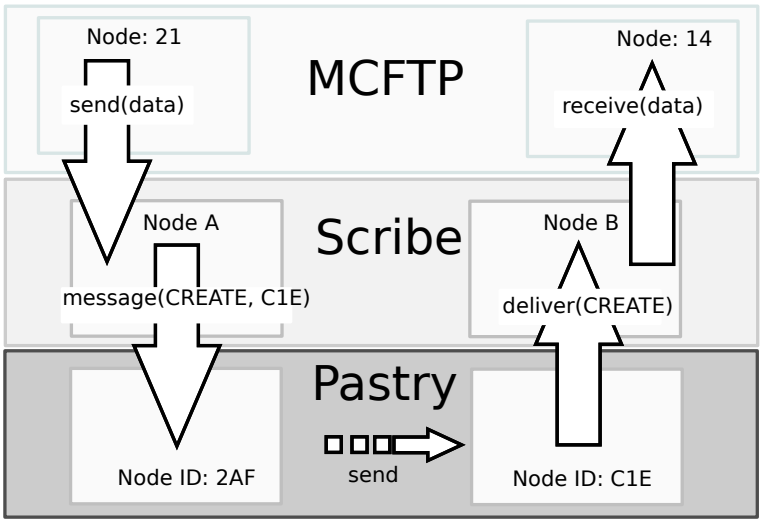


Figure 3.1: MCFTP-ALM runs on top of Scribe/Pastry P2P/ALM framework

3.2.2 MCFTP Swarm Establishment

There are two different set up approaches implemented in MCFTP. These approaches influence the behavior of the nodes and handling of received messages. The choice of the approaches also has an impact on the strategies to prefer. The centralized approach has one file leader node and the rest are regular nodes, whereas in the decentralized approach the MCFTP swarm is homogeneous with regular nodes. The main difference between the two approaches is the responsibilities the nodes are given.

Centralized MCFTP Approach

cMCFTP is the abbreviation for the centralized MCFTP approach, which consists of one file leader node and multiple regular nodes. The file leader node is the only node listening to status messages sent by regular nodes and it is as well the only one that creates keep alive messages. Therefore, it is the only one deciding what should be announced in a keep alive message. The strategy which chunks to announce in a keep alive message has to be defined only by the file leader node. That means, the file leader is the single instance, which has the responsibility to disseminate data as efficiently as possible. The regular nodes do not have any influence on the strategy, and therefore are not able to choose specific chunks to be sent by other nodes. The only choice regular nodes have, is that they can decide which chunk they want to download that was announced in a keep alive message. Figure 3.2 illustrates the handling of messages in cMCFTP. While the file leader node listens only to status messages, regular nodes only listen to keep alive messages.

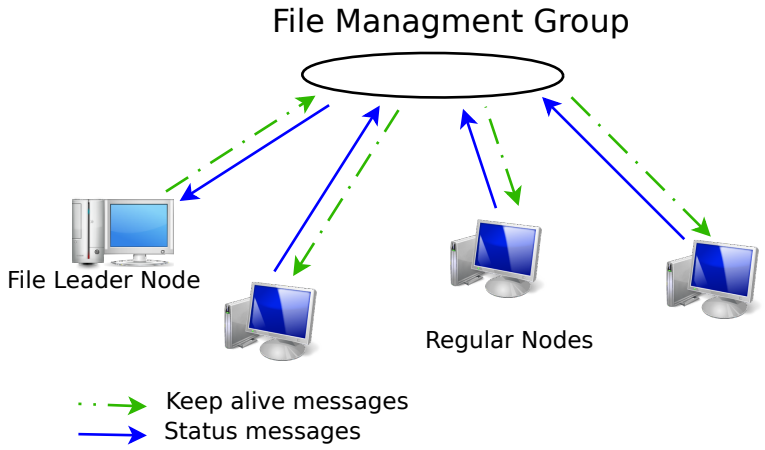


Figure 3.2: Regular nodes send status messages and file leader nodes send keep live messages

Single Point Of Failure

Having the whole responsibility at the file leader node is actually a single point of failure situation. If the file leader node would have any connection problems or technical problems, the whole MCFTP swarm would starve, since there would be no more announcements by the file leader node. There are multiple solutions for this problem which will be discussed in 3.2.2. The file leader node is sending keep alive messages periodically, and therefore these messages are received by all regular nodes. The detection of the failure of the file leader node is predictable as soon as there are no more keep alive messages sent in the MCFTP swarm. If such a failure is detected a new file leader node has to be elected. One solution to the single point of failure situation could be that as soon as a failure of the file leader node is detected, a regular node would take over the the responsibility of creating new keep alive messages, and thus start collecting status messages ans send keep alive messages.

File Leader Election

Theoretically, any node, which has a collection of a few status messages should be able to calculate a new keep alive message with chunk announcements. The relevance and quality of the chunk announcements are depending on the amount of available status messages. If a potential new file leader received a status message by all regular nodes, the chunk announcements will be optimal. An announcement can only be as good as the amount of different status messages received. A keep alive announcement can only consider regular nodes which have sent a status message. Different election methods could be applied in this case. A very basic and most likely not the best election method could be to elect the node with the lowest node ID as the new file leader node. A more advanced election could consider a predefined list of regular nodes, which would have to take over the responsibility of the file leader node one after another. These two solutions are optimistic and assume that there rarely is a failure of the file leader node. These two solutions would wait until there is a failure and start acting afterwards. Another possible way to elect a new file leader node is to have a few regular nodes acting additionally as backup file

leader nodes by storing status messages, but they do not create new keep alive messages. They would be running in the background and only become active as soon as there are indications that there was a failure at the original file leader node. This solution would be pessimistic, since there is not enough confidence in the file leader node, and multiple regular nodes would be collecting status messages additionally. There are a lot of different approaches on how to elect a new file leader node. Most importantly one needs to consider reliable and stable nodes with a good network connectivity as potential new file leader nodes. In our prototype version of MCFTP our file leader node was always predefined and known by all participating nodes inside the MCFTP swarm. We did not provoke any failure of the file leader node.
The election procedure for MCFTP is yet undefined. One way could be to use pessimistic approaches and predefine a list of follow up file leader nodes and distribute this list to every node in the MCFTP swarm. For the optimistic approaches one would need to define a new message type and maybe also a new multicast group, where only potential new file leader nodes would join and interact with each other in a failure situation.

Decentralized MCFTP Approach

The decentralized MCFTP approach is abbreviated with dMCFTP and consists only of regular nodes. This approach has different parts in common with the cMCFTP approach: regular nodes send periodically their status messages and also listen to keep alive messages. But, in dMCFTP, all regular nodes will listen to every status message sent by any other regular node.

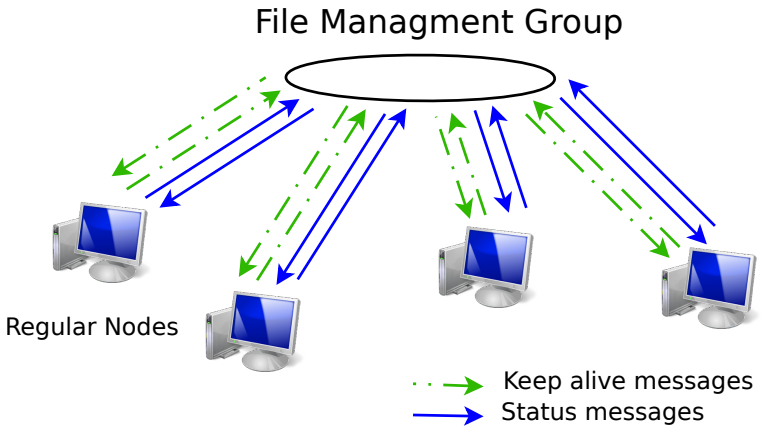


Figure 3.3: Status messages and keep alive messages are sent by regular nodes and also processed by all regular nodes

As soon as they can send a chunk to a multicast group, they will create a keep alive message with their announcements. The only way to ask for chunks is to tell the whole MCFTP swarm, which chunks they are missing. This is done by sending a corresponding status message. All regular nodes will decide by themselves which chunks to announce. This is one of the main differences compared to the cMCFTP approach. The responsibility of disseminating data efficiently is now distributed to every single regular node. There is not anymore a single instance deciding and controlling the announcements of chunks. Therefore, the strategy in dMCFTP differs to the

strategy of cMCFTP, which will be discussed in Section 3.6. Handling of messages in dMCFTP is illustrated in Fig. 3.3. Every regular node sends and receives status messages and keep alive messages.

3.3 MCFTP Nodes

There are two different kind of nodes in MCFTP. A file leader node has a special task assigned to coordinate between regular nodes, while a regular node is disseminating and gathering data. A file leader node is not being used in the decentralized approach (see Section 3.2.2).

3.3.1 Regular Node

A regular node is interested in sharing a file. If a regular node holds the whole file, it is referred to as a seeder, otherwise it is referred to as a leecher. In the beginning, every regular node joins the predefined file management group to communicate with other nodes. Every regular node sends periodically a status message to inform about, amongst others, which chunks are missing. Whenever a keep alive message arrives, a regular node will read it and decide for each of the containing announcements what to do. Either the regular node can join the multicast group to receive a missing chunk or one of the announcements selected the regular node to send a chunk to a multicast group. If a regular node was selected as a sender, it must create a new multicast group on the given address and port and start sending the selected chunk with the given sending rate. A new multicast group is created for every chunk and bandwidth. Other nodes will then be joining to the newly created multicast group and will receive the missing chunk.

Decentralized Approach vs. Centralized Approach

In the decentralized approach of MCFTP, there are only regular nodes and each of them locally manages an *allNodesInfo* table, which stores information gathered from the status messages sent by every regular node. Table 3.1 shows an example of an *allNodesInfo* table.

allNodesInfo

IP	ID	Chunks	First Port	Bandwidth Up (Total)	Bandwidth Up (Free)	Timestamp
130.92.70.81	2	2, 4, 27, 31	7020	60	40	1263128511
130.92.70.75	21	2, 15, 19, 21	7210	80	60	1263127623
130.92.70.77	12	5, 12, 27, 31	7120	50	50	1263125213
130.92.70.88	5	2, 5, 12, 27	7050	90	20	1263129147
130.92.70.65	11	2, 15, 21, 31	7110	60	40	1263128295
…	…	…	…	…	…	…
…	…	…	…	…	…	…

Table 3.1: An example of an *allNodesInfo* table

It stores all information extracted from status messages, such as IP address, node ID, chunks that are available, and bandwidth information of a node. Every time a node receives a status message, the *allNodesInfo* table is updated with the corresponding information and the time stamp is adapted to reflect the arrival time of the latest status message. The *allNodesInfo* table

is the base to calculate a new keep alive message. How to calculate a new keep alive message will be discussed in Section 3.6. Only in the decentralized approach of MCFTP, regular nodes will be collecting status messages and extract information into the *allNodesInfo* table. In the centralized approach of MCFTP, regular nodes will ignore all incoming status messages and let the file leader node collect the information.

3.3.2 File Leader Node

A file leader node is used only in the centralized approach of MCFTP. It is the node, which coordinates all other nodes. Thus, a file leader node is designed to collect information of all other nodes. All information of status messages are stored in the *allNodesInfo* table the same way as described in Section 3.3.1. The file leader node has a special role, since it is the only node that has the whole responsibility to manage the distribution of a file efficiently. There is always only one instance of a file leader node at a time active in a MCFTP swarm.

3.4 MCFTP Messages

In MCFTP, there is a total of three different message types: keep alive messages, status messages and chunk messages. Keep alive messages and status messages are referred to as control messages and do not include any data of the original file being shared. They are only used as an information channel. Chunk messages include parts of the file being shared, and thus they are referred to as data messages. Any MCFTP message is built the same way for both MCFTP modes (native IP Multicast and ALM) and both MCFTP approaches (centralized and decentralized). Since control messages are sent frequently, they have to be as small as possible in order to reduce overhead. Implementation details, such as size, content and overhead of a message will be illustrated in Chapter 4.

3.4.1 Status Messages

Status messages are sent periodically by regular nodes to inform the MCFTP swarm about their status. This includes information about network bandwidth, network address, user identification, and most importantly about currently available chunks. As soon as a regular node joins the MCFTP swarm, it first checks if there are any chunks already available for the given file, then creates a status message and sends it immediately to the file management group. After the initial sending of a status message, the node will create new status messages every *SM_Interval* seconds. Additionally, a new status message is created whenever a chunk has been successfully downloaded. All information inside a status message, except the information about the chunks, is static and is not going to change. Therefore, it would be enough to only send a difference to the previous status message. This would reduce the size of a status message and ultimately save bandwidth. One downside of this idea is related to the single point of failure situation described in Section 3.2.2. Whenever a new file leader takes the lead it will not be able to create an appropriate keep alive message, when it does not have enough full status messages received. A good workaround would be to increase *SM_Interval* and therefore keep sending

full status messages periodically, but to create partial status messages whenever a new chunk is downloaded successfully.

3.4.2 Keep-Alive Messages

In the decentralized approach of MCFTP, keep alive messages are sent periodically by regular nodes. In the centralized approach, they are sent periodically by the file leader node. Keep alive messages contain chunk announcements. The amount of announcements varies depending on the strategy, the MCFTP approach and the available resources in the MCFTP swarm. One chunk announcement contains information about the sending group, the chunk which will be sent to the sending group, the bandwidth with which the chunk will be sent, and the node that has to send the chunk. A keep alive message must have at least one announcement and has an upper limit of *maxChunkInfoInKAM* announcements per keep alive message, where the *maxChunkInfoInKAM* is set to 10 by default to avoid too big messages on the file management group. A keep alive message will be created every *KM_Interval* seconds. In the cMCFTP approach, the file leader node is the only node sending keep alive messages, and the *KM_Interval* should be set as low as possible to create as many chunk announcements as possible. Nevertheless, it should not flood the MCFTP swarm with useless keep alive messages when there are not enough free resources to send or even to receive chunks. A good choice of the *KM_Interval* should depend on the strategy and the approach chosen for a MCFTP swarm. In the dMCFTP approach, chunk announcements always have to have the own node as the sender of a chunk, due to security reasons as pointed out in Section 3.2.2. Since a regular node always knows about its own resources, it would not be optimal if a regular node would have to wait until it is asked by the *KM_Interval* period to create a new keep alive message. It would be optimal to create a new keep alive message as soon as there are enough resources, for example just after a finishing sending a chunk message. Therefore, the influence of the *KM_Interval* period is rather small, once the regular node has a continuous way of creating keep alive messages. But, the *KM_Interval* period is still needed, to guarantee that there will be at least an attempt to create a new keep alive message once every *KM_Interval* seconds, to avoid silent starvation of the MCFTP swarm.

3.4.3 Chunk Messages

When a node announces a chunk using a keep alive message, the sender will split the announced chunk into *subChunkLength* sized parts and create a new chunk message for each of them. These parts are referred to as sub chunks. They only include data of the size of *subChunkLength* Bytes. A chunk message stores information about the current chunk and the sub chunk included in the current chunk message plus the actual data of the sub chunk. Chunk messages are only sent by regular nodes when they are asked to do so by a chunk announcement inside a keep alive message. The *subChunkLength* should not be too large, since it would create big sub chunks which might lead to high delays, whereas too small sub chunks might lead to too many chunk messages being created. Another variable is the *amountOfChunks*, which defines the amount of chunks a file should be cut into. The size of a whole chunk depends on the file size and the *amountOfChunks*. The *amountOfChunks* should be adapted so that a whole chunk will not be

too big. On the other hand, the *amountOfChunks* should not create too many small chunks which then would lead to too many chunk messages being sent.

3.5 Multicast Groups

In MCFTP, there are two different kind of multicast groups, one is the file management group and the other is the sending group. In MCFTP-ALM, all multicast groups are based on the multicast mechanism of the underlying ALM implementation. In MCFTP-IPMC, these groups are all native IP multicast groups. The functionalities and roles of the MCFTP multicast groups are abstracted and are not influenced by the choice of the multicast mechanism. To separate data messages from control messages there are two different kinds of MCFTP multicast groups. Implementation details on the MCFTP multicast groups will be illustrated in Chapter 4.

3.5.1 File Management Group

The file management group is the main multicast group of the whole MCFTP swarm. All control messages are sent to the file management group and every node that is interested in either sharing or managing a file has to join the file management group. There is one file management group for each file. Every node that joined the file management group will send control messages periodically. These control messages are received by every other node that joined the file management group. But not all nodes will interact with the control messages the same way. In dMCFTP, every regular node will only listen to their own keep alive messages and all other status messages. In cMCFTP, the file leader node is the only one processing status messages and all regular nodes will listen to keep alive messages sent by the file leader. In MCFTP-IPMC, the file management group address is a predefined native IP multicast address for a file. In MCFTP-ALM, the file management group has to be predefined as well. But, it is not a native IP multicast address, it is a predefined bootstrap address. This is usually the IP address with a predefined port of the first node joining the MCFTP swarm. Every node is subscribed to the file management group until the MCFTP application is terminated. In our prototype implementation of MCFTP we predefined the file management groups address in a configuration file. It is still an open topic how to predefine these addresses and make them known by all participant nodes inside the MCFTP swarm. This has to be considered when creating an real world application.

3.5.2 Sending Groups

Sending groups are multicast groups, which are used to disseminate the file data. There are multiple sending groups for one file. For each chunk announcement in a keep alive message, there will be a new sending group. Each chunk announcement includes details about the sending group, such as the multicast address and port. Every regular node interested in receiving the chunk described in a chunk announcement will join to the announced multicast group. The announced sender does not have to join the sending group, it is enough to send chunk messages to it. Therefore, all nodes that joined a sending group are also interested in getting the chunk messages. When a node has all parts of a chunk, it will leave the sending group. A sending

group can be reused as soon as all regular nodes have left the sending group and the sender has stopped sending chunk messages to the specific sending group.

3.6 Strategies

There are two different approaches for MCFTP. One approach treats all nodes equally, which is the dMCFTP approach. The other approach has a dedicated file leader node, which takes care of the file dissemination management and the other nodes are all regular nodes. This is the cMCFTP approach. Both approaches use different strategies to disseminate data as efficiently as possible which will be the topic of this section. The efficiency of data dissemination depends mainly on the strategy. The strategies presented here are only a selection of many potential strategies. They are used to show different variations on how to potentially optimize MCFTP. To be able to announce sending groups, it is crucial to have at least a rough idea about the status of some regular nodes.

3.6.1 Centralized Mode

The file leader is a dedicated node, which is responsible for efficient data dissemination in the centralized approach of MCFTP. The file leader collects status messages in order to apply a strategy to disseminate data to the MCFTP swarm. When collecting status messages, a new entry is made in the *downloadQueue* table for each missing chunk reported in a status message. A *downloadQueue*-Entry includes information about the current maximum download bandwidth for a regular node. This can be different from the overall maximal download bandwidth. For each new status message for the same chunk, a counter is increased to show that there is another node requiring this chunk. The higher the counter is, the more nodes are missing the chunk. An example of a *downloadQueue* is depicted in Table 3.2.

downloadQueue

Chunk	List<Queue(Bandwidth, Counter)>
2	[120KB/s, 2], [85KB/s, 1]
21	[135KB/s, 3], [75KB/s, 4], [60KB/s, 2]
12	[185KB/s, 1]
5	[90KB/s, 6], [75KB/s, 3]
11	[80KB/s, 1]
…	…
…	…

Table 3.2: An example of a *downloadQueue* table

The file leader node has another table, the *allNodesInfo* table as described in 3.1. This table stores information filtered from the status messages. With the help of the *allNodesInfo* table, an

up to date list is always available containing chunks, which can be announced. The file leader is the only node with a strategy and the strategy has to work for all nodes. There is no other node which has any influence on the strategy than the file leader node. The goal of a dissemination strategy is to select the amount and order of chunks out of the *downloadQueue* and announce them in a keep alive message as efficiently as possible.

Basic Strategy

The basic strategy collects a subset of the *downloadQueue* table and another subset out of the *allNodesInfo* table with the available chunks. For each chunk, which is at least missing at one node and which is available at another node, a new chunk announcement is created by the file leader node and added into the keep alive message. This selection is done randomly. The chunk announcements are stored and the sending nodes are marked as busy, and therefore are not available for any further chunk announcement, until they finish sending the chunk. After a *KM_Interval* period, a new keep alive message is created the same way. This is repeated until there are no more chunks missing in the MCFTP swarm. A pseudo code of the basic strategy is given in Fig. 3.4. Note that *allNodesInfo* is sorted randomly for the basic strategy.

Figure 3.4 Pseudo-Code Basic Strategy

```
for all missingChunk in downloadQueue do
  for all potentialSender in allNodesInfo do
    if missingChunk.downBandwidth() ≥ potentialSender.upBandwidth() then
      if !potentialSender.busy() then
        sender = potentialSender
        sender.markBusy()
        addToChunkAnnouncements(missingChunk, sender)
        break
      end if
    end if
  end for
end for
```

Advanced Strategy

A more advanced strategy includes being aware of bandwidth limits and its impact on each regular node. The file leader node manages a list with nodes, which have already sent chunks. This list is consulted when new chunk announcements are created. Regular nodes, which did not send many chunks will be preferred in order to distribute the work load evenly on the MCFTP swarm. The performance and thus bandwidth is more important though, and therefore is considered first. Another aspect of the advanced strategy is to use nodes with high bandwidth capabilities to send chunks that are requested most, and to let slow nodes send chunks, which are not requested that often. The selection of high bandwidth capability nodes is part of the advanced strategy. For our implementation we used predefined limits to group between bandwidth capabilities. The

advanced strategy is based on the assumption that less requested chunks are available at many nodes. More nodes having a chunk results in a higher probability to find a node with modest bandwidth. But the more nodes request a chunk, the less this chunk is available at other nodes and thus the probability is low to find a node with a good bandwidth. The important part of the advanced strategy is to use the maximum number of nodes to send with maximum bandwidth but still to guarantee that prioritized chunks are considered with appropriate bandwidth. Prioritized chunks could either be chunks, which are missing for a small group of nodes to finish their download and become seeders as well, or it can be chunks, which are needed most by the whole MCFTP swarm. The definition of prioritized chunks is part of the advanced strategy. Grouping different nodes together according their bandwidth is as well another part of the advanced strategy.

3.6.2 Decentralized Mode

In dMCFTP, all nodes are equal and each of them has to decide what chunk announcements should be sent. Each of them follows a strategy to select chunks to announce. It is possible to set all strategies the same, so each of them will decide for themselves in the same manner. Even if a strategy is very good, it is still possible that the entire MCFTP swarm would disseminate data more efficiently, when there is a mixture of different strategies. Therefore, it is possible to give different strategies to different nodes, although they are all of the same kind. This is called meta strategy. A mixture of strategies would be for example, if nodes with an even node ID would use *strategy A* and nodes with an odd node ID would use *strategy B*. Or, another example of a meta strategy would be if the first 50 nodes would use *strategy A*, the next 25 nodes would use *strategy B* and the rest would use *strategy C*. Any node creating a new announcement is aware of its own status and it knows which chunks are available that can be announced. It also knows about its total bandwidth and its available bandwidth at a given point of time.

Basic Strategy

When using the basic strategy, a node collects status messages and knows which chunks are still needed by other nodes. It also knows about its available chunks. An intersection of those two collections is done in order to determine which chunks are available and are still needed by other nodes. Out of this intersection, a random chunk is chosen and announced to be sent using the full bandwidth available of the node. Another variation is to randomly select n chunks and to announce each of them with $\frac{1}{n}$ of the bandwidth available at the node. As soon as a chunk has finished sending a chunk, either one or two chunks will be selected and announced the same way again. This is done until there are no more chunks required by other nodes in the MCFTP swarm.

Announcement Aware Strategy

The basic strategy does not take note of other announcements, and therefore a chunk can be announced multiple times in a short period of time. This strategy can lead to redundant announcements. A node using the announcement aware strategy stores all recently received keep

alive messages and before announcing a chunk, it checks the recent announcements and compares if its announcement would be redundant. In this strategy it is also possible to announce n chunks with each using $\frac{1}{n}$ of the bandwidth available at a node instead of announcing one chunk with the full bandwidth available at a node.

Serve Fast Strategy

Using the serve fast strategy, a node is not only aware of other announcements, but it also knows the bandwidth capabilities of other nodes. The *allNodesInfo* table (described in Section 3.1) stores information about each node and their missing chunk. This table is used with the serve fast strategy in order to look for nodes with fast connections. For a chunk announcement, the *allNodesInfo* table is checked first and the table is sorted by bandwidth availability. Nodes are categorized according to their bandwidth availability. The nodes inside the category with higher bandwidth availability get a higher priority, and thus all nodes with high bandwidth availability are considered only when new chunk announcements are created. Inside such a category, all nodes are analyzed and a list with missing chunks is created. For each chunk, a counter is added in order to count how often the chunk is missing inside its priority category. The chunk with the highest count will then be announced. If there is no need for any node inside the highest priority category, the next category is consulted and only nodes inside this category are considered for new chunk announcements. Either, there is always one announcement with the full bandwidth available or there are multiple announcements with different bandwidth availabilities according to the priority. The idea behind this strategy is to disseminate chunks to "fast" nodes, which have higher bandwidth available to disseminate the data to further nodes in a second step. The more fast nodes become seeders the faster the MCFTP swarm will finish downloading a file.

Serve First Strategy

The serve first strategy is similar to the serve fast strategy. It also takes recent announcements into account. Categories are used as in the serve fast strategy, but this time, categories are grouped differently. The highest priority is given to nodes that have been participating in the MCFTP swarm for the longest period of time. The motivation for this strategy is to avoid long start to end times for nodes. The higher the priority is, the more chunks will be announced to match the requirements of these specific nodes. Thus, the earlier a node joins the MCFTP swarm the more "valuable" it becomes. As for the serve fast strategy, if there is no more chunk to announce for a priority category, the next category is chosen with the next highest priority. The higher the priority is, the higher the sending rate has to be.

Starvation Avoidance

In the previously described advanced strategies groups are prioritized. In such circumstances it is always possible that higher prioritized nodes only consider each other. Not prioritized nodes will not be considered anymore and therefore never finish gathering data. This situation is called starvation. To avoid starvation of lower prioritized groups, it is possible to announce n chunks in total. A total of o (where $o < n$) chunks for nodes with the highest priority using more than $\frac{1}{n}$

of the bandwidth available could be announced. And a total of p (where $p <= n$) chunks with each less than $\frac{1}{n}$ bandwidth available for nodes inside the second highest priority category could be announced. The same could be done for lower prioritized categories. Many variations are possible. We examined different approaches and finally decided to always consider at least one announcement from a lower prioritized category from time to time. It is also possible to always announce from different priority categories with different sending rates.

3.7 MCFTP Example

To illustrate the Multicast File Transfer Protocol, this section includes two examples of MCFTP with its two different approaches as described in Section 3.2.2. These examples include the whole process from bootstrapping MCFTP until every node has finished downloading a given file. For simplicity reasons, these examples are shown in the native IP Multicast mode as described in Section 3.2. The examples look similar at the first glance, but they are different when comparing the responsibilities and roles of nodes.

3.7.1 Centralized Approach

Figure 3.5 shows how the bootstrap is realized in cMCFTP mode. In the beginning, there is one file leader node (node A), which bootstraps the whole MCFTP swarm by joining a multicast group at the predefined IP Multicast address 225.1.2.3 and port 7000.

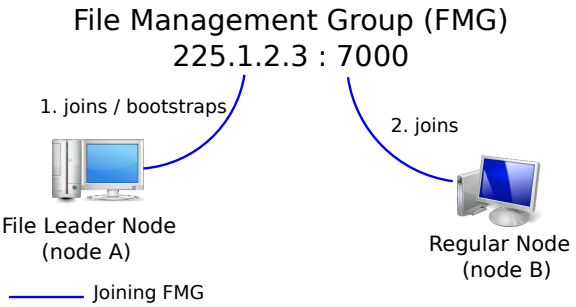


Figure 3.5: Bootstrapping of cMCFTP in native IP Multicast mode

The file leader and the multicast group address are predefined in a configuration file. In our evaluations our file leader node was always bootstraping the MCFTP swarm. This multicast group is the file management group, where all communication is done. Thus every participating node of the MCFTP swarm will have to join the file management group. In the example, a seeder joins the MCFTP swarm (node B) which then starts sending status messages periodically. The status messages indicate that node B has all 4 chunks of the file being shared. The file leader collects these status messages and checks periodically if there is another node in the MCFTP swarm which might need a chunk, which is available at node B.
In Fig. 3.6 a leecher joins the MCFTP swarm (node C) and also sends status messages to the file management group, the file leader node will also store its status messages. The status message of node C states that this node does not have any chunks yet.

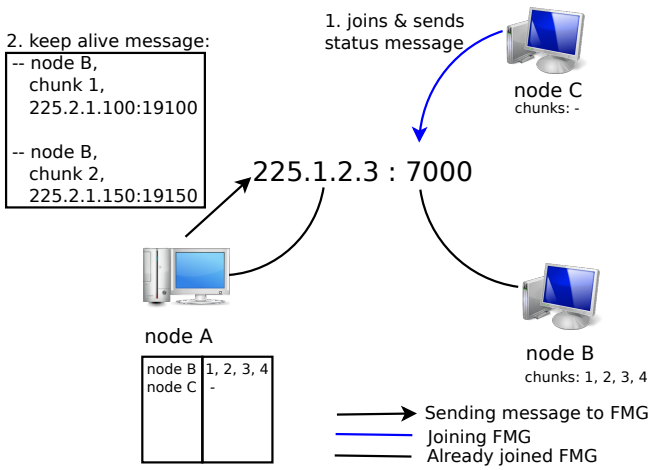


Figure 3.6: Keep alive message created by file leader node

The file leader node now calculates the best possible chunk announcements and adds them into a new keep alive message, which is then sent to the file management group. The calculation of keep alive messages depends on the strategy used. After nodes B and C have received a keep alive message, node C will join to the multicast group announced in the keep alive message. This is 225.2.1.100 on port 19100 for chunk 1 and 225.2.1.150 on port 19150 for chunk 2.

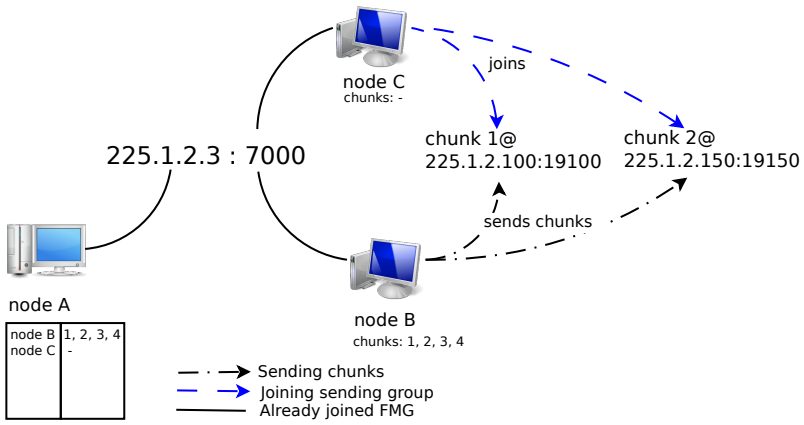


Figure 3.7: Node B sends chunks and Node C joins multicast groups

Node B will send chunk 1 and chunk 2 to the according multicast groups without joining to it as depicted in Fig. 3.7. When node C has finished downloading a chunk, it will leave the multicast group and create a new status message with its new information. According to the new status message, the file leader node will calculate a new keep alive message.

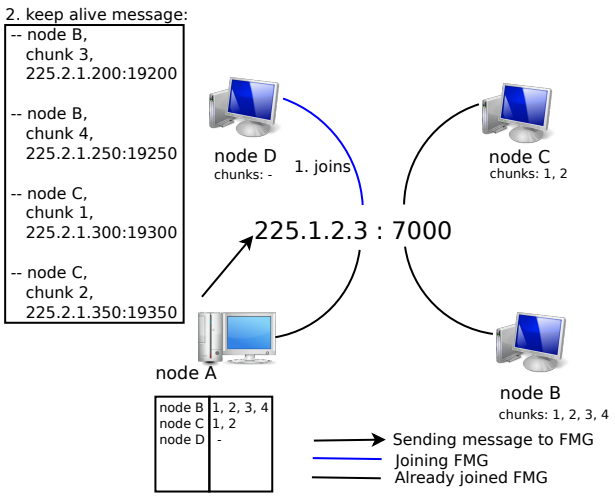


Figure 3.8: Node D joins the MCFTP swarm

When further nodes join the MCFTP swarm, node C can be chosen as a sender for chunks 1 and 2. Figure 3.8 shows the case when a new leecher having no chunks joins the MCFTP swarm.

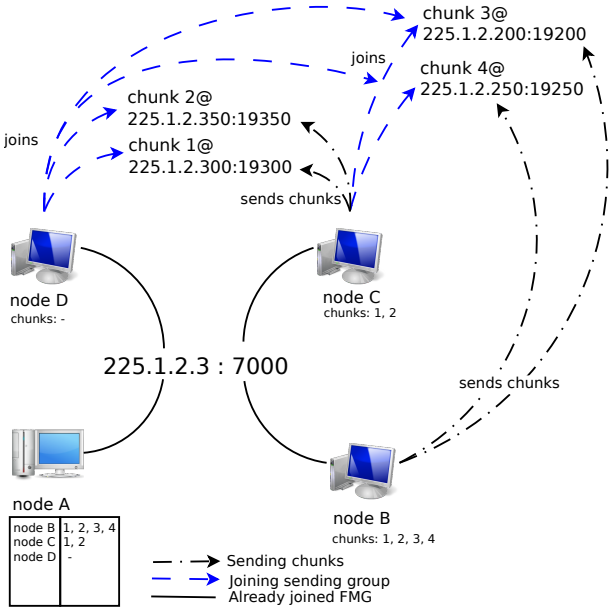


Figure 3.9: All nodes finish downloading the file

The file leader node then calculates a new keep alive message such that the whole network can profit the most. This is depicted in Fig. 3.9. Node D can download all four chunks and node C will download chunks 3 and 4. Hence, all nodes will have finished their downloads with only two announcements by the file leader node.

3.7.2 Decentralized Approach

In dMCFTP, there is no file leader node, thus a seeder (node A) bootstraps the MCFTP swarm on a multicast group at the native IP Multicast address 225.1.2.3 and port 7000. Status messages are sent periodically by node A to show its existence. This is shown in Fig. 3.10.

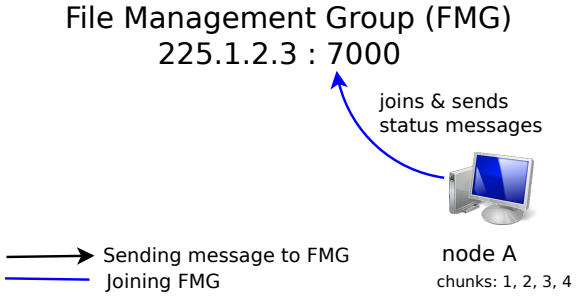


Figure 3.10: Bootstrapping of dMCFTP in native IP Multicast mode

When a leecher node joins the MCFTP swarm (node B), it will as well send status messages. But, this time node A will receive the status message and calculate a keep alive message according to a given strategy. The keep alive message with the announcements for chunk 1 and chunk 2 is then sent to the file management group as illustrated in Fig. 3.11.

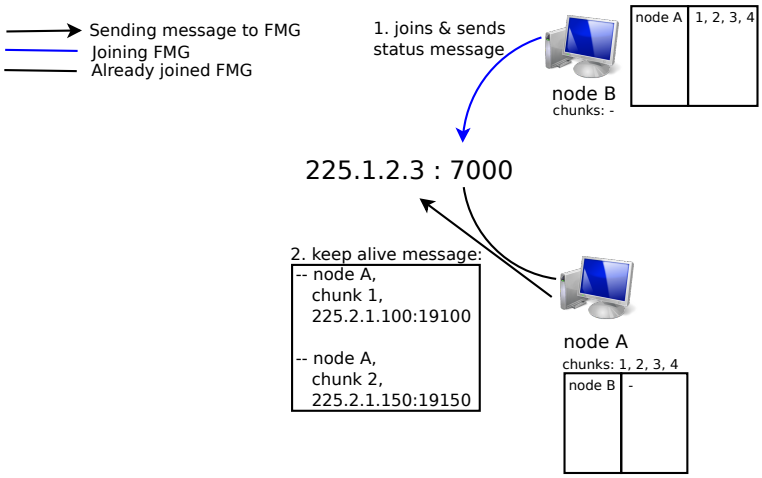


Figure 3.11: Keep alive message created by a regular node

Node A will then send chunk 1 and chunk 2 as announced in the keep alive message and node B will join the two multicast sending groups to receive the chunks as shown in Fig. 3.12. After finishing each chunk download, node B will announce its new status message. When another leecher joins (node C), it will send its status messages. Figure 3.13 shows how the MCFTP swarm looks like after node C joined. All other nodes will receive node C's status message and each of them will calculate new keep alive messages with their own chunks.

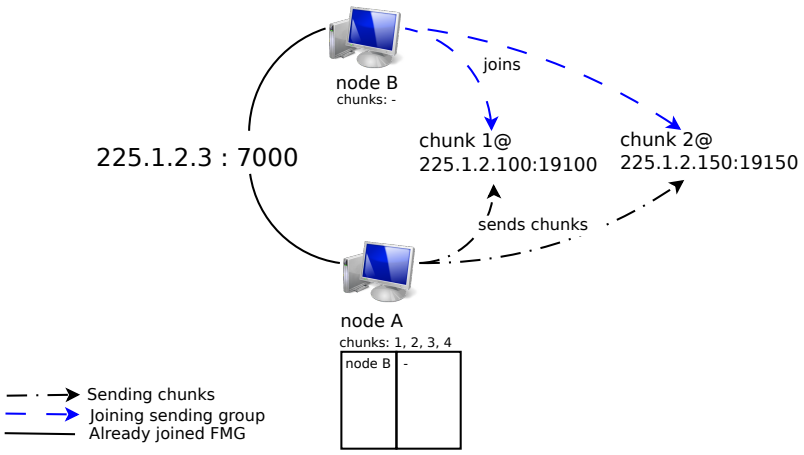


Figure 3.12: Node A sends chunks and Node B joins multicast groups

Thus, node B will be able to announce chunk 1 and chunk 2. Each node will decide with their own strategy which chunks to announce. And according to an advanced strategy, node A would announce chunk 3 and 4.

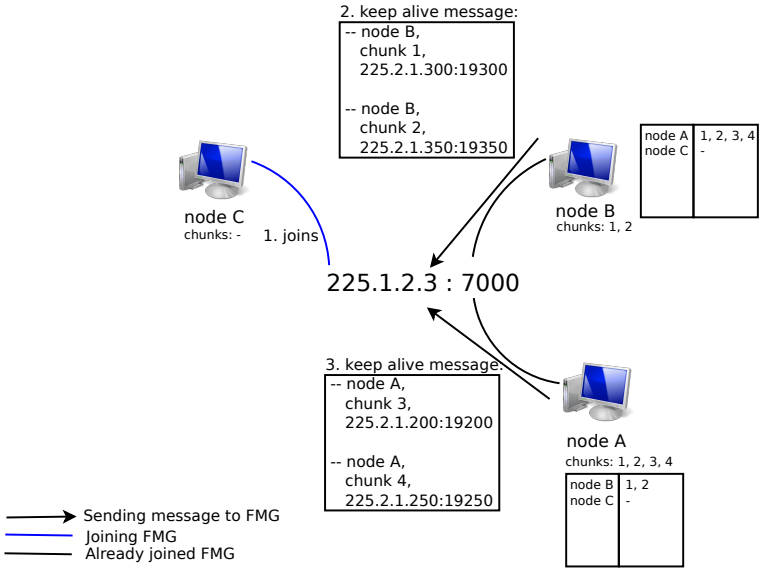


Figure 3.13: Node C joins the MCFTP swarm

Node C will join all 4 announced sending groups, whereas node B will only join the two multicast sending groups for chunk 3 and 4 as shown in Fig. 3.14. All nodes will finish downloading the file with a total of 3 keep alive messages.

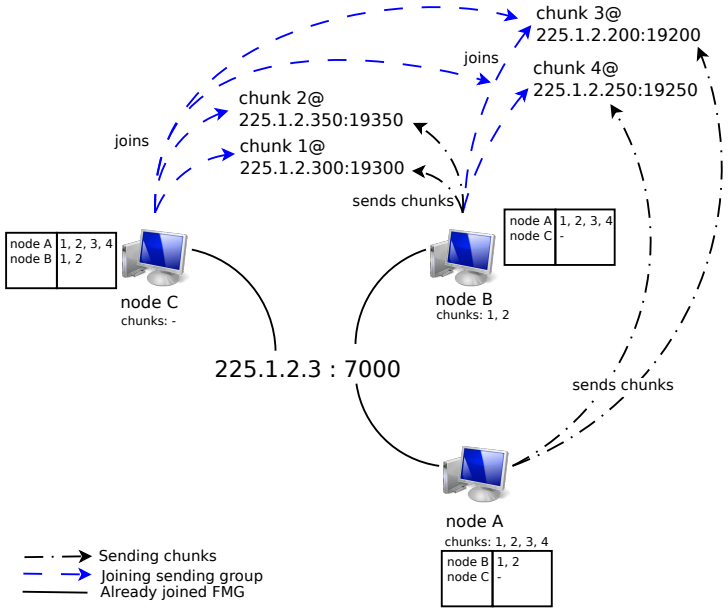


Figure 3.14: Node B and node C finish downloading all chunks

3.8 Summary and Conclusion

In this chapter, we presented the design of MCFTP. We explained the different MCFTP network modes using IP Multicast and Application Level Multicast. Additionally, we introduced two different set up approaches of MCFTP (centralized and decentralized approaches). We explained the different node types (regular and file leader based nodes), the different kind of messages (status, keep-alive and chunk messages) and introduced different types of multicast groups (file management and sending groups). We briefly outlined different strategies for determining which chunks to send and showed comprehensive examples of how MCFTP would work in some simple scenarios.

Chapter 4

MCFTP Implementation

In this chapter we discuss implementation details beginning with some general information presented in Section 4.1, followed by MCFTP variations implementation details shown in Section 4.2. Finally, the implementation of MCFTP messages is presented in Section 4.3.

4.1 Overview

MCFTP was implemented using Java (JDK 6 Update 16) [21]. As Application Level Multicast (ALM) 2.4 infrastructure, we used Scribe/Pastry (see Section 2.4.2), but MCFTP could support any ALM framework. We chose Scribe/Pastry because there exists already a mature implementation of these protocols in Java which is called Freepastry (Release 2.1) [22], which we used as the ALM framework for our prototype implementation.

Generally, the MCFTP protocol has been described in Chapter 3. Our implementation is a proof of concept and not a fully featured application. Hence, some characteristics are missing. There is no support for multiple files to download. We use a predefined configuration file with a size less than 1kB and the application will take care of that file only. There is also no implementation of a graphical user interface. Our prototype works as a command line tool only. A more basic missing feature is the absence of a hand over mechanism for a file leader node which was described in Section 3.2.2. Therefore, there could be a single point of failure in our current implementation. But still, all parts to have a working prototype are implemented. The only missing part is the handover of a file leader node, which was not needed in our experimental environment. The messages are implementation specific and were defined during this thesis.

4.2 MCFTP Modes

The implementation considers both MCFTP variations and the two different modes as described in Sections 3.2 and 3.2.2. There is one application, which is able to run in different modes using different approaches. The regular node and the file leader node inherit from a common super class called User as shown in Fig. 4.1.

Figure 4.1: Class Diagram of MCFTP nodes

A regular node could become a file leader node without any issue and take over the file leader role in a MCFTP swarm. Also, the two different MCFTP modes only differ in creation and usage of the different multicast addresses. The class diagram is shown in Fig. 4.2. When a new ALM framework is considered to be used for MCFTP, one might need to adapt the methods in *McftpScribeClient* to reflect the correct usage of the new ALM framework.

4.3 Sending Messages

The three types of MCFTP messages inherit from a common super class called *McftpMessage* as shown in Fig. 4.3. A keep alive message has many *KeepAliveChunkInfo* which we earlier referred as chunk announcements. Messages are sent frequently and repeatedly in a MCFTP swarm. Therefore, it is important to decrease them to a minimal size. The *KM_Interval* as described in Section 3.4.2 is set to 1 second. This means that nodes, which send keep alive messages will attempt to create a new keep alive message every second. This attempt may fail in the decentralized approach more often than in the centralized approach.

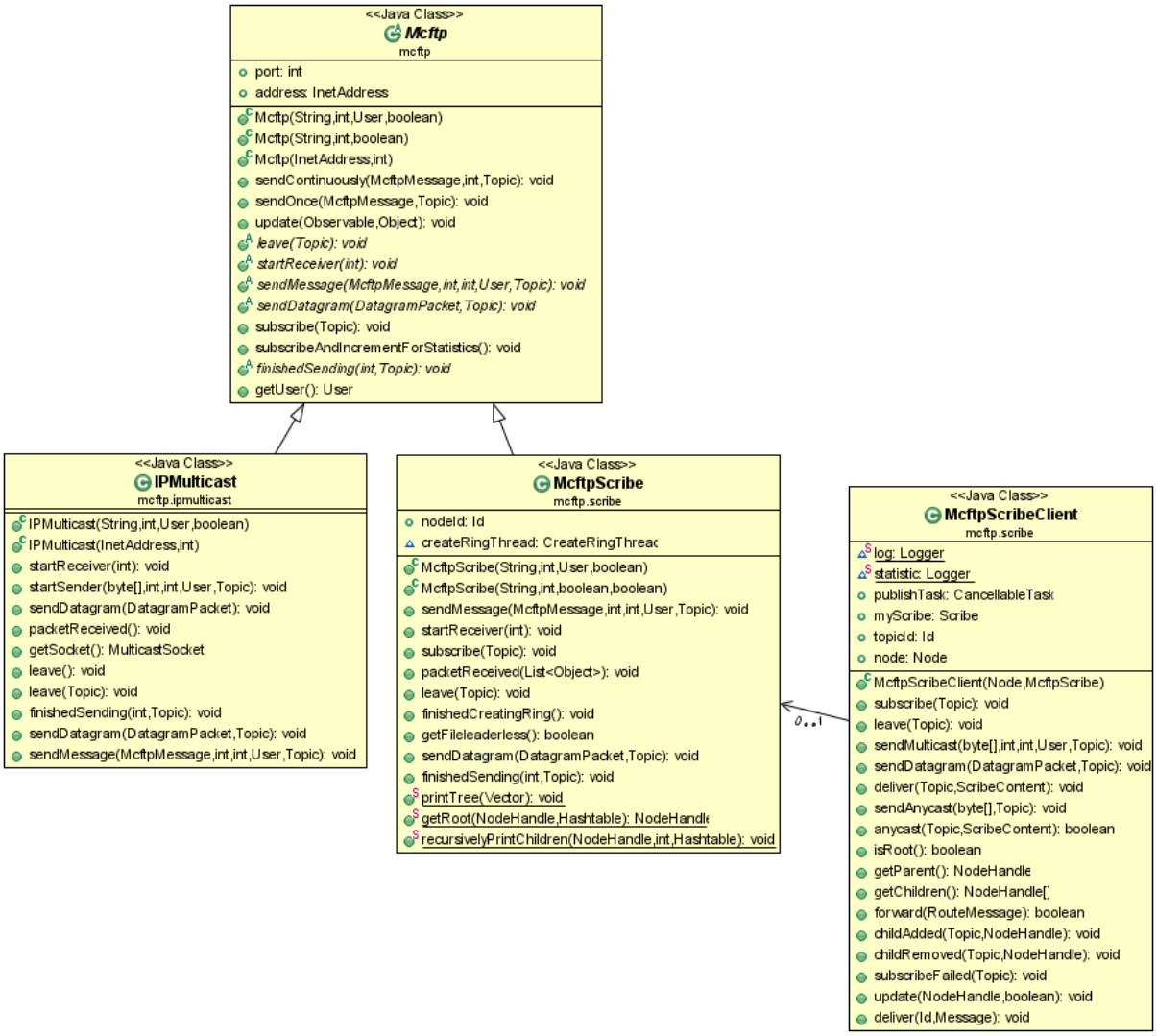


Figure 4.2: Class Diagram of MCFTP network modes

This is because in the decentralized approach, nodes can create keep alive messages only for themselves and there is a high probability that a regular node can not announce new chunks every second. Status messages are sent every 10 seconds and every time a chunk has finished downloading a chunk. Periodic sending of status messages is important in order to inform the MCFTP swarm about the presence of a node. This is especially important in the centralized MCFTP approach. In this implementation of MCFTP, a file is divided into *chunkAmount* equal sized chunks, where *chunkAmount* depends on the file size. When a big file is disseminated, the file is divided into more chunks. When the file to be disseminated is rather small, it is divided into less chunks. Using previous evaluations, we chose to divide a file such that each chunk size is ranging between 150kB and 300kB depending on the file size. A chunk is then divided

into subchunks with the *subChunkLength* length of 8kB each. This procedure was discussed in Section 3.4.3.

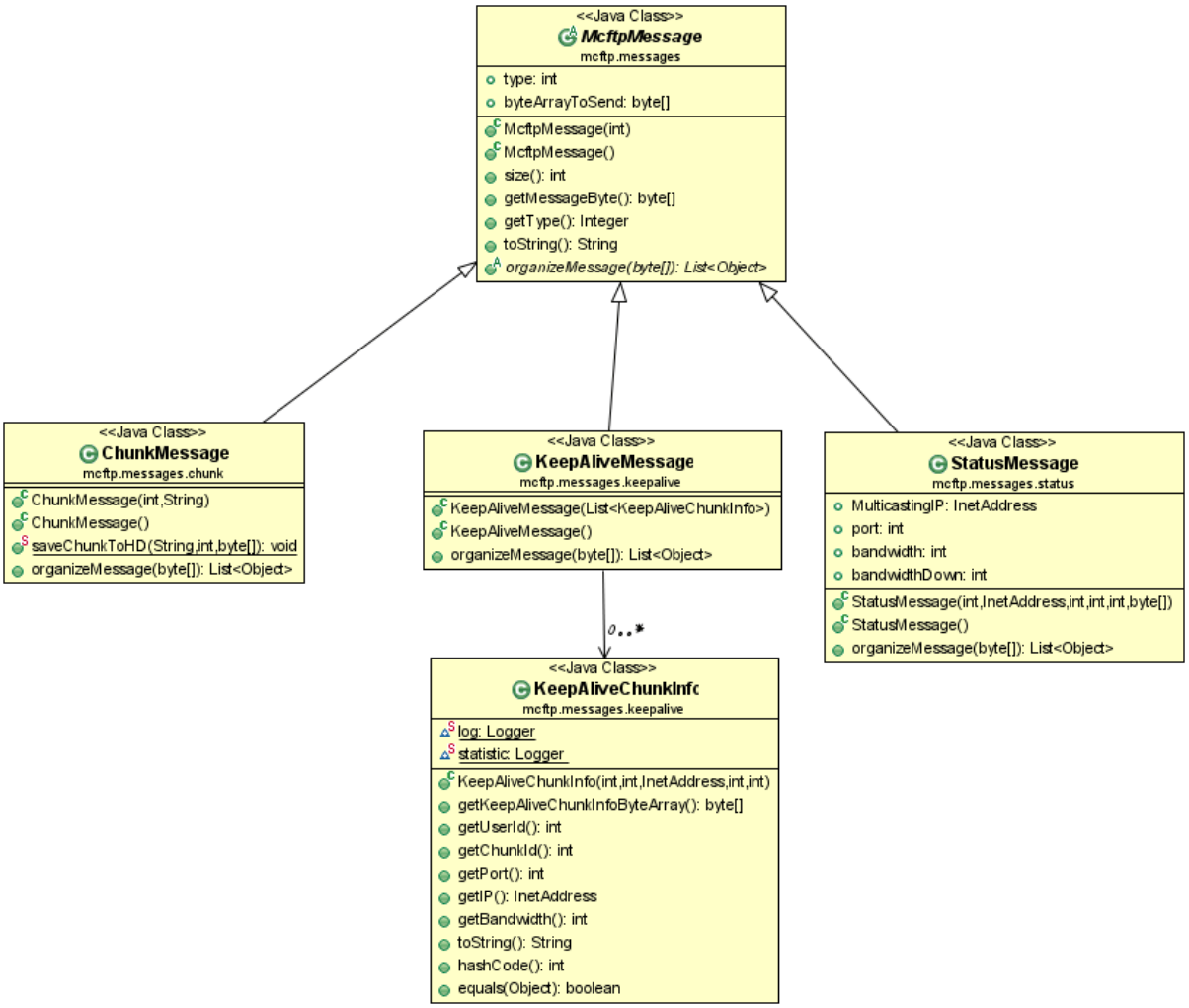


Figure 4.3: Class Diagram of MCFTP messages

4.3.1 Status Messages

In detail, a status message is constructed as illustrated in Fig. 4.4. The field *Type* is an identification number for the message itself. It has a size of one byte, which is fairly large enough since we only have three different message types in MCFTP. The user identification number is as well fit into one byte. This limits a MCFTP swarm to 256 users. The field *IP address* is the user's IP address and fits into four bytes. In MCFTP, every node needs to have a certain number of open ports, which can be used to share data with other MCFTP nodes. The given port in a status message is the first open port that can be used for this purpose. The field *Bandwidth* uses two bytes,

one byte for the available upload bandwidth and one byte for the available download bandwidth. 32 bytes are reserved for available chunk information corresponding to 256 bits using the field *Chunk Bitset*. For each chunk, there is one bit used. Thus, the maximum number of chunks is limited to 256.

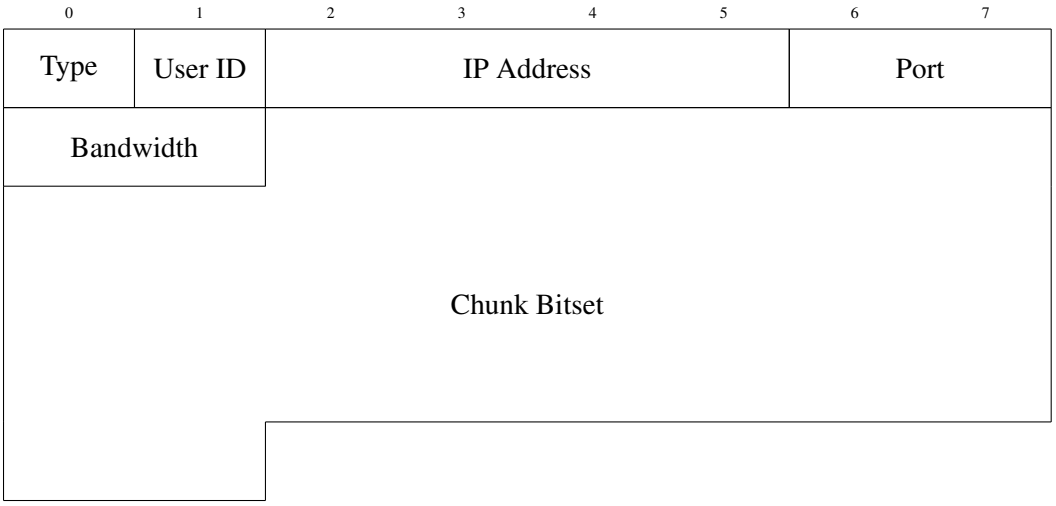


Figure 4.4: Structure of status messages

4.3.2 Keep Alive Messages

In detail, a keep alive message is constructed as shown in Fig. 4.5. The first byte is used for identification of the message *type* itself. For each chunk announcement nine bytes are used. These fields consist of the *User ID* (one byte) identifying the instance that has to send a given chunk identified by the field *Chunk ID* (one byte) using the given *Bandwidth* (one byte) and the corresponding multicast *IP Address* (four bytes) and given *Port* (two bytes). There is at least one chunk announcement per keep alive message but no more then ten. This results in a size of a keep alive message between ten bytes and 91 bytes.

4.3.3 Chunk Messages

A chunk is divided into sub chunks, which are then sent using individual chunk messages. The structure of a chunk message is presented in Fig. 4.6. Next to the message identifier in the first byte (field *Type*), the second byte presents the identifier (field *Chunk ID*) of the whole chunk. The third byte is a flag-byte (field *Next*) which is set when further sub chunks follow. It is not set when this message contains the last sub chunk for a given chunk. The last byte is the sub chunk identifier (field *Sub Chunk ID*), which is finally followed by the *Sub Chunk Data* field.

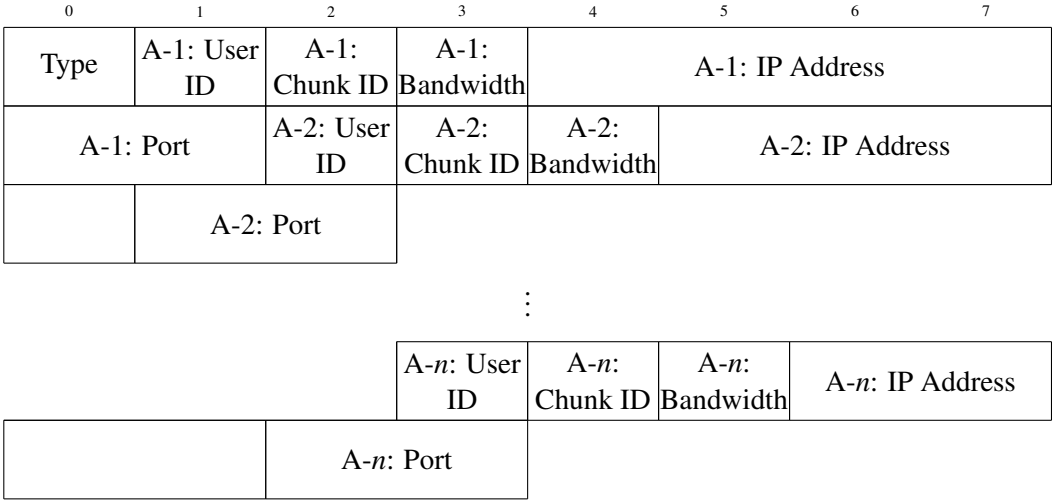


Figure 4.5: Structure of keep alive messages

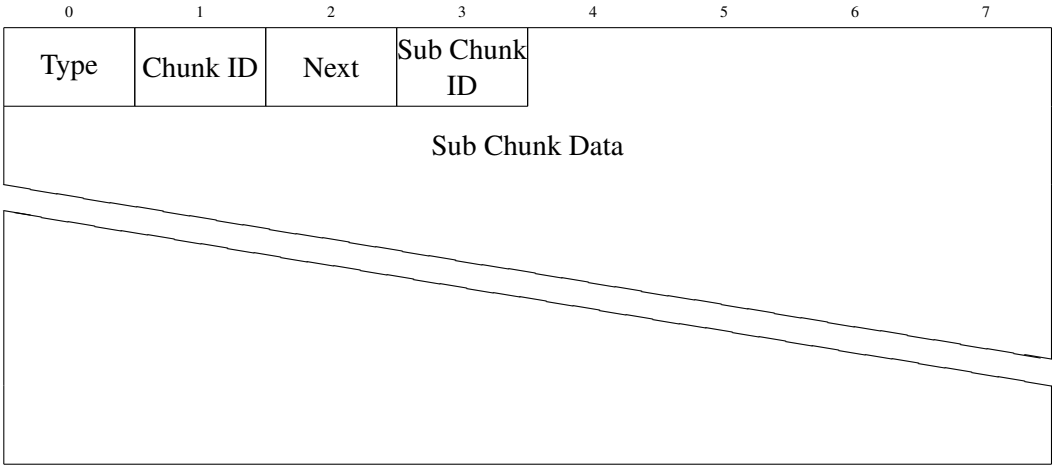


Figure 4.6: Structure of chunk messages

4.4 Application Level Multicast Groups

MCFTP-ALM is based on Scribe/Pastry as described in Section 2.4.2. A Scribe multicast group is referred to as a topic. In MCFTP-ALM, two Pastry rings are used. The first Pastry ring is used for the file management group. There is only one single topic on this ring and it is used only for exchanging control messages. The second Pastry ring has many topics, one topic for each announced chunk in the entire MCFTP swarm. This solution is not optimal because all regular MCFTP nodes are present in this Pastry ring for all chunk topics. This means that there are

Pastry nodes, which actually do not require a specific chunk, but still are used in the underlying Pastry ring as a forwarder for this chunk. This procedure is illustrated in Fig. 4.7. In the example, nodes C and D are not interested in a chunk, and therefore do not join to the according Scribe topic. But, since all nodes use the same Pastry ring, node C and node D could still be used as forwarders if required by the Pastry protocol. A more optimized approach would be to use one Pastry ring for each announced chunk. This would imply creating Pastry rings on the fly or in advance. We tried to use the more sophisticated second approach, but it used too many resources for our scenario setups so that the machines slowed down dramatically. That is why we had to use the first approach with a single Pastry ring for data exchange and multiple topics on that ring.

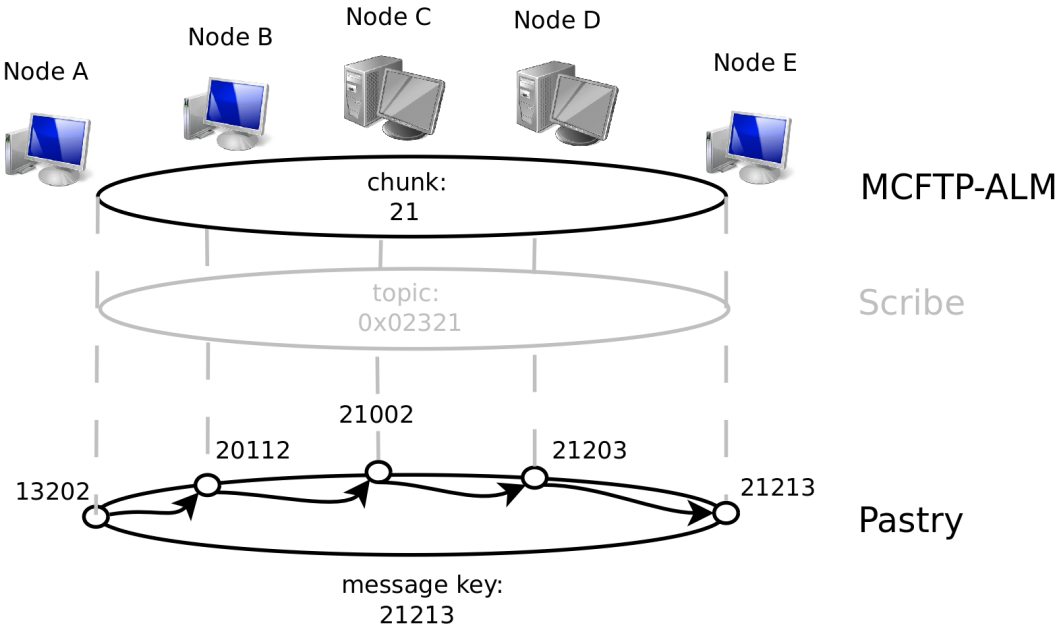


Figure 4.7: MCFTP-ALM nodes used as pure forwarders in a Pastry ring

4.5 Summary and Conclusion

In this chapter, we presented the implementation of our MCFTP prototype. We used Freepastry as our underlying Application Level Multicast protocol. The different variations of MCFTP were implemented and presented. Furthermore, we showed in detail the structure of the different MCFTP messages (Status, Keep alive and Chunk messages). Finally, we presented how MCFTP-ALM nodes in our Freepastry based implementation are used as forwarders. We showed that the MCFTP protocol and its different variants can be easily implemented and that it is also possible to integrate existing ALM implementations to be used with MCFTP.

Chapter 5

Evaluation

The different approaches and modes of MCFTP have been evaluated and compared to BitTorrent using different scenarios. In this Chapter, various results will be illustrated and discussed. In the beginning the testbed will be introduced in Section 5.1. The basic scenarios will be presented in Section 5.2, followed by the BitTorrent client used to compete against MCFTP shown in Section 5.3. Finally, the results of the comparisons will be discussed in Sections 5.4, 5.6 and 5.5.

5.1 Testbed used for Evaluation

We used the infrastructure provided by the research group "Computer Networks and Distributed Systems" of the University of Bern, which offers 26 nodes distributed over three racks. Each rack uses its own switch and all three switches are interconnected. The 26 nodes do not provide the same performance. Some nodes have an Intel Pentium D processor with a clock rate of 3 GHz and 2 GB of RAM while others have Intel Core 2 Quad processors with a 2.8 GHz clock rate and a total of 8 GB of RAM. These nodes are used by several students and thus are not used evenly at different times. For our evaluations, we limited our scenarios to 15 nodes, which were not heavily used. Due to the fact that the values evaluated are not generated in a dedicated simulation environment, we removed 5% of the outliers from all results presented in this chapter.

5.2 Evaluation Scenarios

In our evaluation scenarios, one file was distributed at a time. We tested a total of two different files sizes. The first being a Firefox tar-ball with a size of approximately 8.5 MB. The second being a "Damn Small Linux" image with a file size of approximately 50 MB. The maximum bandwidth assigned for incoming traffic was limited between 60kBps and 180kBps. The maximum outgoing traffic was always $\frac{1}{3}$ of the maximum incoming traffic. We chose the $\frac{1}{3}$ factor because this is currently the most offered setup by ISP's in our region. The assigned maximum bandwidth for each node was uniformly distributed in the given range. Evaluations have been performed using one seeder node and multiple leecher nodes, where the amount of leecher nodes was varying between 10, 20, 30, 50 and 100 nodes. Other evaluations were performed varying the number of initial seeders between 1, 2, 3 and 5 with the total number of nodes set to 20 and

50. Each scenario was executed 10 times with different assigned bandwidths for the nodes in the swarm. The main focus of our experiments was on scenarios where there is only one seeder and the rest of the nodes do not yet have any chunks of the file at all. This could be a valid case for a new and popular file, which is rare to find.
Our evaluations focused on the average download time for downloading one file for all nodes in the swarm. The measured start time is defined by the time a node joins the swarm and the measured end time is defined when all parts of a file are downloaded completely by a given node. Since we use different bandwidth values for the different nodes, we normalized the download values. The actual time spent to download a file was divided by the time, which could have been spent to download the file with full bandwidth usage(optimal download time, ODT). We called this normalized value *download factor*. The lower the *download factor* is the better the dissemination time is. For example, we assume a node took nine seconds to download a file of a size of 6000 bytes and its maximum download bandwidth was 1000 bps resulting in a best case value as follows:

$$ODT = \frac{6000B}{1000Bps} = 6s$$

The *download factor(DF)* would then result in:

$$DF = \frac{9s}{6s} = 1.5$$

5.3 BitTorrent Client used for Comparison with MCFTP

We let MCFTP compete against BitTorrent, which is also based on P2P mechanisms as described in Section 2.3. A current Internet study done by ipoque [23] claims that P2P generates by far the most traffic in all evaluated regions in the years of 2008/2009. A summary is shown in Table 5.1. The study also states that: “in all regions apart from South America, BitTorrent is the dominating protocol followed by HTTP.” We show an example of how popular BitTorrent is in Germany in Fig. 5.1. We used Azureus (now known as Vuze) [24] as a competing BitTorrent client. According to the ipoque study, Azureus/Vuze is the most popular BitTorrent client. We used the latest release of Azureus with command line support (version 3.0.3.0). This version was released in 2007, 4 years after the initial release of the first Azureus client. Although there are options to limit bandwidth for the Azureus client, they do not always stick and are omitted on a irregular basis. This was the reason why we also used trickle (version 1.06) [25] for our BitTorrent evaluations. Even with trickle, it was not always possible to limit the bandwidth for our BitTorrent evaluations. This problem occurred especially with evaluations using many nodes. For evaluations with a small file size, this problem was more crucial than for evaluations with a bigger file.

Protocol class proportions 2008 / 2009

Protocol Class	Southern Africa	South America	Eastern Europe	Northern Africa	Germany	Southern Europe	Middle East	South-Western Europe
P2P	65.77%	65.21%	69.95%	42.51%	52.79%	55.12%	44.77%	54.46%
Web	20.93%	18.17%	16.23%	32.65%	25.78%	25.11%	34.49%	23.29%
Streaming	5.83%	7.81%	7.34%	8.72%	7.17%	9.55%	4.64%	10.14%
VoIP	1.21%	0.84%	0.03%	1.12%	0.86%	0.67%	0.79%	1.67%
IM	0.04%	0.06%	0.00%	0.02%	0.16%	0.03%	0.50%	0.08%
Tunnel	0.16%	0.10%	-	-	-	0.09%	2.74%	-
Standard	1.31%	0.49%	-	0.89%	4.89%	0.52%	1.83%	1.23%
Gaming	-	0.04%	-	-	0.52%	0.05%	0.15%	-
Unknown	4.76%	7.29%	6.45%	14.09%	7.84%	8.86%	10.09%	9.13%

Table 5.1: P2P usage in 2008/2009

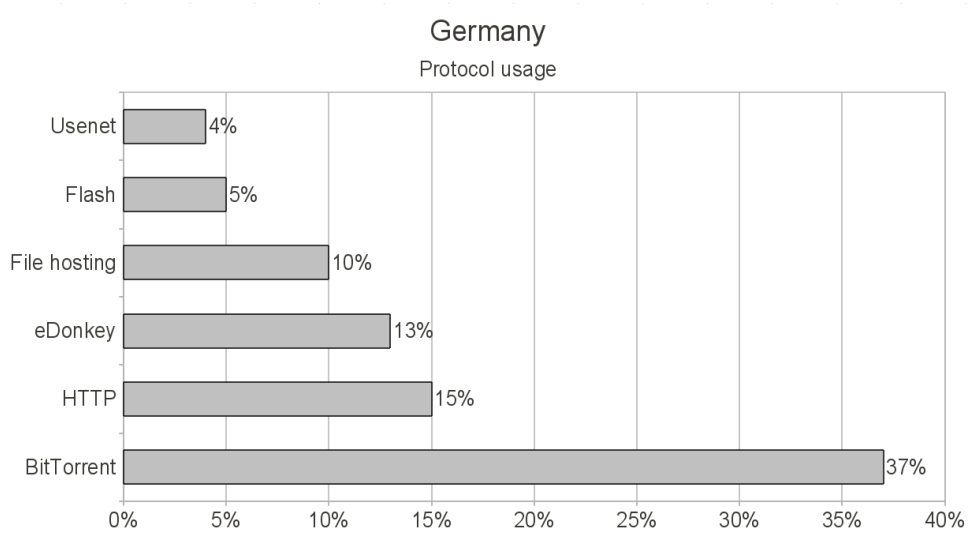


Figure 5.1: BitTorrent usage in 2008/2009 in Germany

5.4 Evaluation using 8 MB Files

5.4.1 Overview

Figure 5.2 shows an overview of all scenarios evaluated using one seeder. The amount of nodes is increasing and reflects regular nodes only. The BitTorrent tracker and the MCFTP file leader node are not included. The box shows the average value of the *download factor* as described in Section 5.2 for each scenario as well as the minimum and maximum values respectively. The file distributed has a size of 8.5 MB. We compared BitTorrent with the two MCFTP modes (Application Level Multicast and native IP Multicast) and with the two different approaches (cMCFTP and dMCFTP) each. BitTorrent has a considerable higher *download factor* compared to MCFTP-IPMC for the scenarios with 10 to 50 nodes, especially when the number of nodes is set to 30.

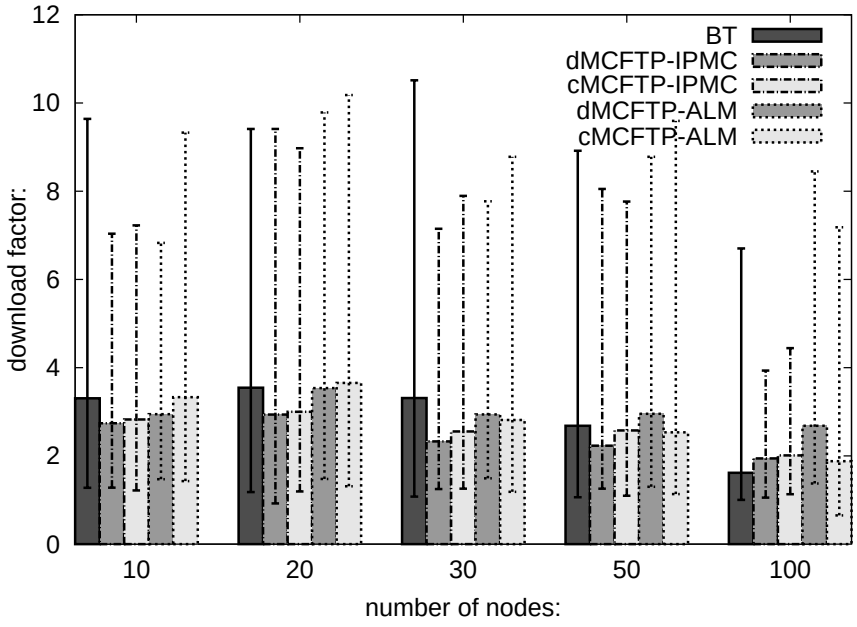
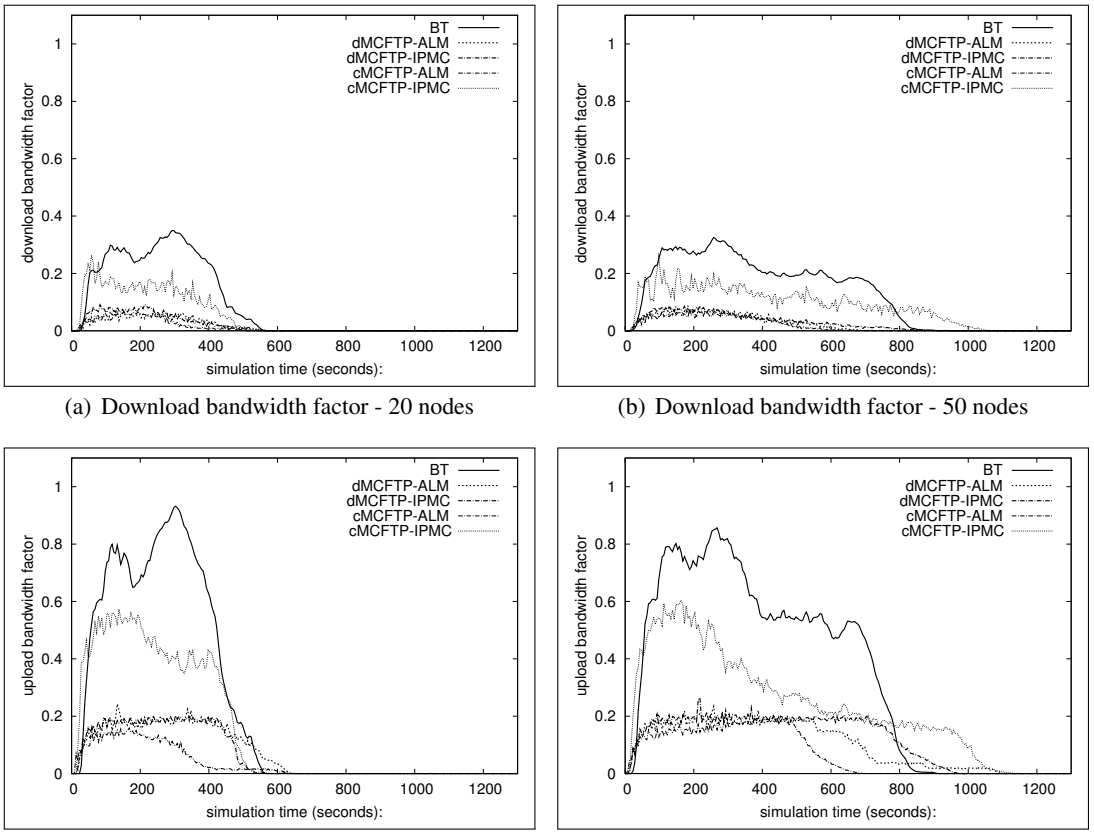


Figure 5.2: Overview of all scenarios with one seeder and a file size of 8MB

MCFTP-ALM is as well competitive to BitTorrent, whereas the difference is not always significant. BitTorrent has a clear advantage with the scenario with 100 nodes. This can be relativized though if we put in perspective that using BitTorrent, it was not always possible to limit the bandwidth. This was especially the case when having 100 nodes in the scenarios. MCFTP-IPMC has a lower *download factor* compared to MCFTP-ALM in both approaches. This is as expected since native IP Multicast is based on a more efficient technology than Application Level Multicast (See Chapter 2). It is interesting to see the different results for MCFTP-ALM and MCFTP-IPMC regarding the different approaches. In MCFTP-IPMC, the decentralized approach has in all scenarios a lower *download factor* compared to the centralized approach. This is not the case for MCFTP-ALM. The more nodes are in the MCFTP-ALM network, the better the centralized approach performs compared to the decentralized approach. For all protocols, there seems to be a tendency to a decreased *download factor* with increasing number of nodes. This is as expected since the more nodes are available, the more seeders will be in the swarm to serve leechers, which join the network later. Overall the results are mostly as expected. We can not clearly state if cMCFTP or dMCFTP is the better approach for data dissemination. We can see that dMCFTP-IPMC is the best approach for MCFTP and that we could still optimize the ALM approach of MCFTP. It would have been interesting to evaluate MCFTP-ALM with another ALM framework as well. We can state that in most evaluated scenarios MCFTP is the better approach to disseminate data than BitTorrent concerning the *download factor*. We implemented a prototype version of MCFTP from scratch in a few month and could proof concept that it is competitive with some state of the art BitTorrent client.

5.4.2 Results Discussion

Bytes transfered have been measured and compared to the available maximum bandwidth. Similar to the *download factor* we introduced bandwidth factors for the upload and for the download. These factors show how much of the available bandwidth is used for upload and download respectively. We compared the bandwidth factors over the whole runtime of a scenario.



(a) Download bandwidth factor - 20 nodes

(b) Download bandwidth factor - 50 nodes

(c) Upload bandwidth factor - 20 nodes

(d) Upload bandwidth factor - 50 nodes

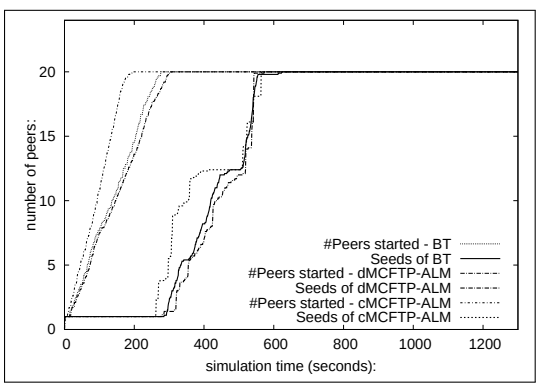
Figure 5.3: Down-/Upload bandwidth factors - 8 MB file size

Thus, every second we measured how many bytes were transfered. For example, if the maximum download bandwidth is 80 kBps and the current received data has a size of 60 kB, this would result in a download bandwidth factor(DBF) of:

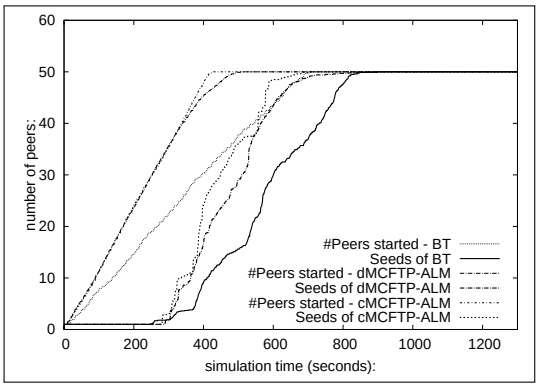
$$DBF = \frac{60kB}{80kBps} = 0.75$$

When all bandwidth is used, the corresponding bandwidth factor is 1 and if there is no bandwidth used at all, the bandwidth factor is 0. The bandwidth factors give an overview of the workload and efficiency of a protocol. The higher the factor is, the higher the workload is in the swarm. Figure 5.3 shows an overview of download and upload bandwidth factors for chosen scenarios.

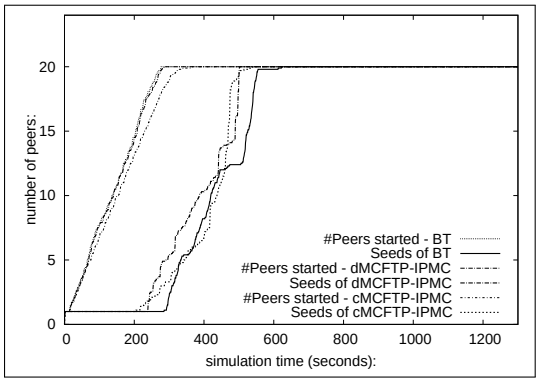
For the 8 MB file, we compared scenarios with 20 and 50 nodes with each having one seeder. For both scenarios, BitTorrent has a significant higher download and upload bandwidth factor compared to any MCFTP variant almost during the whole scenario runtime. This means, there was more data sent and received in total in the BitTorrent network than in any MCFTP swarm. As discussed in Fig. 5.2, all MCFTP variants have either a lower *download factor* or the *download factor* is not significantly worse compared to BitTorrent. Combining the two scenarios, we can conclude that all MCFTP variants can compete regarding *download factor* and additionally they are more efficient than BitTorrent regarding bandwidth factors. Hence, MCFTP can disseminate the same data in a shorter or similar time frame but uses significantly less bandwidth in the whole network. Not only the *download factor* of MCFTP is more efficient, also the *upload factor* is immensely better than at BitTorrent. In Fig. 5.3(a) and 5.3(c) we can see a drop for BitTorrent around second 200. This is about the same time, where the first leechers become seeders. New nodes have to be contacted and new connections have to be made at this time, which explains the drop.
Figure 5.4 shows how many seeders in a swarm develop over time and how many nodes joined the network.



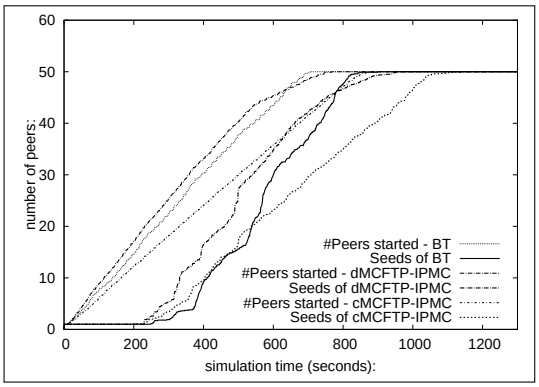
(a) Seeders over time - 20 nodes - MCFTP-ALM



(b) Seeders over time - 50 nodes - MCFTP-ALM



(c) Seeders over Time - 20 nodes - MCFTP-IPMC



(d) Seeders over Time - 50 nodes - MCFTP-IPMC

Figure 5.4: Seeders over time - 8 MB file size

Figures 5.4(a) and 5.4(b) show the amount of finished seeders of BitTorrent compared to MCFTP-ALM. Figures 5.4(c) and 5.4(d) compare seeder development of BitTorrent compared to MCFTP-IPMC. Since the evaluations were not done in an absolutely clean environment, it is crucial to always combine the seeder development with the total peers started/joined. For example on the first view, Fig. 5.4(d) indicates that the number of seeders for cMCFTP-IPMC did not increase as fast as the amount of seeders in BitTorrent. But if one considers the increase of the total number of peers in combination with the seeder development, one can see that in both protocols, the number of seeders is increasing almost in parallel to the number of total peers. Thus, the seeder development in the cMCFTP-IPMC protocol is not or only a little worse compared to the seeder development in BitTorrent. In all four scenarios there are edges in the development of seeders. The growth of seeders is not evenly increasing. For example, in Fig. 5.4(a) theres is an edge for all three protocols around second 500. This means, that there have been a few nodes becoming seeder at the same time. Therefore we can conclude, that there have been a few nodes waiting probably for the same chunk to finish. This is an indication that, the algorithm can be optimized to avoid such irregularities. It is interesting to see, that BitTorrent has the same behavior as MCFTP.

5.5 Seeder Check

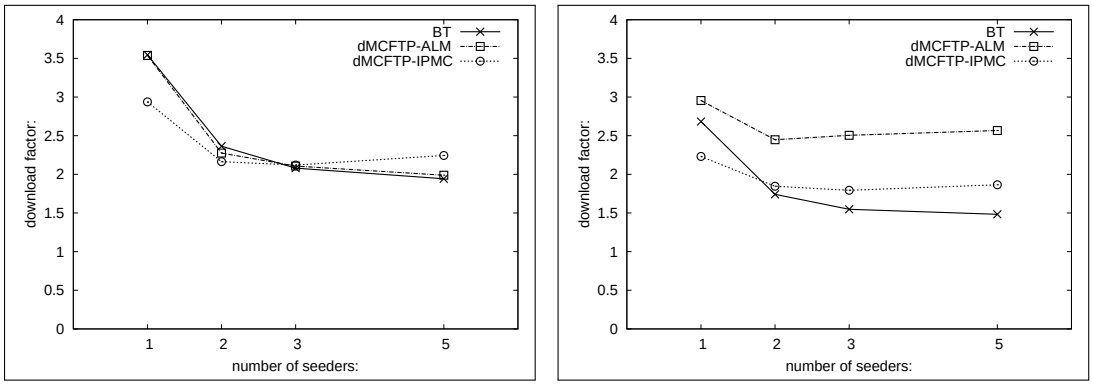
We defined scenarios with the same amount of nodes but with increasing amount of starting seeders as *seeder check* simulations. With the 8 MB file, we ran scenarios with increasing the number of seeders available from the beginning/start of a scenario. We focused on scenarios with 20 and 50 nodes including the seeders but excluding the BitTorrent tracker and the MCFTP file leader node. For the seeder check simulations, we did not use the centralized approach of MCFTP.

Results for 20 Nodes

Figure 5.5(a) shows the summary of the seeder check with a total of 20 nodes. MCFTP-IPMC has its lowest *download factor* with 3 seeders available from the beginning, whereas BitTorrent and MCFTP-ALM have their minimum with 5 seeders available from the beginning. The *download factor* for all three protocols are decreasing significantly from the scenario with one seeder compared to the scenario with two seeders. From the scenarios with 2 to 5 seeders, there is no significant difference between the protocols. The *download factors* are between the values of 2 and 2.5 for all scenarios with more than 1 seeder available from the beginning.

Results for 50 Nodes

The seeder check for the 8 MB file with 50 nodes is depicted in Fig. 5.5(b). The two different MCFTP modes are quite similar to each other regarding the dependency on the number of seeders available from the beginning. MCFTP-IPMC has throughout a lower *download factor* than MCFTP-ALM. The *download factor* for BitTorrent is decreasing continuously with additional seeders available from the beginning.



(a) Seeder check with 20 nodes

(b) Seeder check with 50 nodes

Figure 5.5: Seeder Check with 20 and 50 nodes using a file size of 8 MB

MCFTP-IPMC and BitTorrent scenarios with 50 nodes have lower *download factors* compared to scenarios with 20 nodes. Thus, the tendency described in Section 5.4 can be reconfirmed. The more nodes are in a network, the lower the *download factor*. This is also the case for scenarios with more than one seeder. When more nodes are available, more nodes will be sending different chunks in parallel and thus more leechers can download multiple chunks at the same time. The demands from leechers can better be fulfilled with increasing amount of nodes inside the MCFTP swarm or the BitTorrent swarm. The decentralized approach of MCFTP-ALM does not have such a low *download factor* as compared to BitTorrent nor MCFTP-IPMC. But as Fig. 5.2 shows, the centralized approach works better with MCFTP-ALM. And as mentioned before, there have been difficulties to limit bandwidth usage in BitTorrent scenarios. The issues to limit the bandwidth usage of BitTorrent are reflected in Fig. 5.5(b), where the *download factor* is decreasing to 1.5.

5.6 Evaluation using 50 MB Files

5.6.1 Overview

For the 50 MB file, we evaluated only the decentralized approach of MCFTP. An overview of the evaluated scenarios with one seeder is shown in Fig. 5.6. MCFTP-IPMC has a significant lower *download factor* compared to BitTorrent for all scenarios. The Application Level Multicast mode of MCFTP has a lower *download factor* than BitTorrent for scenarios with 10, 20 and 50 nodes. In the scenario with 30 nodes, BitTorrent has a slightly lower *download factor* compared to MCFTP-ALM. An interesting fact to point out is that the *download factors* for each protocol are comparable to the *download factors* for the 8 MB file respectively (see Fig. 5.2). It is interesting to see how good dMCFTP-IPMC is doing compared to BT with more than 20 nodes each. We expected that dMCFTP-IPMC would perform better than BitTorrent, but we did not expect it to be that clear. Especially not, since we had difficulties to limit the bandwidth usage of BitTorrent.

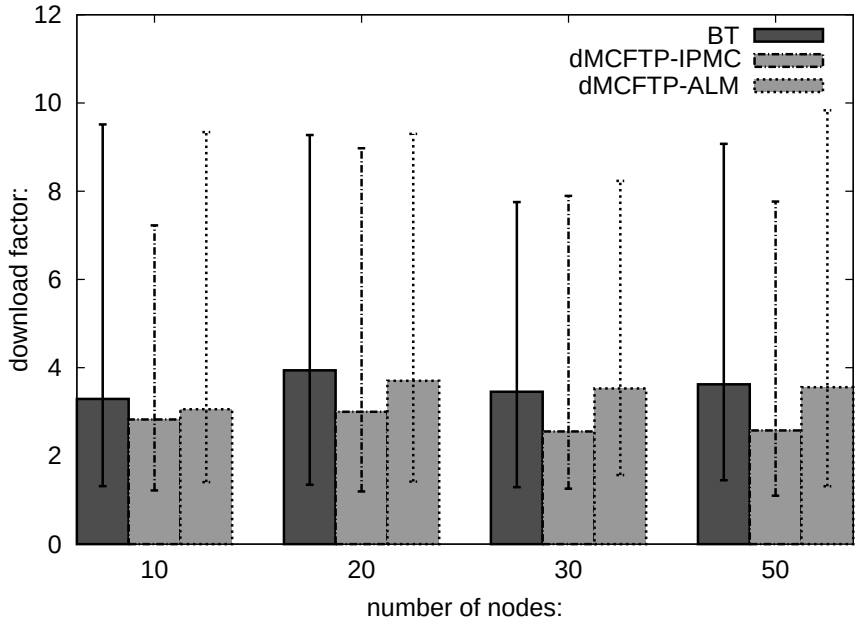
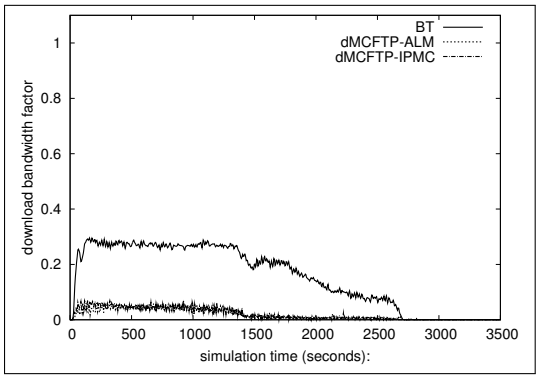


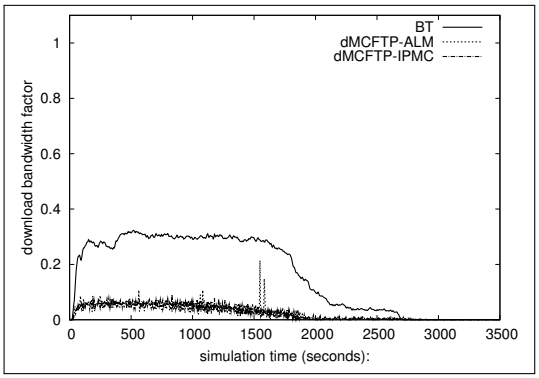
Figure 5.6: Overview of all scenarios with one seeder and a file size of 50 MB

5.6.2 Results Discussion

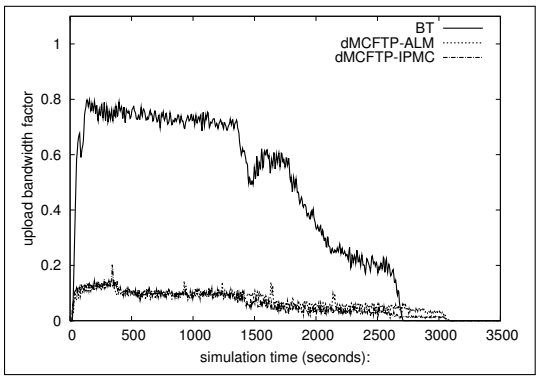
The download and upload factors for scenarios with the 50 MB file are shown in Fig. 5.7. The BitTorrent protocol is using significantly more bandwidth compared to both MCFTP modes. Whereas there is almost no difference between the MCFTP-ALM mode and the MCFTP-IPMC mode. The download and upload factors of all three protocols are comparable with the scenarios disseminating the 8 MB file as seen in Fig. 5.3. The *upload factor* of BitTorrent for scenarios with 50 nodes reaches 0.8 easily, whereas the MCFTP modes rarely reach the *upload factor* of 0.2. There is an amazingly high saving of resources when using MCFTP. There is a drop in Fig. 5.7(c) for the BitTorrent protocol around second 1400. This is the same time, where the first leechers become seeders. New nodes have to be contacted and new connections have to be made at this time, which explains the drop.
The seeder growth of all three protocols do not have any significant difference as depicted in Figures 5.8(a) and 5.8(b). The growth of seeder is stepwise. There is not an evenly increasing amount of seeders. This is an indication for having many nodes waiting to finish. This is possibly due to many nodes waiting for the same missing chunk. This means, that the algorithm to disseminate data should be reconsidered and optimized. The steps are a bit clearer in MCFTP than in BitTorrent. In MCFTP we could optimize the given strategies and try to mkae the growth of seeders linear.



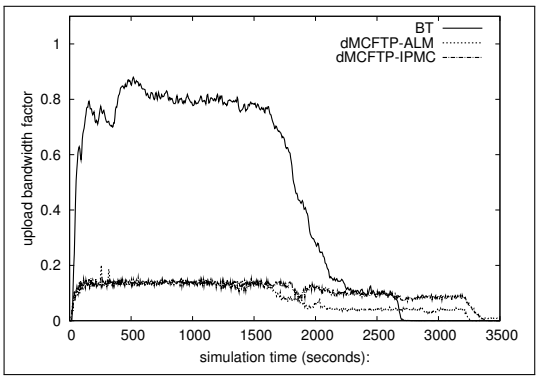
(a) Download bandwidth factor - 20 Nodes



(b) Download bandwidth factor - 50 Nodes

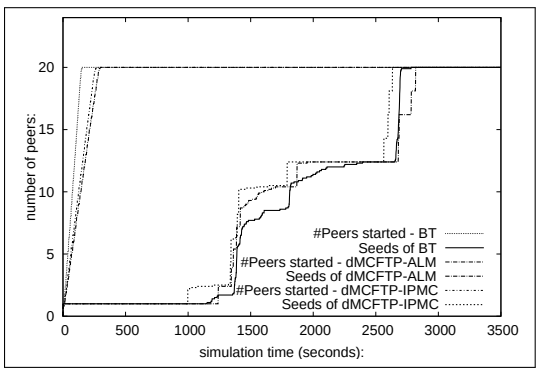


(c) Upload bandwidth factor - 20 Nodes

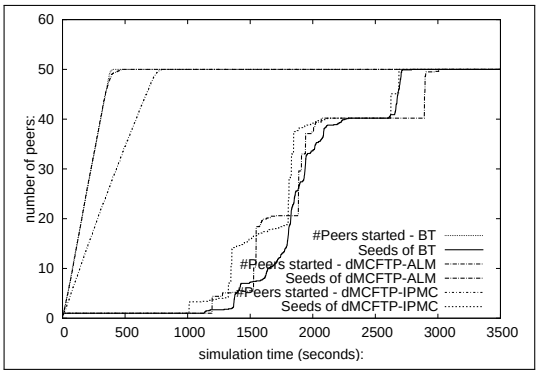


(d) Upload bandwidth factor - 50 Nodes

Figure 5.7: Down-/Upload bandwidth factors - 50 MB file size



(a) Seeders over Time - 20 Nodes



(b) Seeders over Time - 50 Nodes

Figure 5.8: Seeders over time - 50 MB file size

5.7 Simulation Results

In [3], they used the network simulator ns-2 for different MCFTP scenario simulations. They compared MCFTP with BitTorrent in flash-crowd scenarios. They could get precise values from the clean ns-2 environment. In their work, they also implemented a very basic BitTorrent simulator. In [3] they state: "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." One can not truly compare the results of this real world implementation with the results of the MCFTP simulator in [3], because the scenario setups are different and the implementation details are different as well. In this evaluations we used a fully featured BitTorrent client with at least 7 years of work and optimizations behind it. The simulated BitTorrent client was implemented by themselves and was very basic. Also we had difficulties to limit the bandwidth usage by our BitTorrent client.

5.8 Summary and Conclusion

In this Chapter, we presented the evaluations for various scenarios using MCFTP-IPMC and MCFTP-ALM, both with the centralized and decentralized approach as well as scenarios with BitTorrent. The evaluations were performed for file sizes of 8 MB and 50 MB. As a conclusion, we can state that MCFTP can compete to BitTorrent and is even better in various scenarios. BitTorrent is much more resource intensive compared to MCFTP. We can state that the results of our evaluations and the simulations done in [3] show that MCFTP is better in most of the scenarios and that it can still be optimized in various settings.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

We implemented and evaluated the Multicast File Transfer Protocol (MCFTP). The implementation supports native IP Multicast as well as Application Level Multicast (ALM). We characterized and implemented two different MCFTP approaches(cMCFTP and dMCFTP). We compared MCFTP with a well known BitTorrent client on a real world environment.

The evaluated scenarios with one seeder and increasing amount of leechers show that BitTorrent is much more resource intensive than MCFTP. BitTorrent is using up to four times more resources than MCFTP. This is the most significant difference between BitTorrent and MCFTP concerning the performance. Not only is MCFTP more resource saving, it is also in most variations significantly faster to disseminate data to all peers in the swarm. Albeit this MCFTP implementation is not fully optimized, our evaluations have shown that it is truly competitive to state of the art dissemination mechanisms, such as BitTorrent.
Not only the MCFTP-IPMC mode scored a lower *download factor* compared to BitTorrent, but also the MCFTP-ALM scored lower *download factors* in various scenarios. MCFTP-IPMC came out with better results compared to MCFTP-ALM which was as expected since the IP Multicast mechanism is more efficient than ALM. MCFTP-IPMC scored a lot better than expected and should be considered as first choice when using MCFTP.
Different scenarios show, that dMCFTP performs generally better than cMCFTP. Especially when using native IP Multicast. Surprisingly, cMCFTP scored better than dMCFTP in Application Level Multicast. But, dMCFTP is as not topic to a single point of failure and therefore to be chosen when MCFTP is used.
Evaluations on seeder growth over time have shown that there are possibly a few nodes waiting for the same chunk, which would indicate optimization potential. There is as well optimization potential for MCFTP when multiple initial seeders are used. Although the results are still competitive with BitTorrent for various scenarios, a new strategy taking in account multiple initial seeders could be implemented.
Different scenarios show that the overall *download factor* is decreasing with increasing amount of nodes. Overall, there were no significant differences when the file size changed for most of the scenarios.

6.2 Outlook

The most important part of our implementation is the possibility to adapt dissemination strategies. Our evaluations have shown, that there is a great potential to optimize MCFTP by adapting the different strategies. The strategies used for the final evaluations as presented in this thesis are not yet fully optimized. As a next step, multiple strategies could be introduced and evaluated to obtain even better results for the Multicast File Transfer Protocol implementation. The change of a strategy is done modular and therefor it should not be a difficult task to replace strategies. Even the support for meta strategies is implemented, but not evaluated properly yet. Additionally to the strategies, there might be better solutions for different approaches. The centralized and decentralized approaches were only basic ways to introduce two different nodes. There could be approaches, which split the MCFTP swarm into clusters to gain a reduction of sent control messages which might have a negative impact on the *download factors*.
The current implementation was used as proof of concept and is therefore not user friendly. A graphical user interface and support for many files at the same time would increase the value of the user experience with MCFTP dramatically. The introduced hand over of a file leader node in the centralized MCFTP approach is not implemented and could be used to avoid single point of failure situations. For future evaluations, a cleaner environment could be advised and many evaluation runs should be considered to guarantee a clearer outcome. Another step for future work could be to test MCFTP-ALM with many Pastry rings, one for each chunk multicast group. To do so, an environment is needed which has many high performance computers in order to be able to generate this amount of Pastry rings, which is very resource intensive. Another interesting evaluation could be to use different ALM frameworks, such as CAN [26], Chord[27] or Tapestry [28]. The size of MCFTP messages could be further decreased by dividing messages on bit level. As an example, we used one byte (256 variations) for the message type, which would actually easily fit into two bits (4 variations), since we do not have more than three different message types. Even data message could be further decreased in size. Out of the minimizing message, more data throughput could be achieved. One major issue encountered during our evaluations was the need to limit the bandwidth usage of BitTorrent consequently. For future work, a better limitation would be advised.

Glossary

Notation	Description
ALM	Application Level Multicast
API	Application Programming Interface
B	Bytes
Bps	Bytes per second
BT	BitTorrent
cMCFTP	Centralized Multicast File Transfer Protocol
DBF	Download Bandwidth Factor
DF	Download Factor
DHT	Distributed Hash Table
dMCFTP	Decentralized Multicast File Transfer Protocol
FMG	File Management Group
FTP	File Transfer Protocol
GB	Gigabyte
GHz	Gigahertz
hex	hexadecimal
HTTP	Hypertext Transfer Protocol
ID	Identification
IM	Instant Message
IP	Internet Protocol
IPv4	Internet Protocol version 4
ISP	Internet Service Provider
JDK	Java Development Kit
kBps	Kilo Bytes per second

Notation	Description
MB	Megabyte
MCFTP	Multicast File Transfer Protocol
MCFTP-ALM	Multicast File Transfer Protocol with Application Level Multicast mode
MCFTP-IPMC	Multicast File Transfer Protocol with native IP Multicast mode
ODT	Optimal Download Time
P2P	Peer-to-Peer
RAM	Random-Access Memory
RTT	Round Trip Time
SHA1	Secure Hash Algorithm-1
URL	Uniform Resource Locator
VoIP	Voice over IP

Bibliography

[1] Zattoo Webpage. [Online]. Available: http://www.zattoo.com

[2] wilmaa Webpage. [Online]. Available: http://www.wilmaa.com

[3] D. Papritz, M. Brogle, and T. Braun, "MCFTP(Multicast File Transfer Protocol): Simulation and comparison with BitTorrent," 2010.

[4] S. E. Deering, "RFC 1112: Host extensions for IP multicasting," 1989.

[5] W. Fenner, "RFC 2236: Internet Group Management Protocol, Version 2," 1997.

[6] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan, "RFC 3376: Internet Group Management Protocol, Version 3," 2002.

[7] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, pp. 72–93, 2005.

[8] R. Schollmeier, " A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications," in *Proceedings of the First International Conference on Peer-to-Peer Computing*, ser. P2P '01, 2001.

[9] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon'03)*, 2003.

[10] BitTorrent Webpage. [Online]. Available: http://www.bittorrent.org

[11] *Secure Hash Standard.* National Institute of Standards and Technology, 2008, federal Information Processing Standard 180-3.

[12] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Proceedings IPTPS*, 2002.

[13] C. P. Fry and M. K. Reiter, "Really truly trackerless bittorrent," Tech. Rep., 2006.

[14] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," 2001.

[15] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas, "A survey of application-layer multicast protocols," *IEEE Commun. Surveys and Tutorials*, 2007.

[16] S. Fahmy and M. Kwon, "Characterizing overlay multicast networks and their costs," *IEEE/ACM Trans. Netw.*, 2007.

[17] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth, "The Essence of P2P: A Reference Architecture for Overlay Networks," in *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[18] R. S. Ryan, R. Braud, and B. Bhattacharjee, "Slurpie: A Cooperative Bulk Data Transfer Protocol," in *Proceedings of IEEE INFOCOM*, 2004.

[19] J. Postel, R. Comments, and J. Reynolds, "RFC 959: File Transfer Protocol (ftp)," 1985.

[20] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 20, p. 2002, 2002.

[21] Sun Java Webpage. [Online]. Available: http://java.sun.com

[22] Freepastry Webpage. [Online]. Available: http://www.freepastry.org

[23] K. M. Hendrik Schulze, "ipoque. Internet Study 2008/2009," 2009. [Online]. Available: http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009

[24] Azureus Webpage. [Online]. Available: http://azureus.sourceforge.net

[25] Trickle Webpage. [Online]. Available: http://monkey.org/~marius/pages/?page=trickle

[26] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-Level Multicast Using Content-Addressable Networks," 2001.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," 2001.

[28] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph, B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep., 2001.