Communication & Distributed Systems

RWTH AACHEN UNIVERSITY

A P2P Shared Storage System based on XMPP

Bachelor Thesis

Nicolas Inden

RWTH Aachen University, Germany
Chair of Communication and Distributed Systems

Advisors:

Dipl.-Inform.	Elias Weingärtner
Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr.	Bernhard Rumpe

Registration date:	2012-03-09
Submission date:	2012-07-09

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, July 15, 2012

Abstract

This thesis proposes an approach of a P2P shared storage system based on XMPP called Peergroup. The ambition of Peergroup is to deliver functionality for sharing data with a group of users allowing each user to work with this data while files and changes are distributed using P2P data transfer. To reduce traffic we only redistribute changed parts of a file. We use XMPP as a reliable protocol to manage users and handle signaling data exchange. We further use Apache Thrift to provide a clean and extendable interface for P2P data transfer between users. Peergroup thus works decentralized storing the data only at the peers participating in the network. Using XMPP Multi-User-Chat to distribute signaling data provides a facility to clearly determine the users who we share data with. We do in-depth tests of the system, evaluating file distribution times with up to 80 nodes that show how it scales with a large number of users, concluding that Peergroup is already feasible for the use in networks with up to 40 nodes.

Acknowledgments

I am really happy that I found a place at ComSys to write this thesis in the best imaginable atmosphere. It was a great time for me working at “the chair”. A big “Thank you!” goes to all people at ComSys, especially to Elias Weingärtner who did a great job supervising me during the development process of my thesis. I would also like to thank Prof. Dr.-Ing. Wehrle for accepting my thesis and giving me the chance to scientifically investigate and evaluate my project “Peergroup”. Further gratitude goes to Prof. Dr. Rumpe for being second examiner. I am very grateful for the possibility to use the nodes of the German-Lab which empowered me to evaluate “Peergroup” in larger scale. Therefore I thank the University of Würzburg for providing me with access to the German-Lab. I finally want to thank my family and my girlfriend Michaela for their sympathy and support. You helped me a lot!

Contents

1 Introduction **1**

2 Background **3**

2.1 Peer-to-Peer Networks 3

2.1.1 General Properties 3

2.1.2 BitTorrent 4

2.2 eXtensible Message and Presence Protocol 5

2.3 Lamport Clocks 7

3 Software Design **9**

3.1 Event-Based Programming 9

3.2 Inter-Client Communication 10

3.2.1 Signaling Data 11

3.2.2 Userdata: Methods improving P2P performance 12

3.2.2.1 Dividing files into filechunks 12

3.2.2.2 Chunk distribution strategies 14

4 Implementation **17**

4.1 Software Architecture 17

4.1.1 Objects 18

4.1.2 Threads 19

4.2 Event processing 20

4.3 Using XMPP 23

4.4 Peer-to-Peer file transfers 24

5 Evaluation **29**

5.1 The German-Lab 29

5.2 Measurements 31

6 Related Work **37**

6.1 Distributed File Systems 37

6.2 Cloud-based storage systems 38

6.3 PAST 38

6.4 File sharing protocols 39

7 Conclusion **41**

Bibliography **43**

1

Introduction

With the increasing speed of consumer internet-lines more and more systems evolve providing online data storage. The feasibility of such services increases as access to larger files is more convenient with faster connections. However, most consumer storage services base on a centralized approach storing data in data centers raising questions concerning data security and trustworthiness [19]. We propose "Peergroup" a tool to share storage in a decentralized fashion across multiple users and devices.

A typical use-case for online storage systems is the synchronization of data between a user's devices. Often users need access to certain data at home and at work, or on the go. Using Peergroup we can synchronize these devices without involving a third party that is in possession of the data. This rises advantages concerning sensible data, as the data is never located outside your own devices. Though we need the peer with updated data (device at home) to be online to transfer updates to a non updated peer (device at work), it is possible to gain a persistent accessibility of updated data by adding a peer to the network that is permanently online. By adding multiple of such peers we can even create redundancy for data availability.

A second use-case deals with the sharing of data among multiple users. Centralized storage services empower a user to share certain files with other users. This allows them to work on the same set of data. This is especially useful if a group of people works on a project and every member of the group needs access to the recent version of a file. Peergroup can synchronize project folders across the project members storing data at the members' devices and distributing only changes of modified files.

We are furthermore able to use Peergroup for the distribution of content and applications. With only one Peergroup client announcing new or changed files of a shared folder we can exemplarily distribute applications or other content to customers running a download-only version of Peergroup.

To encounter these scenarios, we propose a peer-to-peer (P2P) shared storage system allowing users to share data directly without using centralized storage mechanisms. Files are distributed blockwise in order to benefit from multiple peers uploading

contents of the same file. We further handle changes in files by only redistributing the corresponding blocks. The result is a system being able to synchronize folder contents across multiple peers. We can use Peergroup to maintain shared documents with friends, synchronize own data across multiple devices or distribute content to a certain set of users.

Synchronizing data in a decentralized fashion raises certain challenges. We need a facility responsible for signaling data which firstly organizes the users/peers in the network, and secondly manages the current set of shared files. A second facility responsible for userdata processing is needed to perform the direct data exchange between the peers. In order to keep Peergroup decentralized, all management regarding current files and users is done by each peer for itself. This is made possible by directly propagating changes happening at a specific peer to the other peers. We use the eXtensible Message and Presence Protocol (XMPP)[1], which makes peers easily addressable, to communicate file metadata and changes to participating peers. As soon as changes are recognized by a peer, the changed blocks are transferred using P2P connections.

In this thesis we introduce Peergroup, explain its functionality by introducing the used techniques and evaluate it by showing how far the current version of Peergroup scales with the number of peers. We finally interpret the results and propose possibilities for the improvement and enhancement of Peergroup.

[1] For general information about XMPP see: http://xmpp.org/about-xmpp/

2

Background

This chapter introduces the general techniques that are used by Peergroup. We will give an overview over different kinds of P2P networks, their advantages and disadvantages, and have a special focus on BitTorrent and its properties of splitting up data into blocks to optimize data distribution. Further, an introduction of XMPP is given, which is used by Peergroup to exchange signaling data. At last we will describe versioning systems with a special focus on Lamport Clocks which preserve the global order of signaling events.

2.1 Peer-to-Peer Networks

P2P is one of the two big paradigms of networking [31]. It describes the way data is requested and distributed in a network. Accurately, it says that data is directly transferred between peers without intermediate nodes in the overlay network. For a proper understanding of the different P2P approaches we need to distinguish between *userdata* and *overhead data*. *Userdata* is the actual data that a user requests from or provides to the network, whereas *overhead data*, also called *signaling data*, is necessary information to manage the peers and data in the network.

2.1.1 General Properties

P2P networks have certain advantages over networks following the client-server paradigm having a central server maintaining the data. The two most obvious are the lack of a single point of failure and scalability. P2P networks deliver decentralized storage of data and, depending on the used approach, decentralized peer and route management, meaning that the load for overhead processings is spread among the nodes of the network. In theory this enables us to provide endless scalability as no node will ever reach its computational limits.

Current approaches of P2P-Systems like Chord [32], Pastry [25] and Gnutella [3] can be divided into *structured* P2P-Systems (Chord, Pastry) and *unstructured* P2P-Systems (Gnutella, eDonkey). The basic difference between structured and unstructured systems is their handling of data location management. Where unstructured systems use flooding to request a specific piece of data among the peers, structured systems maintain a distributed hash table (DHT). This DHT is generally a piece of routing information telling a node either which other node owns the requested data or which of the known nodes is logically nearer to the requested data and is aware of the owner. Structured approaches are always self organizing networks. The joining or leaving of a peer in the network renders a reconstruction or correction of the DHT on logically near peers necessary.

Unstructured P2P-Systems can be realized in *pure, hybrid* or *centralized* fashion. [31]

Pure P2P-Systems consist of equal peers. All organization of data location and peer availability is done by the peers themselves. The flooding overhead is heavy and must be limited, raising the possibility that actually existent data may not be found.

Hybrid P2P-Systems have higher leveled nodes responsible for a basic infrastructure. The flooding used in unstructured approaches is limited to the infrastructure nodes limiting the overhead.

Centralized P2P-Systems rely on a central facility to maintain information of data location and peer availability. The scalability of this approach hardly depends on the hardware performance of the central facility. However, centralized P2P-Systems do not rely on flooding.

Peer-to-Peer networks deliver a practical way to exchange big amounts of data between many peers, without needing a central server with a lot of storage and uplink capacity. Additionally, P2P networks with signaling data management enable the user to exactly know at which peers the data is stored. This property is especially important to Peergroup as the user should always know who is in possession of his data.

2.1.2 BitTorrent

BitTorrent [16] is a file sharing protocol for static content distribution using P2P networking. Key properties of BitTorrent are the .torrent file and the tracker. For each content that should be shared via BitTorrent, the initial seeder must create a .torrent file. This file contains block hashes of the content and the addresses of the tracker that manages the swarm of this torrent. The .torrent file can now be distributed among the peers we want to share the content with. Once a peer starts the download of content, it looks up the tracker stated in the .torrent file and connects to it. The tracker replies with the IPs of other peers currently downloading or seeding the content of the same .torrent file. The peer can now connect to the other peers and download the blocks of the content.

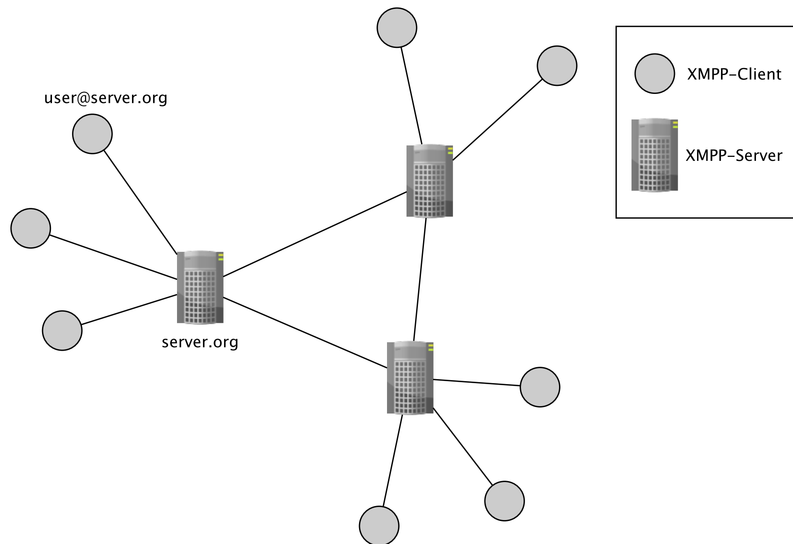


Figure 2.1 Abstract XMPP infrastructure showing connections between clients and servers

BitTorrent uses multiple methods to make the distribution of the content efficient. First, content gets split up into blocks of variable size to quickly be able to seed completed blocks to other peers. Furthermore, BitTorrent uses different strategies of acquiring specific blocks of a file depending on the distribution level of this block and the download progress of the file. In detail, BitTorrent follows a rarest block first strategy, downloading those blocks at first being most rare among the peers. This minimizes the probability that blocks are not available because the only peer having them disconnected. [16]

This principle of data distribution has proven of value, but has the drawback that only static content can be exchanged with other peers. It is not possible to do changes to the shared content and immediately spread them to the other peers. It appears feasible to use techniques of BitTorrent to realize the distribution of non-static content via Peergroup.

2.2 eXtensible Message and Presence Protocol

The eXtensible Message and Presence Protocol (XMPP) [13], also known as *Jabber*, is an XML based multi-purpose-protocol providing functionality for distributed real-time communication over an eMail-like architecture. Additionaly, XMPP is free, meaning that it is not controlled or owned by a company. Everyone is free to extend the protocol by his own means, and everyone can setup his own server. Similar to eMail a XMPP-Network consists of multiple XMPP-Servers where users can register, see Figure 2.1. The resulting accounts (JIDs) are in the *username@server* scheme followed by a */resource* representing the device used to connect to the network, see Figure 2.2. Messages sent between two users are routed over the corresponding servers where the users are registered.

The most common use cases for XMPP are chatting and presence checking. A user can add other users to his roster, a friend list where contacts can be sorted into

```
    User    : Bob
    Server  : jabber.ccc.de
    Resource: Laptop
    -> JID: Bob@jabber.ccc.de/Laptop
```

Figure 2.2 Jabber Identifier (JID)

groups. For all contacts in the roster the user gets information about their presence like *online, away, offline* or some custom set presence status. Further, users can chat with each other either by contacting a specific user via his JID, or by choosing a contact from the roster. Google and Facebook use XMPP as a basis for their services *Google Talk*[4] and *Facebook Chat*[2].

The data sent between clients and servers is structured as XML. A connection between a client and a server is called a stream and thus is started with **<stream>** and ended with **</stream>**. All structures sent between these stream-tags are called stanzas and are simplified formed as follows:

```
<_stanzatype_ id=_someID_ from=_sender_ to=_receiver_>
    <body>My chat message</body>
    <property name=_somename_ value=_somevalue_ />
    <property name=_somename_ value=_somevalue_ />
    ...
</_stanzatype_>
```

Figure 2.3 Simplified XMPP-Stanza

XMPP defines three different XML Stanzas, the *<message/>*, *<presence/>* and *<iq/>* stanza[15].

Message-Stanzas provide functionality to communicate with the peers stated in the "to" field of the stanza. We can set a chat message (between <body/> tags) as well as properties.

Presence-Stanzas serve to distribute information about the users presence status. Next to the predefined presence options, the user can set a custom presence here.

IQ-Stanzas are Information/Query stanzas being used to request arbitrary information from the server or other clients.

As proposed in [15] the definitions of XMPP-Stanzas allow arbitrary enhancements within the limits of XML. Due to the general definition of stanzas and the body-tag being optional, XMPP-Stanzas can be used to exchange arbitrary data between users. This property of XMPP-Stanzas is used by Peergroup to encapsulate signaling data to be exchanged between peers.

XMPP is extended by many XMPP Extension Protocols (XEPs). As the standalone XMPP specification has no support for chat rooms, XEP-0045 defines the Multi-User-Chat (MUC)[26]. The MUC defines a room hosted on an arbitrary XMPP-Server. Rooms have similar identifiers as users; they usually have the form *roomname@conference.server.tld*. This identifier serves as the receiver in a message stanza

sent to the room. The server will then forward this message to all users connected to this room.

MUC rooms have several facilities for user management. Besides the association of a password to the room, MUCs provide functionality to treat users as admins, moderators or users. XMPP calls these roles *affiliations*. So, depending on the affiliation of a user, different rights can be granted.

Peergroup uses MUCs to determine sets of users that want to share data. As rooms can freely be created and secured, we can easily set what data is to be shared with which users.

2.3 Lamport Clocks

In distributed systems events need to be synchronized in order to keep a consistent view on a shared set of data. This means that we need to determine a total ordering of all events happening in the distributed system. Lamport Clocks provide an easy solution for this problem by introducing the $\rightarrow$ (*happened-before*) relation, which helps us to determine a partial ordering of events. The $\rightarrow$ relation is illustrated by the following rules from [20]:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Practically exerting Lamport Clocks is using logical clocks attached to each message sent within a distributed system. Each node has its own logical clock starting with a value of i. Each time the node n sends out an event e to other nodes, this event is attached with a clock of value i or in other words, the event happens at $C_n(e) = i$. Nodes receiving this event compare the clock of this event with their local clock and set their local clock to $max(C_n(e), local_clock) + 1$. Thus, their next outgoing event will have a higher clock value than the last event they received. However, as this approach only delivers a partial ordering, it still leaves the possibility that events on different nodes get attached with the same value for their logical clocks. We can illustrate this with an example:

- Node A, B and C have an initial clock of $C_A = C_B = C_C = 1$.
- Node A sends a message m to B and C timestamped with $C_A(m) = 1$.
- Both B and C receive m and set their clocks to $max(C_A(m) = 1, C_B = C_C = 1) + 1 = 2$
- Both B and C answer the message m at the same time by attaching their answers n_B and n_C a logical clock of $C_B(n_B) = C_C(n_C) = 2$

- Node A now receives two answer messages having the same logical timestamp.

In the given case, node A has no information on the succession in which to process the messages from B and C. So, constructing a consistent total ordering of events is crucial to a distributed systems, as applying the same events in different order may lead to different states of the system.

To transform the partial ordering created by the logical clocks into a total ordering, we need an additional treatment for messages delivered with the same timestamp. As proposed in [20], we can use any arbitrary total ordering $\prec$ of the nodes participating in our system, to handle messages with the same logical timestamp. A simple example for this would be using the node ID as a total ordering for messages having the same timestamp.

After applying Lamport Clocks to all events communicated between peers, these events can be put in a globally consistent total order. This especially affects situations in which multiple users simultaneously work on the same set of data in a distributed system.

3

Software Design

The software design of Peergroup can be divided into two main parts: The handling of occurring events, and the inter-client communication. Events are either messages from the file system or incoming messages from the network, they are crucial regarding the behavior of the client as all client actions are triggered by events. Thus, Peergroup is an event-based system. The important part concerning performance is the inter-client communication. It defines the way data is exchanged and thus has the biggest impact on time and resource requirements for the data exchange.

3.1 Event-Based Programming

Peergroup belongs to the class of event-based systems as they are proposed in [23]. Due to the handling of file-system originated and network originated events, the application flow is completely determined by external events and thus is a chain of cause and reaction. The only user-caused influences are the parameters the user sets when starting the program, and the file-system events he causes by changing the content of the shared folder.

Event-based programming enables us to clearly define which states the program passes through. Using a finite state machine we always know in what state the program currently is, and which state will be the next, depending on the next event that has to be processed. The default and initial state is a waiting state. In this state the program waits for events to be processed. For each possible event we then define states the program visits when processing the event. After the computations for an event have finished, the program will return to the waiting state, being ready to process the next event.

Looking at an event-based system we see that it generally consists of two parts that compose its full functionality:

1. Receiving a new event and categorizing it

2. Handling categorized events appropriately

Receiving a new event happens by monitoring a specific input of a special category of events. An example for such an input is the API of a file-system driver. Linux provides iNotify[5] to monitor events occuring at the file-system. The events received here can be categorized as file-system events and be processed accordingly.

In the case of Peergroup, categorizing events is done by distinguishing between the different sources where the events are monitored. This results in two main categories of events in Peergroup:

1. File-system events
2. Network events

File-system events inform Peergroup about any changes that occurred to the shared folder, allowing Peergroup to maintain a list of files and directories to be shared.

Network events serve the purpose of communicating locally occurring changes to the other Peergroup clients. This makes the other clients able to react to these changes.

However, the distinction between these two categories does not suffice to achieve a proper handling of the events. After determining the category of an event, we must handle the event with regard to the information it delivers. In fact we have to know what kinds of events each category can include in order to implement a proper handling.

As most conditions that affect the application flow of Peergroup are (external) events and we only have a small fraction of tasks to be processed linearly, using event-based programming seems feasible to be used by Peergroup.

3.2 Inter-Client Communication

Peergroup's primary task is to maintain a consistent view on the same set of data over multiple participants. To fulfil this task, Peergroup causes two different kinds of data traffic. First, a Peergroup client needs to maintain a common list of files together with other Peergroup clients. The management data exchanged for this purpose contains messages about created, modified and deleted files among other information and is called *signaling data*. Signaling data always follows a one-to-many relation with regard to senders and receivers, as the sender of a signaling message always delivers this message to all other clients. Second, a Peergroup client needs to exchange the eventual file content with other Peergroup clients. File content data or *user data* is the biggest part of the traffic caused by a Peergroup client.

Inter-Client communication has the biggest influence on the performance of Peergroup. We therefore use a BitTorrent-like technique of transferring user data. BitTorrent has proven of value and provides an efficient way of distributing data[16]. By distributing signaling data using the XMPP protocol there is no need for a tracker like they are used in classic BitTorrent applications.

In this section we will have a deeper look at how Peergroup works with special regard on how signaling data and user data are exchanged between clients. We will explain in detail how Peergroup uses XMPP to exchange signaling data. After discussing the advantages and disadvantages of XMPP in this use-case, we will focus on BitTorrent-like methods which improve the exchange of user data like dividing data into blocks and block requesting strategies.

3.2.1 Signaling Data

Signaling data causes all network related actions and reactions of the Peergroup client. Data from other clients signaling for instance that a new file was created remotely, or that a file was changed remotely, is received and handled accordingly. To maintain a globally consistent list of files, signaling data has to be communicated to all other peers participating in Peergroup. Each peer receives all changes that are made to the shared folder and thereby is able to apply these changes to its local list. In order to maintain the consistency of the file lists located at the peers it is important to ensure that signaling messages arrive without errors and with least latency possible at each peer.

An existing system to efficiently distribute messages and organize users is the eMail system. Users register for a mail address at their preferred mail server to get a mail address consisting of their username and the servername. This leads to a determined identifiability with regard to senders and receivers of messages. Looking at an abstract eMail infrastructure we can see that eMail servers are equi-valued nodes between whom messages are passed. The sender is the entry point at one of the servers, and the receiver is the leaving point of a message from the network. Regarding message delivery eMail has proven its value, so it is desirable to use a system based on a similar infrastructure for Peergroup to distribute signaling data. However, eMail is designed as a delay tolerant network and by lacking real-time support does not completely meet the requirements of Peergroup.

A very similar protocol with regard to message routing is XMPP proposed in [13]. In contrast to eMail, XMPP is a real-time protocol and provides functionality to distribute signaling data to a large amount of clients. In its standard coverage, XMPP does not support creating chat channels where messages can be sent to all participating users. However, this feature is proposed in [26] and was accepted by the XSF as an official XEP. Peergroup makes heavy use of the Multi-User-Chat in order to clearly determine the group of people to share data with and to distribute signaling data. We call this channel *Activity-Stream*, or *AS*.

Assume Alice wants to share data with her friends Bob and Charly. In this case they need to communicate to each other:

- Files being created
- Files being modified
- Files being deleted

To limit the set of receivers to Bob and Charly, Alice creates an XMPP Multi-User-Chat channel and externally communicates the name and password of this channel to Bob and Charly. Once Bob and Charly are logged into the channel they can receive the signaling data Alice addressed to the channel. By processing the signaling data they are able to maintain a consistent list of files they share in their directory.

Alice	(Alice)1:45.23 – NewFile(name,size,noOfBlocks,hash)
Bob	(ChatUser1)1:45.29 – Hi there!
Charly	(Bob)1:45.33 – ModifiedFile(name,size,affectedBlocks,hash)
ChatUser1	(ChatUser2)1:45.57 – Hey!
ChatUser2	(Charly)1:52.41 – DeleteFile(name,hash)

Figure 3.1 Activity-Stream

Each action communicated in the AS delivers several information about the concerning file object. As illustrated in Figure 3.1 we always need to know the filename and the filehash to head the action to the correct object. For new and modified files we also provide the size (after the update) and a list of information about new/updated blocks. This is only a fraction of the delivered information, for a complete list see chapter 4.

3.2.2 Userdata: Methods improving P2P performance

The P2P paradigm applies to data transfers happening directly between two or more peers. However, regarding this description even directly copying a file from one node to another in a LAN would be a P2P transfer. We assume that one node is the owner of a set of data that the other node requests. Both nodes have a fixed and limited up- and download bandwidth. This does not seem to be problematic concerning two nodes as a simple copy is the fastest possibility to get data from one node to the other. Expanding this example to n nodes arises the problem that each downloading node only gets $\frac{1}{n-1}$ of the upload rate. Finally, the uploading node distributes the same content $n-1$ times with its limited upload bandwidth. We see that simply copying files from one node to multiple other nodes is not efficient. As this is a very common scenario using Peergroup, we need to find a more efficient way of distributing data among multiple nodes.

The BitTorrent [16] file sharing protocol proposes multiple techniques for speeding up data distribution among a high number of nodes by utilizing every node's upload capacity to serve other nodes[16]. In this section we introduce these techniques and explain how Peergroup's file transfer mechanisms are inspired by BitTorrent in order to speed up data distribution.

3.2.2.1 Dividing files into filechunks

Obviously a node in a network can only distribute data that it is completely in possession of, meaning that this data has to be fully available on its local storage.

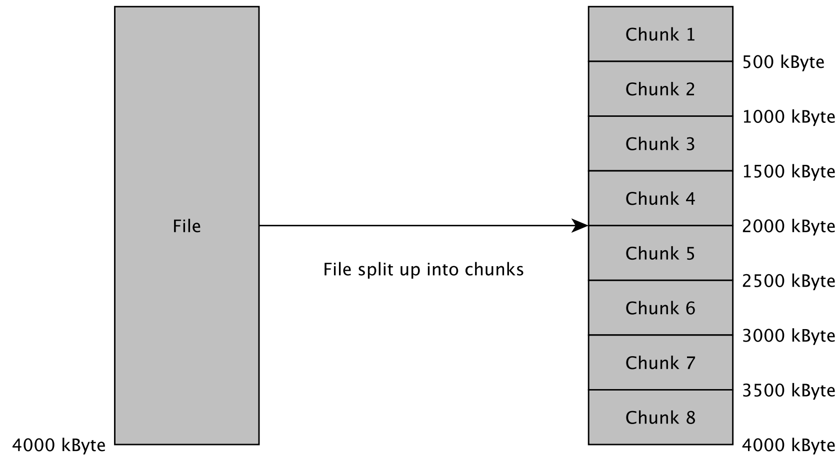


Figure 3.2 Dividing up a 4000kByte file into chunks of 500 kBytes

From the mentioned division of upload capacity and redundant data delivery we can conclude that the full transfer of a file can take very long depending on the filesize and the amount of nodes downloading this file from the initial uploader. This time can be shortened by dividing the file into multiple logical chunks as illustrated in Figure 3.2. We therefore define an amount of bytes after which a new logical chunk of a file starts. As soon as the initial uploader uploaded a complete chunk to another node, this node is able to instantly upload this chunk to other nodes still needing this chunk. This has two effects improving the performance of the file distribution:

1. The cumulated upload capacity concerning the completely uploaded chunk increases.

2. The initial uploader has much less redundantly[1] uploaded data and can use the free upload capacity to distribute chunks that have not been distributed yet.

The first effect is obvious. The initial uploader anbandons the exclusive responsibility of distributing this chunk to the peer(s) that have completely downloaded it, relieving itself. The second effect is a result of the first, as the initial uploader can focus its upload capacity on uploading chunks that have not been completely uploaded yet.

Generally a user has files of all sizes on his computer and thus is probably willing to share files of all sizes from small textfiles to big databases. To provide optimal distribution performance we need to choose the chunksize used in Peergroup wisely. Both too small chunks and too large chunks will provide worse results than correctly sized chunks. Using undersized chunks will cause too much overhead because the client will often have to request new chunks whereas using oversized chunks will delay the time of completing a copy of a chunk and hence decrease the time in which we could benefit from other peers also uploading data. Further, we have to consider the available network link to find a suitable chunksize. Faster connections allow us to use bigger sizes as they need less time to distribute the chunks. Using undersized

[1]Uploading the same piece of data to multiple nodes.

sizes on fast connections will decrease overall performance due to frequent chunk requests and each request causes a short delay in the download progress of a file.

The optimal chunksize depends on the size of the file we want to distribute and on the network link capacities we have. The filesize is clearly determined so we could calculate a chunksize depending on the filesize[16]. However, the network link is variable and we could at best assume a value near the average of all links used in a network if we are in possession of this information. Actually the file size is the only reliable variable we could use to calculate a suitable chunksize if we have no information about the used network links.

Peergroup differs from BitTorrent clients as it is interactive in the way that data can be changed at any time. This includes that filesizes are also not fixed anymore. By editing files, they can grow or shrink heavily so that they would need a different chunksize. Adjusting the chunksize of an edited file leads to a complete recalculation of all chunks and finally to the need of completely redistributing the file as all chunks get different hashes. For this reason Peergroup uses a fixed chunksize. However, we will experimentally determine the influence of specific chunksizes with regard to download speeds in the evaluation section.

We can conclude that treating files as multiple chunks empowers us to use the available upload capacity of many nodes better compared to treating files as one big part. The chunksize used by Peergroup is experimentally determined and thus a good choice in most situations. It is up to the user to choose a different chunksize, in this case every user sharing files with him should choose the same chunksize in order to avoid frequent redistributions.

3.2.2.2 Chunk distribution strategies

Once files are treated as multiple chunks we can use different strategies to distribute these chunks. These strategies can aim at producing a logical copy among all peers in the least time possible by downloading chunks with low availability, or at providing the fastest download rates for the downloading peer by choosing chunks with high availability. We will discuss what advantages and disadvantages these strategies have, and which strategy is appropriate for the use by Peergroup.

Though chunk distribution strategy indicates an active mechanism of the uploader, the strategy is in fact chosen by the downloading peer. The downloading peer has a list of all files and chunks available in the network and decides depending on the strategy which chunk is to be downloaded next.

Once a new file is added to the shared folder or a file is changed we are facing the situation that only one peer is in possession of the new file or the updated content. So it is desirable to use a chunk distribution strategy that quickly produces one logical copy of the new file spread among all peers. This way we maintain the availability of the new file for the case that the initial uploader leaves the network.

One approach requesting chunks is to download them sequentially. We start downloading the first chunk of a file and we finish with the last chunk. A second approach is requesting chunks randomly. Both approaches ignore the availability of chunks and therby potentially lead to unavailable chunks if nodes having rare chunks leave the network.

A simple, yet effective strategy to avoid the unavailability of new data is *Rarest-Chunk-First* as proposed in [16]. Following this strategy a client looks up which chunk has the lowest availability in the network among the chunks it still needs to download. Requesting and downloading this chunk increases the availability of the chunk and also the probability that the chunk is still available if a client leaves the network.

However, the *Rarest-Chunk-First* strategy is not optimal in all cases and needs some modifications. Assume peer A introduces a new file f. A is the only peer being in possession of the data belonging to the new file f. As a result, all chunks of f have an availability of 1. Strictly following the *Rarest-Chunk-First* strategy will lead all other peers to download the first chunk of f, then the second, and so on. This behavior is undesirable because it results in the same problem as downloading the file as a whole. The initial uploader will have to serve all other peers, and there will be no exchange between the other peers. One possibility to solve this problem is to include randomness. In particular this means that a peer not necessarily starts the download of a file with the first chunk, but a random one. Using a random order of downloading chunks will much faster create a logical copy spread among all peers, so that the peers can share all missing blocks among one another. Regardless of network link speeds, the optimal case would be that the amount of peers equals the amount of chunks, and each peer has exactly one chunk, that no other peer has, and no other chunks. In this case we have exactly one logical copy of the file and can create our own complete copy by downloading each chunk from a different peer. A random order in downloading chunks will not cause this optimal case but it will increase the benefit of being able to download from multiple peers heavily.

4

Implementation

Peergroup is written in Java to provide functionality on the most used end-user operating systems. We use threaded programming in order to be able to react quickly on external events without blocking the flow of the main application. In the days of multi-core CPUs it seems feasible to use threaded programming to solve this kind of tasks. We further make use of the Smack API[1] to realize XMPP communication, and we use Apache Thrift[2] to have a uniform interface for P2P transfers between Peergroup clients. After giving a short introduction into Peergroups software architecture, we explain in this chapter how events are used to coordinate the application flow, we will discuss how Lamport Clocks are utilized to maintain a global ordering of messages in the network and finally we will focus on the Smack API and Apache Thrift explaining how Peergroup's communication is realized.

4.1 Software Architecture

This section gives a quick overview over the general construction of Peergroup. Starting with the classes we shortly introduce the responsibilities of each of them and their interplay. After explaining the used threads, we further explain how intermediate data like the current file list and node affiliations concerning single blocks are created and cached.

Peergroup consists of several classes defining special objects being used by Peergroup. These objects are either own objects used by Peergroup or objects created by Thrift which serve the case that instances of these objects are transferred from or to another peer. We further have classes representing the threads we use.

[1]Documentation and Source available at: http://www.igniterealtime.org/projects/smack/
[2]Documentation and Source available at: http://thrift.apache.org/

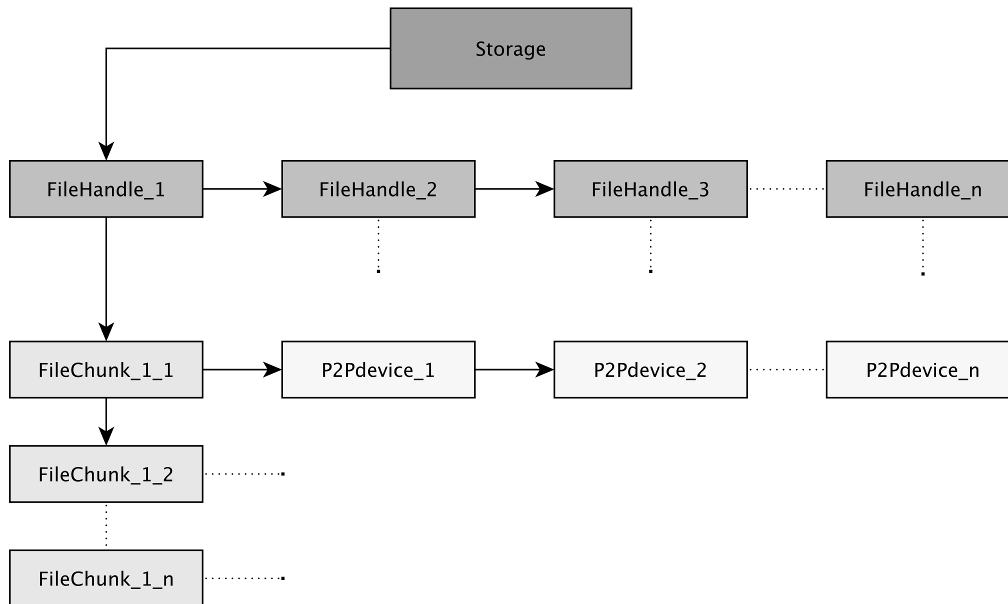


Figure 4.1 Storage datastructure used for maintaining shared files

4.1.1 Objects

In order to provide its functionality, Peergroup internally has different objects handling occurring data. Here we will give a short list and explanation of the most important objects used by Peergroup:

The Storage object is a singleton[3] and contains information about the location of the shared directory. Furthermore, it includes a linked list of *FileHandle*s representing the files shared in the network. At last the storage has a version number which is increased each time if a change happens to the file list.

The FileHandle contains information about a single file. The most important attributes are the *name*, the *hash*, the *version* and a linked list of *FileChunk*s which this file is composed of.

The FileChunk represents a piece of binary data of fixed size. All FileChunks of a FileHandle together compose the original file. A FileChunk most importantly has an *ID*, a *hash*, a *version*, an *offset* and a linked list of *P2Pdevice*s. If a file is completely downloaded and up to date, the version of all FileChunks match the version of the corresponding FileHandle.

The P2Pdevice object contains information about a peer in the network. Next to the JID of this peer it includes the IP address and the port where the peer waits for incoming connections.

The objects described above together result in the data-structure shown in Figure 4.1.

[3]Design pattern that avoids creating more than one instance of a class

4.1.2 Threads

We use threads in order to react to events either caused internally or externally. It is important that events are quickly recognized and processed so we chose using multiple threads with each having a blocking listener for a specific category of events. A thread waiting for an event therefore blocks for the time that no event occurs which is fitting to this thread. Once such an event happens the thread continues processing the event by extracting relevant information from it. The thread then creates a request object containing these information which is then enqueued in a main queue where requests from all events awaiting threads are collected.

Because we are using threads we need a facility to store commonly used variables. To serve this purpose we introduce an own class called `Variables`. This class contains no functions but all variables that should be shared among the threads. All variables are declared *public* and hence are editable by all threads as soon as they are not set and declared *final*.

It is important to manage a thread safe usage of shared variables. We know that concurrent access to shared variables can lead to faulty reads of a variable if this variable is changed at the same time. Further, threads caching variables could read wrong values if the variable was changed by another thread. Especially commonly used lists and queues are prone to faults due to concurrent access of threads. Java can avoid such faults by synchronizing concurrent accesses to shared variables. Variables that are known to be concurrently accessed need to be declared with the tag *volatile*. Once these variables are accessed, Java cares for locking and unlocking this variable in order to provide consistent reads and writes. Further, each access is given a happens-before relation concerning following reads of the same variable [6]. Java also avoids caching of variables and redirects all changes directly to main memory.

In particular Peergroup has six classes defining the necessary threads used in run-time:

The MainWorker is responsible for sorting and processing the requests collected in the main queue.

The NetworkWorker listens for messages sent in the activity-stream and enqueues corresponding requests in the main queue.

The StorageWorker listens for file system activity enqueueing `ENTRY_CREATED` and `ENTRY_DELETED` events in the main queue, and `ENTRY_MODIFIED` events in an intermediate queue which is processed by the ModifyQueueWorker.

The ModifyQueueWorker processes the `ENTRY_MODIFIED` events found in the intermediate queue. The purpose of this thread is to collect fast occurring subsequent `ENTRY_MODIFIED` events of a specific file and merge them to a single `ENTRY_MODIFIED` request for the main queue as soon as the corresponding file has not changed for two seconds.

The ThriftServerWorker is the base thread where the ThreadPoolServer of thrift is run. Depending on the demand for chunks, this thread creates further threads serving other peers' download requests.

The ThriftClientWorker is the base thread for establishing connections to other peers and downloading file chunks. Depending on the maximum size of the ThreadPool, this class creates up to this amount of threads performing the eventual download.

4.2 Event processing

In Chapter 3 we discussed that Peergroup belongs to the class of event-based systems. In this section we present where events are taken from and how they are processed in detail. We therefore describe how we use threads and queues to recognize events, enqueue them, and finally process them in the main thread.

Peergroup has two main sources where it monitors the relevant events that have to be processed. The first source is the file-system which delivers information about any changes made to the shared directory. On one side these information are absolutely necessary to maintain a list of files and directories in our shared folder, on the other side we need to filter the information coming from the file-system with regard to the original cause of this information. This means that events reported by the file-system can either be caused by the user changing files in the shared folder, or they can be caused by Peergroup itself while downloading filechunks from other peers which have to be written to the corresponding file in the shared folder. Both causes an identical event reported by the file-system, but the latter has to be ignored as it does not represent a local change that is actually done to the file by the user. We call these *Ninja-Events*. This situation is illustrated in Figure 4.2.

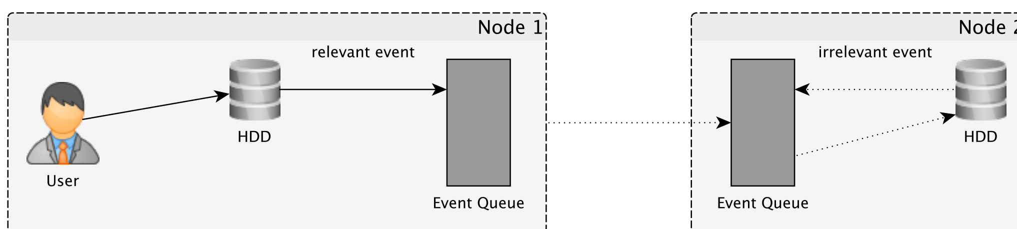


Figure 4.2 Reportable and non-reportable events

We can see that file-system events originating from local changes caused by the local user are of relevance and thus need to be reported to other clients in order to publish the change (Node 1). Further, we need to ignore file-system events originated from the local storing of data that is downloaded from other peers in order to avoid to republish an already published set of data as new data (Node 2).

The second source of events is the network. Events from the network are delivered as XMPP messages and are mostly resulting from local file-system events. We send out an event to the network if we have locally done changes to the shared folder. The kind of change together with some meta data about the change then is composed into a XMPP message and communicated to the other peers as an event. Network events firstly serve to communicate local file-system events to the other peers for their file-list maintenance and secondly act as overhead information about the downloading process. So on one side we have the same events as received from the

```
public void run(){
// Initialize variables

    while(!isInterrupted()){
        // Get the next event from the network
        // Blocks as long as no message is available
        Message next = Network.getNextMessage();

        // Handle message appropriately
        switch(next.getType()){
            case ENTRY_CREATED:
                Constants.requestQueue.offer(
                new Request(ENTRY_CREATED,next));
            case ENTRY_MODIFIED:
            ...
        }
    }
}
```

Figure 4.3 Scheme of a thread waiting for events

file-system and on the other side events informing other peers about what chunks are completely available. Additionally we have two network events responsible for the synchronization of the file-list. New peers joining a network are typically not knowing what files are currently in the shared directory. Thus, on joining the activity-stream they send an event requesting the file-list versions from the other peers. The other peers reply to this event by sending their file-list version. Afterwards the newly joined peer grabs the current file-list from the peer with the highest file-list version, and from a random peer if all other peers have the recent version. For the list of events occurring in Peergroup see table 4.1.

File-System Events	Network Events
ENTRY_CREATED	ENTRY_CREATED
ENTRY_MODIFIED	ENTRY_MODIFIED
ENTRY_DELETED	ENTRY_DELETED
	COMPLETED_FILECHUNK
	REQUEST_CURRENT_FILELIST
	SEND_MY_FILELIST_REVISION

Table 4.1 Table of events used by Peergroup

Peergroup has two threads responsible for catching occurring events, the `StorageWorker` and the `NetworkWorker`. Both threads basically consist of a loop running for the time the thread is not interrupted. An exemplary scheme of such a thread is shown in Figure 4.3.

In the case of the scheme we wait for the next message to be received from the network. After determining the type of the message, we create a request object which is then enqueued into the main request queue. The request queue is a point of concentration for all requests that arise from events caught in the event threads.

Peergroup uses a request queue where all events are collected and enqueued in order to be processed. In the main thread of Peergroup this queue is permanently pro-

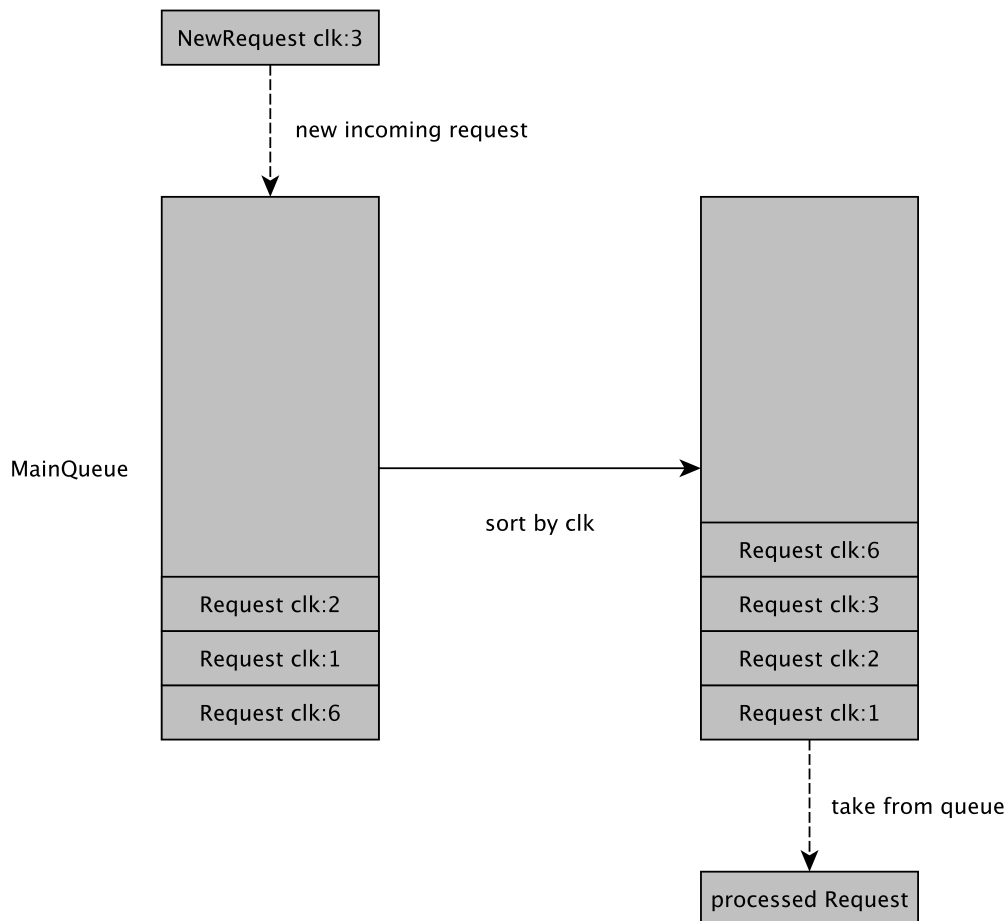


Figure 4.4 Main queue with lamport sorting

cessed. The thread waits until requests are available in the queue and then processes them.

One purpose of collecting all request in a central queue is being able to sort the requests. Especially requests originating from network events need to be processed in the correct order to maintain a consistent directory content. Events received from the network have a Lamport Clock value attached to them, this value is also attached to the corresponding request enqueued in the main queue. We are now able to sort the requests with regard to their attached clock value, and thus apply changes at all peers in the same order.

Figure 4.4 illustrates Peergroup's principle of enqueueing, sorting and processing requests. At first requests are added to the main queue. At this time we have two cases in which it is possible that the requests found in the queue are not in the correct order:

1. We previously just received requests, but processed none.
2. We previously processed requests, but the new request is not in the correct place.

We sort the requests in the queue when taking the next request from the queue. So if no requests have been taken from the queue so far we have never sorted it resulting in

a potentially wrong order of requests. For easy maintainability we use a LinkedList as queue. Further, we cannot assume that requests are always added to the queue in correct order as the events do not necessarily arrive at the client in the order according their Lamport Clocks. So adding a new request to a previously sorted queue does not necessarily result in a correctly sorted queue again. This however is mitigated as the queue is always resorted when taking the next request from the queue. This practice assumes that if we process a request with a Lamport Clock value of c, we will not receive a new event with a Lamport Clock valued lower than c. In this case we cannot ensure processing requests in the globally correct order. Forcing to process requests in globally correct order requires a lot more signaling data [33]. Each time a client wants to take the next request from its queue it has to "ask" all other clients if their next locally enqueued request has reached the same or a higher value. If this is not the case we will receive new events from these clients having lower Lamport Clock values than the request we are about to process locally. If globally all lower valued requests have been processed, we can be sure not to receive an event lower valued than the one we are about to process locally.

4.3 Using XMPP

We use the eXtensible Message and Presence Protocol to deliver signaling data between Peergroup clients. XMPP is a well documented [13] [14], free and open protocol. To build upon a distributed architecture of multiple servers XMPP seems feasible in order to deliver signaling data. In contrast to single server solutions it is possible to avoid the problem of a single point of failure by implementing a fallback mechanism continuing signaling data delivery over an Activity-Stream hosted on a different XMPP server. Looking at a current list[7] of XMPP servers we see that there are more than enough choices to find alternative servers. The only requirement for the server is to allow *message*-stanzas without <*body*>-elements. In comparison to normal chat behavior, a Peergroup client sends more messages in shorter time periods, it is therefore recommended that the server allows frequent message deliveries, and does not block clients sending many messages in short time.

To have access to the vast majority of XMPP functionalities we use the Smack API [8] providing a simple interface to most XMPP functions. Smack is a Java implementation of the XMPP protocol stack. The access to XMPP functions is delivered by the use of the implemented classes of Smack. Embedding XMPP functionalities in Java classes with easy methods for many use-cases saves us from packet level coding.

XMPP provides a variety of manipulating standard messages without having to code on packet level. As discussed we need to modify messages sent over XMPP in order to hide these signaling data messages from regular chat clients. We therefore have to know that the actual chat message displayed by a chat client is embedded in the <*body*> tags of an XMPP message. By simply using XMPP messages without <*body*> tags and thus without chat messages, we have created a XMPP message construct that is delivered by XMPP servers but not displayed by regular chat clients.

The XMPP message is a variable construct allowing us not only to omit chat content, but also to embed other arbitrary data. For this case Smack provides functionality

to set *properties* on a message stanza. *Properties* can be seen like variables being attached to the message. Smack supports Integer, Long, Float, Double, Boolean, String-, and Java-Objects to be attached to a message [9]. We use this functionality to deliver signaling data from one peer to the remaining peers.

Here we show an example of how to create an "ENTRY_DELETED" message in Java and how the real XMPP Stanza looks like:

```
MultiUserChat muc = new MultiUserChat(getConn(),roomAndServer);
Message newMessage = muc.createMessage();
newMessage.setType(Message.Type.groupchat);

// set properties
newMessage.setProperty("LamportTime",10);
newMessage.setProperty("Type",2);
newMessage.setProperty("JID",
                       "node57@jabber.server.tld/peergroup");
newMessage.setProperty("name","test.file");

// send message
muc.sendMessage(newMessage);
```

Figure 4.5 Creating a Smack *Message* object containing signaling data

In Figure 4.7 we see the resulting XMPP message as it would be received by the other peers in the activity stream. Each property we set for the message is listed in the XMPP stanza with a <*name*> and <*value*> tag containing the respective values. The parsing of this information is done by Smack, so we can easily extract the properties when receiving such a message. A short example of extracting the integer value determining the message type is:

```
int type = ((Integer)newMessage.getProperty("Type")).intValue();
```

Figure 4.6 Extracting an integer value from a property attached to a *Message* object

We see that Smack provides a convenient way of delivering XMPP messages. The messages are easily changeable and extendable in case of the need for more functionality.

4.4 Peer-to-Peer file transfers

Peer-to-peer transfers have some problems to take into account on the implementation level. The first problem is the way most people are connected to the internet. In many cases end-users have a DSL or cable internet connection which they locally share using routers. That makes it possible to use one internet connection with multiple devices at home. The disadvantage for peer-to-peer communication is the Network-Address-Translation (NAT) performed by the router in combination with a firewall blocking all ports of incoming direction. The second problem is the definition of what data is sent between peers and how this data is structured. It is desirable to have easy interfaces for peer-to-peer data transfer to simplify extendability and the interplay with other programs that want to access our system.

```
<message from="channel@conference.server.tld/node57"
         type="groupchat"
         id="N0bG8-9"
         to="node@jabber.myserver.tld/res" >
  <properties
  xmlns="http://www.jivesoftware.com/xmlns/xmpp/properties">
    <property>
        <name>Type</name>
        <value type="integer" >2</value>
    </property>
    <property>
        <name>JID</name>
        <value type="string" >
        node57@jabber.server.tld/peergroup
        </value>
    </property>
    <property>
        <name>name</name>
        <value type="string" >test.file</value>
    </property>
    <property>
        <name>LamportTime</name>
        <value type="long" >10</value>
    </property>
  </properties>
</message>
```

Figure 4.7 Creating a Smack *Message* object containing signaling data

It is in the nature of peer-to-peer connections that one peer needs to be able to establish a connection to the other peer. A connection is typically established by one peer listening for a connection on a defined port, and the other peer connecting to this port of the first peer. However, often peers are not connected directly to the internet but through a NAT performing router. The router is a gateway between the local network and the internet. Most internet service providers (ISP) only assign one IP address to each user so the router's task is to replace the local IP address in the headers of outgoing packets with the external IP address assigned by the ISP and vice versa. As now all devices in the local net are represented by the same IP on the internet side, we also need to tell the router if a local device is listening for external connections on a specific port, so that the router can forward connection attempts on this port of the external IP to the local device. This is necessary as most routers by default block any connection attempts done to any port. Incoming data is by default only accepted if a connection was previously actively established to this peer.

A feasible feature to automatically enable port forwarding at the router side is Universal Plug and Play (UPnP). UPnP proposes a standard way of informing the router to open a specific port and redirect connection attempts on this port to the corresponding local device[24]. The major drawback of UPnP is that the router must support UPnP and the UPnP support must be activated. As UPnP can also be considered as a security vulnerability some users might have this option deactivated in the router firmware. In this case the port has to be forwarded manually in the router firmware.

Defining clear interfaces for transferring data between two nodes is important in order to keep related code easily maintainable and extendable. A utility for creating such interfaces is the cross-language software framework Thrift [30]. After defining what data is to be exchanged in a thrift definition file, Thrift translates this definition file into Java source code that can directly be used by Peergroup. The classes created by Thrift contain all methods being declared in the definition file.

A thrift definition file basically consists of *structs* and *services*. Thrift structs are similar to structs as they are used in C. They are a collection of variables together forming a new type which can be again part of other structs.

```
struct ThriftFileChunk {
    1: i32 chunkID,
    2: i32 blockVersion,
    3: i32 size,
    4: string hash,
    5: list<ThriftP2PDevice> devices
}
```

Figure 4.8 Definition of a struct in Thrift

As shown in Figure 4.8 we define the `ThriftFileChunk` as a collection of *chunkID, blockVersion, size, hash* and *devices*. We see that standard variables as integers (i32) and strings are used and a list of type `ThriftP2PDevice`. The prefix *list* defines a list of variables of the same type. This is similar to the use of LinkedLists in Java.

```
service DataTransfer {
    ThriftStorage getStorage(),
    binary getDataBlock(1:string filename,
                        2:i32 blockID,
                        3:string hash)
}
```

Figure 4.9 Definition of a service in Thrift

Figure 4.9 shows us how *services* are defined within thrift definition files. In this case we define Peergroup's service *DataTransfer* which is responsible to define the methods invokable by the client to get the file list or a data block from another peer. This service consists of two invokable methods "ThriftStorage getStorage()" and "binary getDataBlock(3)". The first one only returns an object of type `ThriftStorage` containing all information of a node's current file-list. The second one requests the data of a file chunk specified by the three delivered variables *filename, blockID* and *hash*. The return type *binary* applies to a *ByteBuffer* object in Java.

Establishing a successful Thrift connection requires one peer to have a running thrift server set up and the other peer connecting to the other peer's Thrift server with a specified protocol. As Peergroup clients should be able to serve more than one peer, we use the `TThreadPoolServer` object provided by thrift. Using a thread pool has the advantage that we can exactly define how much connections are to be served at maximum by limiting the maximal amount of threads in this pool.

Depending on the data that has to be transferred Thrift offers different classes of type `TProtocol`. With Peergroup primarily exchanging binary data via Thrift, the

most feasible protocol to use is the `TBinaryProtocol` [10]. This protocol converts all types to be submitted into binary format. The binary result is then sent preceded by a length value that makes proper parsing possible. A string for example is converted to a byte array and then sent. Important for Peergroup is that binary data available in a *ByteBuffer* object does not require any more processing by thrift which causes the lowest possible load on the peer [10].

5

Evaluation

We described a system used to synchronize data over multiple peers using XMPP for signaling data exchange and a peer-to-peer connection for user data exchange. As such systems can be used in a variety of environments ranging from sharing data in rather small friend-networks to distributing data in large productive environments e.g. program updates, it is of interest how this system performs.

The performance of a distributed system can obviously be defined by multiple aspects. As Peergroup primarily deals with data distribution, the following aspects appear to be a feasible measure for its performance:

File distribution time is a timespan starting at the moment a peer announces a new file in the Activity-Stream, and ending as soon as the last peer finished downloading the file.

Degree of bandwidth utilization measures how much the available bandwidth of the network link is utilized. Are there peers fully utilizing their (upload) bandwidth while other peers at the same time only perform little uploads?

Scalability across the number of nodes means that the deviation in distribution times of the same amount of data across different amounts of peers should optimally be as small as possible in a network of peers with equi-valued network links.

5.1 The German-Lab

In order to gather meaningful results we need to perform our tests in a test facility providing enough peers. The German-Lab (gLab)[1] is a union of several universities located in Germany. Each university provides a certain amount of peers to the gLab

[1] Homepage with closer information: http://www.german-lab.de/

and in return gains access to all available peers. For the tests we performed with Peergroup we had 80 peers available.

The gLab peers are running Linux and grant access via SSH. The peers do not feature a firewall and are directly accessible from the internet. However, due to the different location of the peers we have different kinds of network links. Mostly, peers from the same location are connected very well among one another, whereas the connection to peers of other locations varies. To give an overview over the network links between the gLab peers we used for testing Peergroup, we performed RTT and Iperf[2]-throughput measurements.

As the inter-location communication has the biggest impact on the results (we will see that peers in the same location are well connected to each other), we will cluster the peers according to their location. We have four locations where the 80 test nodes are spread among:

1. Technische Universität Darmstadt (**DS**, 18 peers)
2. Karlsruhe Institute of Technology (**KIT**, 9 peers)
3. University of Kaiserslautern (**UKL**, 31 peers)
4. Technical University of Munich (**TUM**, 22 peers)

In order to show how the gLab performs, meaning how well nodes are connected to one another with regard to delays and throughput, we tested the average Round Trip Time (RTT) and throughput within the different locations of the nodes. Figure 5.1 shows the average RTT between the nodes of each location. The data is gathered by performing five pings between each pair of nodes. The *min*, *average*, *max* and *deviation* of these five pings are stored. Afterwards we take all results from nodes of location A and calculate the average of the stored average values concerning the nodes from location B. The diagram additionally includes the *standard deviation* for each measurement.

Figure 5.1 shows that peers from the same location usually have low delays among one another, whereas inter-location delays are higher. The standard deviations of the measurements show that the measured values only have few deviations from their mean in average and thus the mean is a good representative for the delays measured.

In Figure 5.2 we tested the average throughput capacity between the nodes of different and equal locations. Similar to the delay tests one bar (A-B) in the diagram is created by calculating the average of all measurements originating from A and destining at B. We additionally include the *standard error* here to show the accuracy of the calculated means.

We generally see that intra-location throughputs in e) are considerably higher than inter-location throughputs shown in a)-d), which is an expected result. However, also inter-location measurements show fluctuations between 10 Mbit/s and 120 Mbit/s.

[2] Iperf is a tool for measuring maximum TCP and UDP bandwidth performance, see: http://sourceforge.net/projects/iperf/

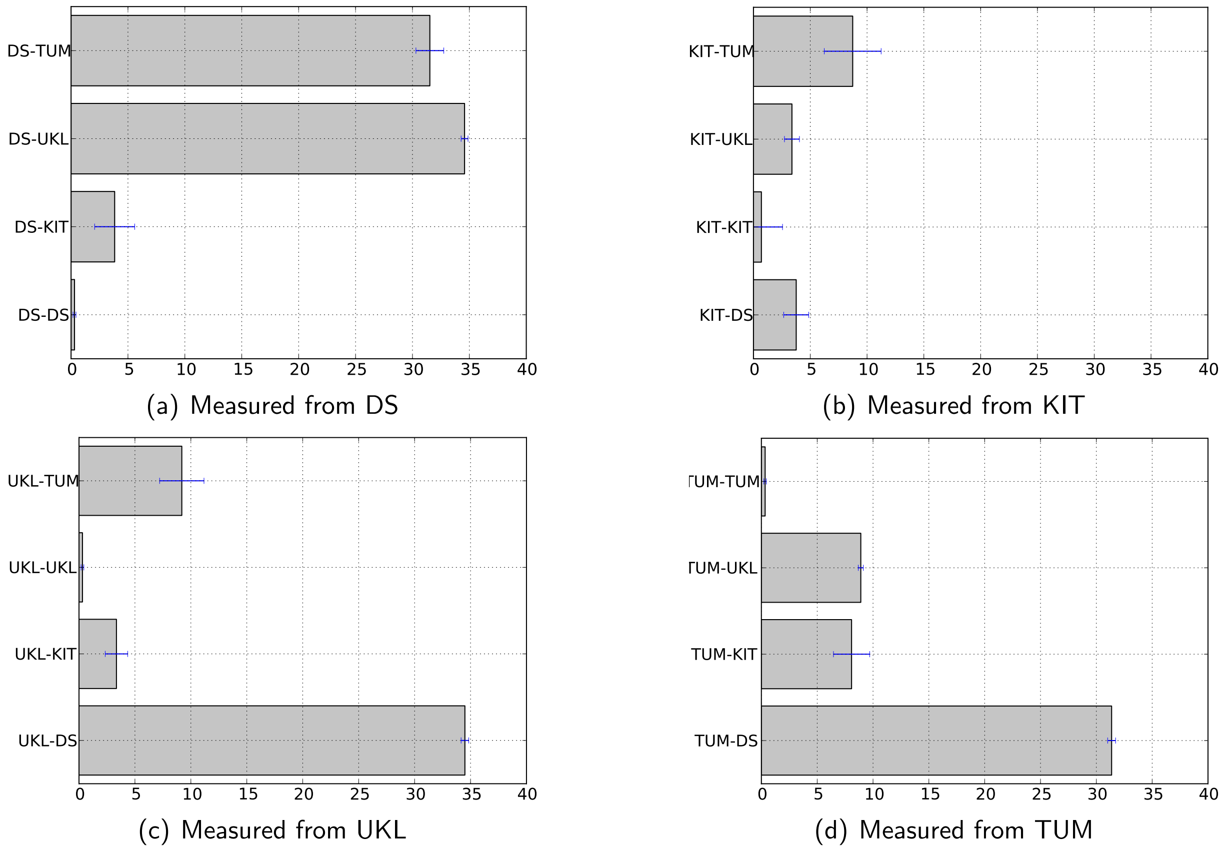


Figure 5.1 Average RTTs in milliseconds measured from location A to B (A-B)

In most cases the connections are similar valued in both directions, meaning that measured throughput from A to B is similar valued compared to the throughput measured from B to A. We can finally calculate an average throughput for inter-location communication from our test results making a value of approximately 46 Mbit/s.

This overview of the gLab nodes used for Peergroup testing shows that we have to deal with network links of strongly different bandwidth. This environment is comparable to the use of a Peergroup client exchanging data with peers located in the same LAN and with peers from a WAN. This is a typical scenario occurring when end-users use Peergroup at home. Analyzing the test results Peergroup delivers when used on the gLab, will enable us to make a statement about its behavior when used with differently speeded nodes.

5.2 Measurements

Knowing the capabilities of the gLab we are now able to correctly interpret measurements we do with Peergroup on the gLab. The tests cover scenarios of 5, 10, 20, 40 and 80 nodes distributing files sized 5MB, 10MB, 50MB, 100MB and 200MB. Every distribution of a file is repeated five times. As the result we calculate the average time in seconds from this five repeats for each node to fully receive the file. The averages of all nodes are then illustrated in barplots showing the average time needed to fully distribute the file among all nodes.

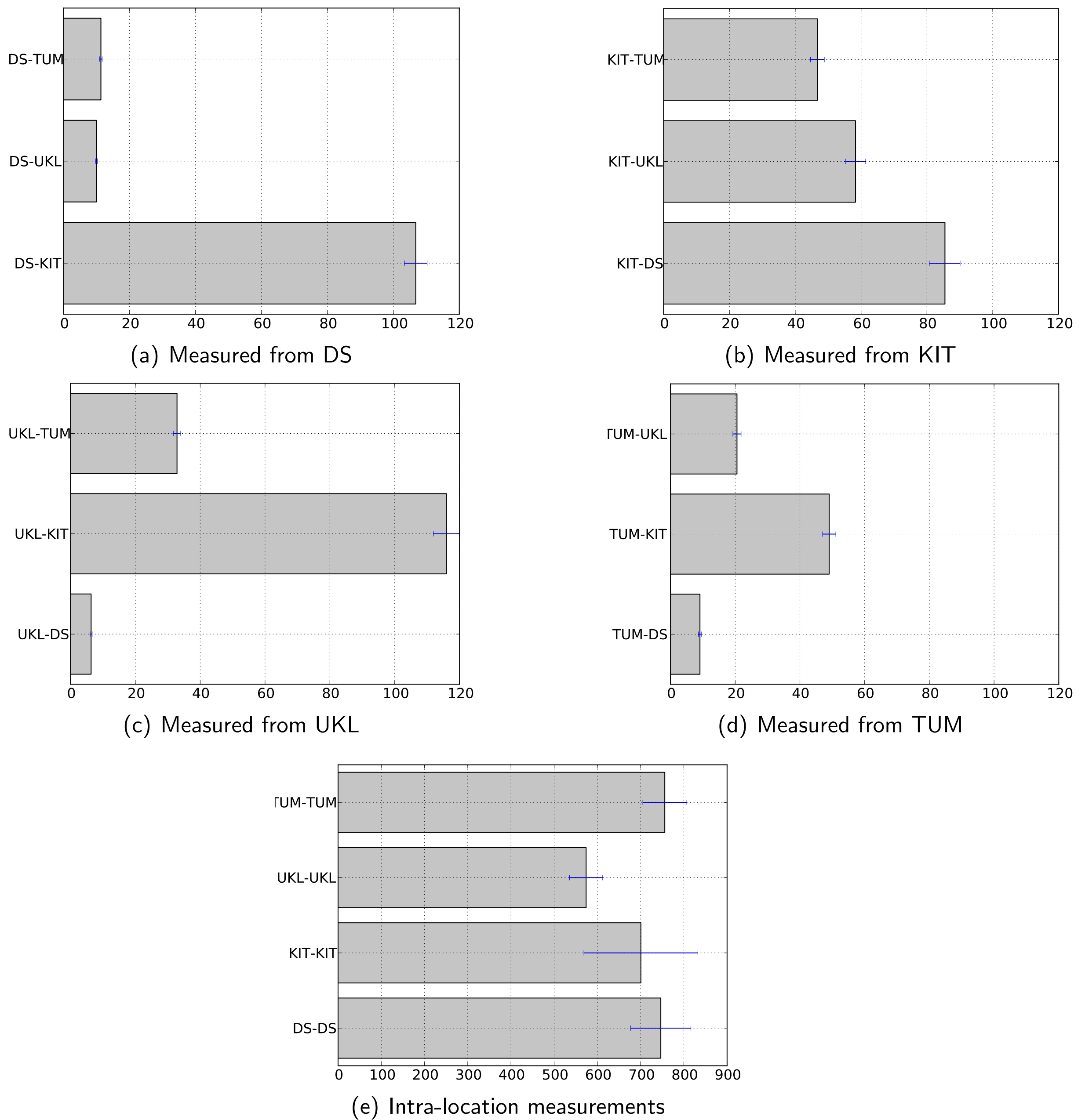


Figure 5.2 Average throughput in Mbit/s measured with Iperf from location A to B (A-B)

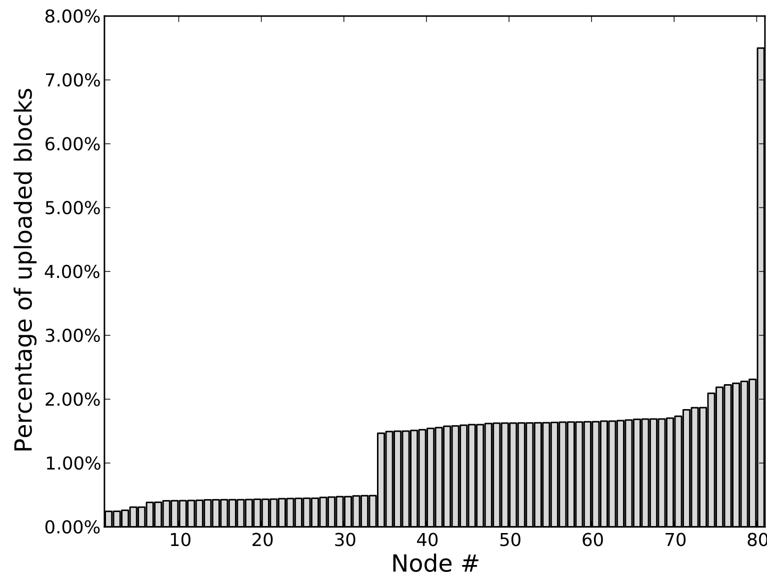


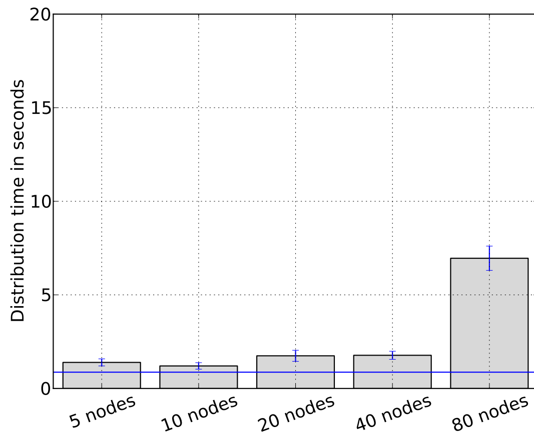
Figure 5.3 Distribution of chunk-download requests across the participating nodes

An important factor in distributed systems is fairness. In our case we mean fairness concerning the amount of chunk requests received by a single node. As far as possible, chunk requests should be equally distributed among all nodes. This is of course not possible if only single nodes are in possession of the required chunk. Only considering the set of nodes being in possession of the chunk an external node wants to download, Peergroup is fair by design. We have no preferences concerning the choice of the nodes when attempting to download a chunk as we always choose a random node from the set of possible nodes.

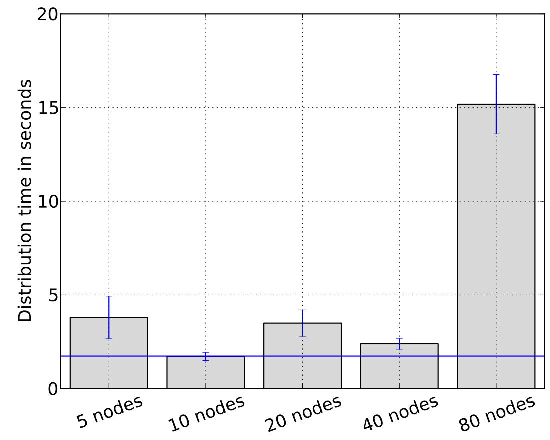
In Figure 5.3 we illustrate what percentage of requests each node processed after distributing and deleting a 100MB file six times in a row. Figure 5.3 states two characteristics. We initially see that the first group of nodes serves less requests than the remaining nodes. A look into the log files of the experiment reveals that all nodes belonging to this group are located at UKL. These nodes turn out to be slower than the other nodes in the gLab. Being slower they take longer to upload requested chunks. As each nodes handles a maximum of five simultaneous uploads, this leads to less chunk uploads for nodes from UKL in comparison to the other nodes. However, we can see that the chunk-download requests are quite equally distributed among the UKL nodes. The second characteristic is node 80. Being the inital uploader of the test-file this node has to handle more requests than other nodes. The remaining nodes have nearly the same percentage of handled block-download requests across one another, which states our chunk download strategy is fair.

For our measurements to be a good result we expect that the file distribution times are near the time we can expect from the average throughput of the gLab with regard to the filesize. At the same time this measure is an indication of how well the available bandwidth of the network is used. Furthermore, we expect to have little fluctuations in distribution times when distributing the same amount of data among different numbers of nodes.

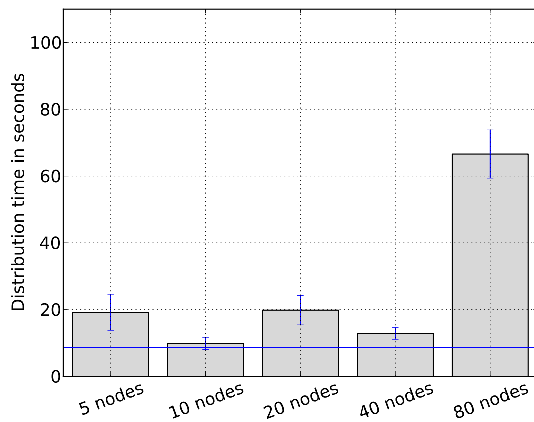
As the upload bandwidth of the initial uploader affects the distribution time, the node adding the test file to the network is always picked from the nodes located at KIT. In Figure 5.4 we show the results of the testings. Each diagram represents



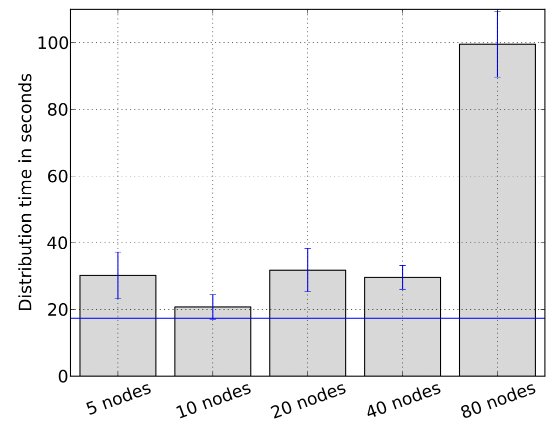
(a) Filesize: 5 Megabyte



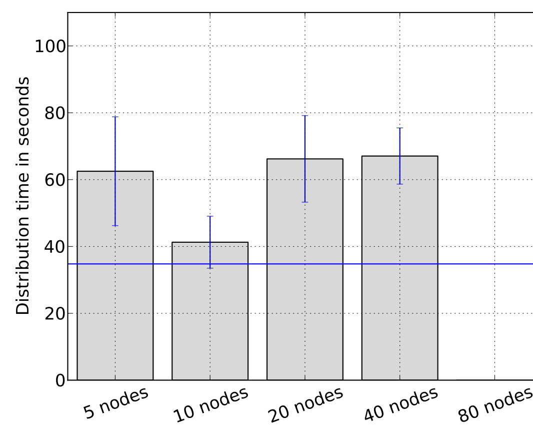
(b) Filesize: 10 Megabyte



(c) Filesize: 50 Megabyte



(d) Filesize: 100 Megabyte



(e) Filesize: 200 Megabyte

Figure 5.4 Distribution time in seconds reached by Peergroup

tests of a specific filesize with one bar standing for a specific amount of nodes (x axis) needing a specific amount of time in seconds (y axis) to download the new file and the blue line representing the possible download time according to the average throughput of 46 Mbit/s calculated from the Iperf measurements.

We can see that the measurements have a low standard error which is increasing with the tested filesize. Considering that the nodes are differently well connected with one another, it is clear that the standard error must increase with the filesize as the difference between fast nodes and slow nodes in download time is better recognized. Looking at the diagrams we can further see that the achieved file distribution time approximately stays the same for a fixed filesize no matter if the file is distributed among 5, 10, 20 or 40 nodes. The little fluctuations we can see within these cases can be explained by different connection capacities of the nodes and primarily by the random choice of the nodes. The tests handling 80 nodes delivered much higher results than the others. This is probably an effect of the amount of established connections used for downloading data. In our tests a peer connects to up to five other peers downloading a different chunk from each of them. In order to determine this statement we will need future tests with a variable amount of connections for chunk downloads.

All in all we were able to achieve a good scalability in tests with up to 40 nodes. This suffices for small sets of users sharing data, but is not feasible for use in larger environments. Tests allowing a higher number of established connections will show if Peergroup then will be able to scale with larger amounts of peers.

6

Related Work

Services providing synchronization of folder contents are famous nowadays. However, synchronizing data can be achieved in many different ways. Techniques used to achieve this goal are for example distributed file-systems, cloud-based storage services and file sharing protocols. We cover some examples of each technique such as Coda [27], Wuala [11], PAST [17] and BitTorrent [16]. In this chapter we classify Peergroup in the context of the mentioned techniques and talk about the advantages and disadvantages.

6.1 Distributed File Systems

Distributed file systems deliver functionality for a set of multiple users to access a common file system. Depending on the file system, data is stored either centralized on a Network Attached Storage (NAS), or data is spread over multiple servers avoiding a single point of failure. Much used systems are the Common Internet File System (CIFS) [21] and the Network File System (NFS) [29]. Both store data centralized on servers with NFS 4.1 adding support for clustering servers to improve scalability [28]. Slightly different approaches are proposed with the Andrew File System (AFS) [18] and its descendant Coda [27]. Like NFS, AFS and Coda aim to provide a UNIX compatible file system in order to maintain compatibility with existent applications. However, their underlying technique is deprecated as AFS and Coda were proposed in 1988 and 1990. AFS treats files as a unit and thereby is not able to benefit from the increased data distribution speed of block transfers. Both AFS and Coda store files centralized with Coda adding a "caching" layer between server and client in order to improve failure resilience [27].

Most distributed file-systems are still relying on servers to store data and thereby have a differentiation between clients and servers. More over, the initial ambition of most systems is to achieve access to the file-system within a local area network. Both, AFS and Coda started with the intention to provide users access to their data

at each workstation of a campus-network [18, 27]. Such an assumption includes that data is managed locally and transfers over internet connections are not intended. Peergroups ambition is to connect peers over internet connections and allow them to work on the same set of data avoiding centralized data storage.

6.2 Cloud-based storage systems

Most techniques found in current end-user storage services like Dropbox [1] and Wuala [11] are cloud-based. The term “cloud” describes services that are customizable on demand, be it processing time or storage space. As a consequence the data stored using these services is located at data centers of the respective provider which is at least questionable regarding data security and privacy [19] as data centers are subject to the different data protection regulations of the countries they are located in.

Wualas initial attempt of providing cloud space deals with a hybrid system of data-center-based storage and P2P storage. User data is not only stored in large data centers but each user is also able to contribute space from his own computer trading it for space granted by Wuala. The P2P file exchange is managed centralized by Wuala servers [12]. After Wuala was bought by LaCie they stopped using the P2P file storage and reverted to pure cloud storage. As all cloud-based approaches have centralized elements by nature, they require the user to register for their service. This empowers the providers to track user behavior.

Peergroup circumvents these problems as data is stored among the peers in the Acitivity-Stream. At first, the user only needs to create a XMPP account, no matter on which server. Existing accounts can be used without any problems rendering the registration of an additional account unnecessary. Second, the user creating a Peergroup network is able to decide which other users he wants to allow the access to the Activity-Stream, we can thus assume that all peers storing data are trustworthy. Or if we create a channel with less trustworthy peers we know not to share sensible data in the respective folder. The main advantage of Peergroup is the decentralized storage of data.

6.3 PAST

DHT-based storage systems like PAST [17] face a fully decentralized approach of storing data using DHTs to locate data chunks in a set of peers. Usually a node does not have much information regarding trustworthiness about the other nodes in a DHT, so DHT based storage systems must work with potentially untrustworthy nodes.

PAST [17] is a large-scale, persistent peer-to-peer storage utility providing a fully decentralized, DHT based storage facility. While based on Microsofts DHT *Pastry* [25], PAST aims at providing strong persistence, high availability, scalability and security [17].

PAST consists of a set of nodes organized in an overlay network. This overlay network is created using *Pastry* [25]. Each node is able to insert and retrieve data in a PAST network. Additionally, nodes are able to process routing requests from other nodes. PAST assumes that nodes are anonymous and potentially everyone can join a PAST network. To maintain security, data is asymmetrically encrypted using public and private keys provided by smartcards at each node [17]. However, it is possible to use PAST without smartcards using existing key pairs. As an result PAST does not require the peers to trust each other.

Unlike regular file systems data cannot be directly changed and deleted in PAST. Each file has a quasi-unique file ID allowing the localization and identification of this file in the network. As file IDs cannot be inserted multiple times, a changed file always results in a completely new file with a new file ID. To delete a file in a PAST network we reclaim the space that was previously allocated for the file. Regaining this space does not guarantee the deletion of the file. [17]

PAST benefits from the logarithmic amount of routing steps provided by the underlying *Pastry* DHT. However, PAST seems not feasible for active shared storage usage because of the limited abilities concerning file changes and deletions. Due to the direct exchange of signaling data with XMPP, Peergroup performs routing in $O(1)$.

6.4 File sharing protocols

File sharing protocols like BitTorrent [16] and Gnutella [3] typically provide functionality to distribute a fixed set of data across many nodes. Especially the techniques of BitTorrent have improved and are widely used today. The general ambition of file sharing protocols is to deliver an efficient way of distributing data across a varying set of peers. They share Peergroup's idea of decentralized data storage.

One general assumption posed by file sharing protocols is that the content that has to be shared is fixed. This means that the data is determined before the initial upload, and then is never changed. This assumption follows from the ambition of file sharing protocols to distribute content, and not to provide interactive access on the distributed data. However, it seems feasible to adopt techniques like the separation of files into blocks and the respective block download strategies in order to speed up the distribution of data (-changes) within a Peergroup network.

A related system utilizing P2P transfers is Novasky [22]. With Novasky a P2P-based Video on Demand (VoD) service is proposed. The goal is to provide a VoD service on a campus-network delivering up to 720p video with 1-2 Mbit/s. Next to a central storage for all videos, much frequented videos are cached in a P2P network of workstations at the campus. The caching of videos is done in a proactive fashion in order to provide resilience in cases of load peaks for videos with a high watching probability. Similar to Peergroup, data is distributed in blocks, but other than in Peergroup the blocks of a file are spread across all nodes following that no node is in possession of a complete (video)file. To compensate leaving nodes in the P2P network, Reed-Solomon encoding [34] of blocks is used to make non-available blocks reproducible up to a certain degree. In contrast to Peergroup this system is designed to work on a campus-network where good high network bandwidths can be assumed.

7

Conclusion

We proposed Peergroup, a P2P shared storage system using XMPP for the distribution of signaling data and P2P connections for effective data transfer. The ambition of Peergroup is to create a system allowing the user to share data with other users while always being straight about the locations where the data is stored. The combination of XMPP, which allows us to clearly determine the users to share our data with, and P2P data exchange ensuring that data is never stored at a central facility, lets the user know which peers are in possession of the data. We can thus be sure that sensitive data we share with trusted users will only be located at the devices of these users.

Peergroup is still in the beginning of its development process and the evaluation results show that Peergroup currently scales up to 40 nodes. Using this amount of nodes results in a reasonable distribution time in comparison to the best possible distribution time concerning the average inter-location bandwidth of the gLab. However, the evaluation results of the tests involving 80 nodes exceeded the expected times by far. One possibility that could reason these results is the limited amount of established connections. For the tests a peer is allowed to keep five simultaneous connections, resulting in five blocks being downloaded simultaneously. Each peer is allowed to serve an infinite amount of incoming block requests, which should not be a problem because of the randomization in the block request strategies. A closer investigation of the 80 nodes tests with file sizes of 100 Megabyte emphasized that the peers can be divided into two groups according to their downloading times. The first group of peers is well within the expectations and finishes downloading in less than 40 seconds, whereas the second group of peers starts with downloading times of 120 seconds with the last peers finishing around 170 seconds. In order to ensure that the distribution of signaling data does not slow down the distribution process, the node hosting the Activity-Stream is not included in the tests as a Peergroup node. We conclude that further evaluation is necessary to find the bottleneck which is currently avoiding scalability above 40 nodes. Possible tests to arise findings concerning our scalability problem cover scenarios with 80 nodes or more, varying the limit of established connections and further randomization of the choice of nodes.

The randomized choice of nodes should ensure that the results are not affected by the network environment of a certain node constellation in the gLab.

Besides the scalability problems we further conclude that Peergroup already provides a feasible solution for sharing folder contents with up to 40 nodes which is enough for middle-sized friend-to-friend networks. It is also feasible to use Peergroup for content distribution in environments with up to 40 nodes.

We finally conclude that the combination of using XMPP for signaling data, and P2P connections for user data exchange provides reasonable performance. Solving the stated scalability problems will make the XMPP node hosting the Activity-Stream the single limiting factor for scalability.

Peergroup offers several starting-points for future work as well. It is easy to enhance Peergroup to support SSL data transfers preventing the sniffing of data between two nodes. To provide better security it is desirable that the data is also encrypted locally. Furthermore, we need a solution to encrypt the signaling data as this data contains information about the files shared between the users. For sharing sensitive data it is not only wise to set a password for the Activity-Stream but it is also desirable to encrypt the signaling data. We need a system where the peers in the Activity-Stream securely agree on a common key for the encryption of signaling data without the chance for a third party to replicate this key by listening to the handshake of the other peers. It is further desirable to enhance Peergroup in order to allow a user to store private data in a Peergroup network which is only accessible for himself. As a consequence Peergroup would also be usable to backup private data in a Peergroup network. In order to provide user friendly usage in networks using Network Address Translation (NAT) we need to add further support for opening needed ports in a NAT performing router.

Bibliography

[1] Dropbox. Accessed: 2012.06.22. Online Resource. https://www.dropbox.com/developers.

[2] Facebook chat docs. Accessed: 2012.06.24. Online Resource. http://developers.facebook.com/docs/chat/.

[3] Gnutella documentation. Accessed: 2012.06.11. Online Resource. http://rfc-gnutella.sourceforge.net/.

[4] Google talk developer page. Accessed: 2012.06.24. Online Resource. https://developers.google.com/talk/open_communications.

[5] iNotify Manpage. Accessed: 2012.06.05. Online Resource. http://linux.die.net/man/7/inotify.

[6] Java atomic access. Accessed: 2012.06.16. Online Resource. http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html.

[7] List of XMPP servers. Accessed: 2012.06.07. Online Resource. http://xmpp.net/.

[8] Smack API. Accessed: 2012.07.03. Online Resource. http://www.igniterealtime.org/projects/smack/.

[9] Smack API: Packet Properties. Accessed: 2012.06.14. Online Resource. http://www.igniterealtime.org/builds/smack/docs/latest/documentation/properties.html.

[10] TBinaryProtocol java source file. Accessed: 2012.06.14. Online Resource. http://svn.apache.org/repos/asf/thrift/trunk/lib/java/src/org/apache/thrift/protocol/TBinaryProtocol.java.

[11] Wuala. Accessed: 2012.06.26. Online Resource. http://www.wuala.com/.

[12] Wuala tech talk. Accessed: 2012.06.27. Online Resource. http://www.youtube.com/watch?v=3xKZ4KGkQY8.

[13] Xmpp request for comments (rfc) 6120. Accessed: 2012.05.24. Online Resource. http://xmpp.org/rfcs/rfc6120.html.

[14] Xmpp request for comments (rfc) 6121. Accessed: 2012.06.07. Online Resource. http://xmpp.org/rfcs/rfc6121.html.

[15] Xmpp stanzas. Accessed: 2012.05.24. Online Resource. http://xmpp.org/rfcs/rfc6120.html#stanzas.

[16] Cohen, B. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems* (2003), vol. 6, pp. 68–72.

[17] Druschel, P., and Rowstron, A. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS VIII* (2001), pp. 75–80.

[18] Howard, J. H. An overview of the andrew file system. In *Winter 1988 USENIX Conference Proceedings* (1988), pp. 23–26.

[19] Ion, I., Sachdeva, N., Kumaraguru, P., and Čapkun, S. Home is safer than the cloud!: privacy concerns for consumer cloud storage. In *Proceedings of the Seventh Symposium on Usable Privacy and Security* (2011), ACM, p. 13.

[20] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565. http://doi.acm.org/10.1145/359545.359563.

[21] Leach, P. Cifs: A common internet file system. *http://www. microsoft. com/-mind/1196/cifs. asp* (1996).

[22] Liu, F., Shen, S., Li, B., Li, B., Yin, H., and Li, S. Novasky: Cinematic-quality vod in a p2p storage cloud. In *Proceedings of the 2011 INFOCOM* (april 2011), pp. 936 –944.

[23] Mhl, G., Fiege, L., and Pietzuch, P. *Distributed Event-Based Systems*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[24] Miller, B., Nixon, T., Tai, C., and Wood, M. Home networking with universal plug and play. *Communications Magazine, IEEE 39*, 12 (dec 2001), 104 –109.

[25] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, R. Guerraoui, Ed., vol. 2218 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001, pp. 329–350. 10.1007/3-540-45518-3 18. http://dx.doi.org/10.1007/3-540-45518-3_18.

[26] Saint-Andre, P. Xep-0045: Multi-user chat. *XMPP Standards Foundation* (2008).

[27] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*, 4 (apr 1990), 447 –459.

[28] Shepler, D., Eisler, M., and Noveck, D. Rfc 5661-network file system (nfs) version 4 minor version 1 protocol. *URL: http://tools. ietf. org/pdf/rfc5661. pdf, j an* (2010).

[29] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Rfc 3530: Network file system (nfs) version 4 protocol, 2003.

[30] Slee, M., Agarwal, A., and Kwiatkowski, M. Thrift: Scalable cross-language services implementation. *Facebook White Paper* (2007).

[31] Steinmetz, R., and Wehrle, K. *Peer-to-Peer Systems And Applications.* Lecture Notes in Computer Science. Springer, 2005.

[32] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference* (San Diego, California, August 2001).

[33] Tanenbaum, A. *Computer Networks.* Computer Science. Prentice Hall PTR, 2003.

[34] Wicker, S. B. *Reed-Solomon Codes and Their Applications.* IEEE Press, Piscataway, NJ, USA, 1994.