A real-time Streaming Protocol for large-scale P2P Networks

Bachelor Thesis

by

Christopher Michael Probst

born in
Solingen

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

October 2014

Supervisor:
Jun.-Prof. Dr.-Ing. Kalman Graffi

Abstract

This thesis concentrates on implementing and evaluating different distribution algorithms for large data transfers in heterogenous P2P networks, especially for incremental transfers, which yields the possibillity for P2P video streaming. The main focus is on $1 : n$ scenarios where only one peer has a given data set completely and n peers try to distribute this data set among themselves as fast as possible, though other scenarios are implemented and evaluated as well.

The main problem is the choice of the best distribution algorithm, which should be able to use most of the available network bandwidth of all participating peers. In a traditional client/server system the server uploads the data to all clients sequentially, which means that only the server upload bandwidth is used but none of the client upload bandwidths.

P2P networks in combination with efficient distribution algorithms can help to solve this problem. Those algorithms are always based to the same technique where all participating peers download specific chunks from the peer with the whole data set and upload those chunks to other peers. This way the upload bandwidths of the peers are not idle. The difficulty is the choice of the specific chunks the peers download and the tuning of parameters like chunk count and chunk size. Good algorithms are also flexible enough to adjust themselves to changing network conditions.

– work in progress

Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

Traditional network applications are mostly based on the client / server model, where the server only responds to client requests and the clients do not know each other. For some server applications more clients also require more upload bandwidth of the server in order to guarantee quality of service.

In case of real-time video streaming, a server should guarantee, that the ability to play a video fluently does not depend on the number of clients watching the video stream in parallel. In consequence of that, a video streaming server can fundamentally only serve a certain number of clients, which are able to watch the video stream in parallel, so this approach has scalability issues.

To improve this imbalance clients could use their own upload capacity to help the server distributing a given data set. These are called *Peer-to-Peer* networks, where every participant is called a *peer* and is connected to other peers in the network. If all peers have enough upload bandwidth, it is possible to create a Peer-to-Peer network, which is able to distribute a given data set evenly among all peers within a fixed period of time.

1.2 Objective

This thesis concentrates on implementing a Peer-to-Peer software, which is able to distribute a given data set among any number of peers and never exceeds $2\,T_0$, where T_0 is the time needed for a single transfer between two peers measured in seconds. To reach this goal different distribution algorithms have been implemented and evaluated to compare their advantages and disadvantages. The resulting software is able to evaluate these algorithms in terms of efficiency, overhead and performance. The

solution is generic and can be used for any kind of data transfers like video streaming or large file transfers.

1.3 Structure

The thesis starts with the evaluation of existing concepts in chapter 2 and compares them with the concepts, which are proposed and theoretically discussed in chapter 3. Then the architecture of the software is explained in chapter 4. The framework and application modules are presented in chapter 5 and 6 respectively. The evaluation of the software follows in chapter 7. The last chapter 8 presents future concepts based on the developed framework.

Chapter 2

Related work

Chapter 3

Theory

3.1 Model: Sequential

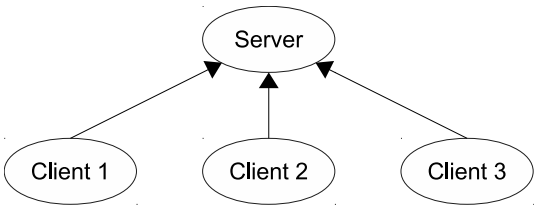


Figure 3.1: Sequential model

In the sequential (client / server) model, as shown in figure 3.1, every client is only connected to the server. The server transfers data sets to all clients in parallel. These peers share the upload bandwidth of the server. This concept does not scale well, because every new client connecting to the system slows down every other client connected to the same server.

To estimate the efficiency of this model, let b the bandwidth of the server in $\frac{bytes}{seconds}$, n the number of clients connected to the server and s the size of the whole data set in bytes. Then $T = n\,T_0$ is the time in seconds it takes to transfer the data set to all clients, where $T_0 = \frac{s}{b}$ is the time in seconds for a single transfer from the server to a client. To fulfill this formula, each client needs at least as much download bandwidth as the shared upload bandwidth of the server. Without adding more servers or using the upload bandwidth of the clients this linear relationship can not be removed. The implementation of this model is explained in section 5.3.0.5 and is obviously not able to keep the limit of $2\,T_0$ seconds, but it has been implemented to show the immense difference. A more efficient approach is presented in section 3.2.

3.2 Model: Logarithmic

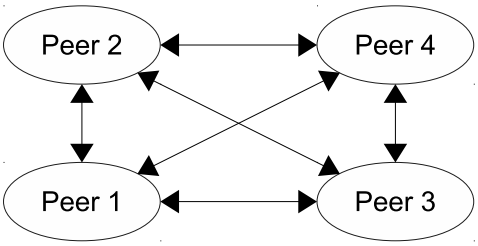


Figure 3.2: Peer mesh

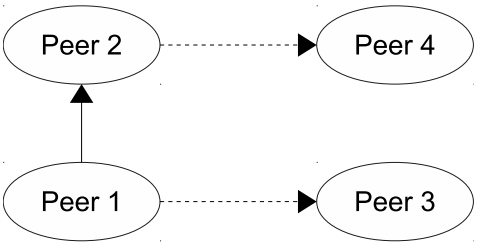


Figure 3.3: Logarithmic model

If a Peer-to-Peer model is used instead of the Sequential model, every so called peer is connected to one or more other peers depending on the used topology. In this thesis only the mesh topology is of importance, as shown in figure 3.2, where every peer is connected to every other peer in the network.

In the Logarithmic model every peer can do both uploading and downloading data sets, though peers are restricted to only upload to one other peer in parallel. If a new data set is announced by one peer in the Peer-to-Peer network, all peers try to download this data set from the given peer. Only one peer, basically the fastest one, is allowed to download the data set. The rest of the requesting peers are rejected and do nothing while the other peers download the data set. When the peer completed the download, the number of peers containing the data set changes to two. This time another two peers will be allowed to download the data set. Therefore, the number of downloading peers doubles periodically, which is considered exponential growth.

This model guarantees, that every peer has the complete data set in $T = \log_2(n)\, T_0$ seconds, where n is the number of peers in the network, $T_0 = \frac{s}{u}$ the time for a single transfer from one peer to another, s the size of the data set and u the upload bandwidth of each peer. This is only true for peers with a higher download bandwidth than upload bandwidth and applies to all Peer-to-Peer models presented in this chapter. In reality this makes sense, because clients connected through an *ISP (Internet Service Provider)* often fulfill this condition. All peers also have the same upload bandwidth u. Unfortunately, this is not so common in real world situations. But it can illustrate the relationship between the upload bandwidth and the number of peers. The figure 3.3 shows four peers, where peer one is the only peer with the data set at first. It transfers the data set to peer two in T_0 seconds. Then peer one and two transfer this data set to peer three and four in parallel, which also takes T_0 seconds. So the whole transfer takes $2\,T_0$ seconds.

In comparison to the Sequential model, this one increases the complexity of the implementation only marginal, but the efficiency is significantly higher. The more peers participate the higher is the efficiency. The overhead is also manageable, because during the transfers the idle peers do nothing except

waiting. As a major drawback of this model, there are never more than 50% of all peers uploading in parallel. Additionally, only one peer uploads the data set at the beginning, so for the first T_0 seconds no other peer is uploading. This model is implemented and explained in section 5.3.0.5. Finally, this model can not always keep the $2\,T_0$ limit. An improved version of this model is presented in section 3.3.

3.3 Model: Chunked-Swarm

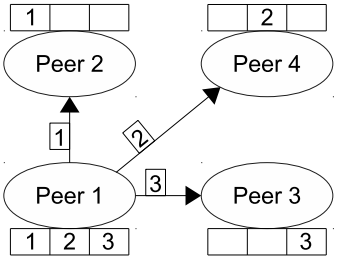


Figure 3.4: Chunked-Swarm model 1

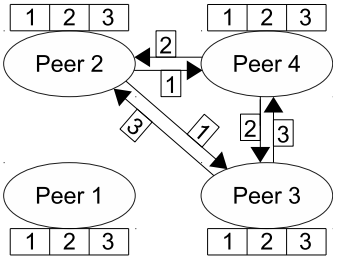


Figure 3.5: Chunked-Swarm model 2

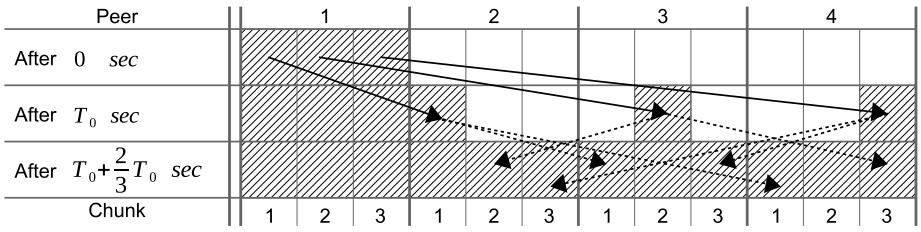


Figure 3.6: Chunked-Swarm chunk completion diagramm

The Chunked-Swarm model is an enhancement of the Logarithmic model. It allows multiple uploads in parallel and splits the data set into chunks before transferring it. Therefore, peers can not download the whole data set at once. Instead, they have to request each chunk individually. Since the size of these chunks is smaller than the size of the whole data set, peers can help to distribute the data set more quickly. The figure 3.4 and 3.5 show four peers distributing three chunks. At first peer one has the data set completely. The remaining peers each request one chunk in parallel. These peers will then distribute their chunks among themselves. They do so by announcing their own chunks to all other peers they are connected with, which can then request these chunks. It is important to note, that this model is always pull-based. A chunk is only uploaded, if a peer explicitly requests it.

Compared to the Logarithmic model, the implementation of this model is more complex, because the efficiency essentially depends on the right choice of chunks the peers request from each other. Since

this approach is pull-based, peers do not know, which chunks other peers request. Thus two peers may request the same chunk, which is called chunk duplication and decreases the efficiency considerably. In the context of figure 3.4 and 3.5, chunk duplication would occur, if all peers requested chunk one from peer one. Therefore, they could not exchange this chunk afterwards. An implementation of this model has to make strong considerations about this issue. The implementation of this model is described in section 5.3.0.6, which also explains the problem of chunk selection in more detail.

To determine the runtime of this model, distinct chunk selection is assumed, so there have to be at least as many chunks as peers, because otherwise chunk duplication can not be prevented. In figure 3.6 there are just as many chunks as peers, so it takes T_0 seconds to transfer all distinct chunks to all peers, because every peer gets a third of the available bandwidth and also has to download a third of the whole data set. Then the peers distribute their chunks among themselves, which takes $\frac{2}{3}\,T_0$ seconds, because every peer uploads its chunk to two other peers. After $T = T_0 + \frac{2}{3}\,T_0$ seconds every peer has all chunks available. Similiary a variable number of peers need $T(n) = T_0 + \frac{n-1}{n}\,T_0$ seconds. Since $0 < \frac{1}{n} \leq 1$ is always true, T never exceeds $2\,T_0$. If the chunk count is doubled, the model behaves even better, because the peers can start to upload their own chunks while they are downloading the next distinct chunks from peer one. This leads to the general formula $T(n,c) = T_0 + \frac{n}{c}\,\frac{n-1}{n}\,T_0 = (1 + \frac{n-1}{c})\,T_0$, where $c = n\,2^i, i \in \mathbb{N}_0$ is the chunk count. As a major disadvantage, the chunk count has to be chosen before the transfer can actually happen. Since the efficiency depends on the $\frac{n-1}{c}$ ratio, the maximal number of peers should be determined before. Without any further optimizations the main objective of this thesis is already fulfilled, as long as the chunk count is equal to or greater than the peer count and no chunk duplication occurs.

If the data set is first seperated into parts, which are then segmented into chunks, the data set can be downloaded sequentially, as long as each peer downloads part after part, starting with the first part. Each part then represents a data set as well, which simplifies the implementation of this concept. One domain relying on such a download behaviour is video streaming, where peers want to watch the video while downloading it. Unfortunately, this model introduces an initial delay, because the first part has to be downloaded completely before the video can be started. The duration of the delay depends on the number of parts and chunks and on the upload and download bandwidth of each peer. Those parameters should be chosen wisely to reduce this delay as much as possible and thus to improve the quality of service. This concept is explained and implemented in section 6.3 and chapter 8.

Chapter 4

Architecture

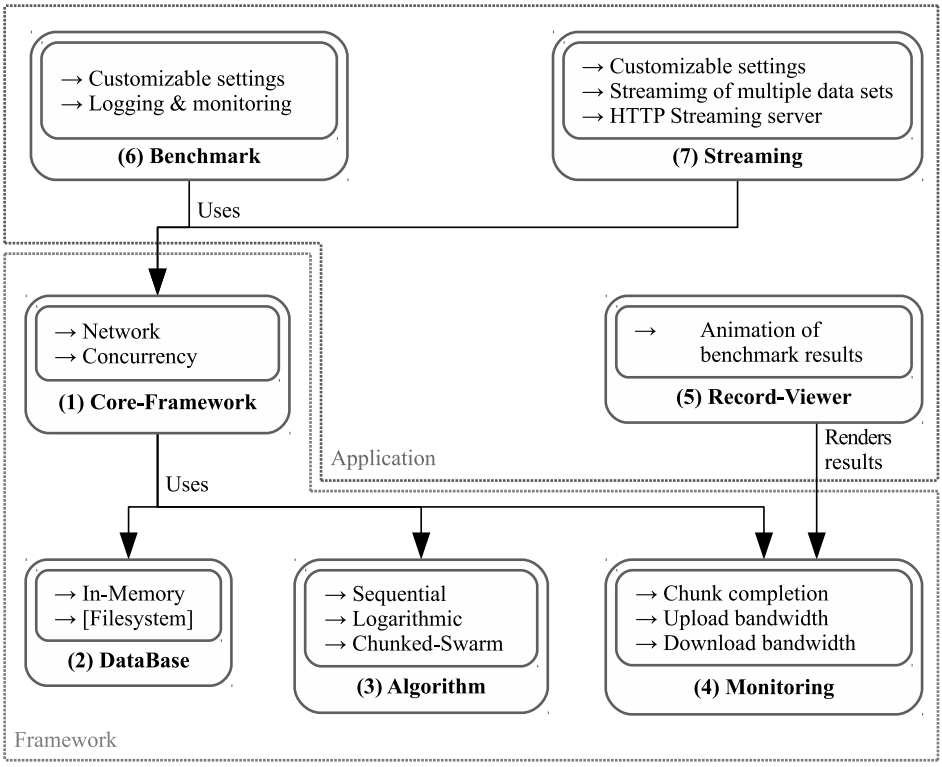


Figure 4.1: Software architecture

This chapter explains the architecture of the software in detail. From the beginning, one of the main goals were to achive high modularity, so individual parts can be enhanced or replaced easily. The software is separated into the framework and the application part, which consists of multiple modules. Figure 4.1 shows the architecture.

The application part depends on the Core-Framework and the Monitoring module. It contains three modules which are Benchmark, Streaming and Record-Viewer. The Benchmark module is the most

important application part, because it can help to create, monitor and evaluate different scenarios in terms of performance and efficiency. All measurements were done using this module. The Streaming module is related to future work and demonstrates the possibility for an incremental stream of data sets. The Record-Viewer module can render and animate results taken from the Monitoring module, which belongs to the framework part and thus only depends on it.

The framework part contains the Core-Framework, DataBase, Algorithm and Monitoring module. While the DataBase, Algorithm and Monitoring module can be replaced or even omitted the Core-Framework module is the most important module. It manages network, concurrency and the communication of the three other modules located in the framework part. The DataBase module represents an interface for a generic data storage, which might be the file system or even completely in-memory. The Algorithm module contains the implementations of the concepts presented in chapter 3. To implement and test new distribution algorithms only this module has to be modified. The Monitoring module records data like current bandwidth usage and chunk completion, which are important data for evaluating the distribution algorithms. This module creates *csv* files, which can be plotted with tools like *gnuplot* and a log file, which contains a chronological stream of events occured during the recording and can be rendered using the Record-Viewer module.

In the next two chapters the framework and application modules are explained in detail. The order is bottom-up, which means the Core-Framework module is explained first in section 5.1 followed by the DataBase, Algorithm and Monitoring module in section 5.2, 5.3 and 5.4 respectively. The application modules are explained in section 6.1, 6.2 and 6.3 referring to the Benchmark, Record-Viewer and Streaming module respectively.

Chapter 5

Framework Modules

5.1 Module: Core-Framework

5.1.1 Network

The network implementation uses the concept of *leechers* and *seeders*. A leecher connects to one or more seeders and downloads chunks from them. A seeder has multiple leechers connected to it and uploads chunks to them. Chunk transfers are pull-based, so they can only be requested by the leecher, while meta data or address announcements are push-based and thus transferred without being requested. Therefore, every leecher knows exactly what a seeder has to offer without asking, but only requests the chunks it needs. See section 5.1.1.3 for more information.

While leechers and seeders are independent of each other, it is possible to couple them together. A leecher coupled with a seeder always announces the address of its seeder to any other seeder it is connected with. This concept is explained in more detail in section 5.1.1.1.

Both, a leecher and a seeder, keep a reference to a database instance. A leecher stores downloaded data in its database and a seeder uploads data from its database. It is possible that both reference the same database instance, so the seeder is able to upload data, which was downloaded by the leecher. The DataBase module and all related concepts are explained in section 5.2.

Leechers and seeders are associated with one of the multiple distribution algorithms defined in the Algorithm module. These algorithms determine the distribution behaviour, which is further described in section 5.3.

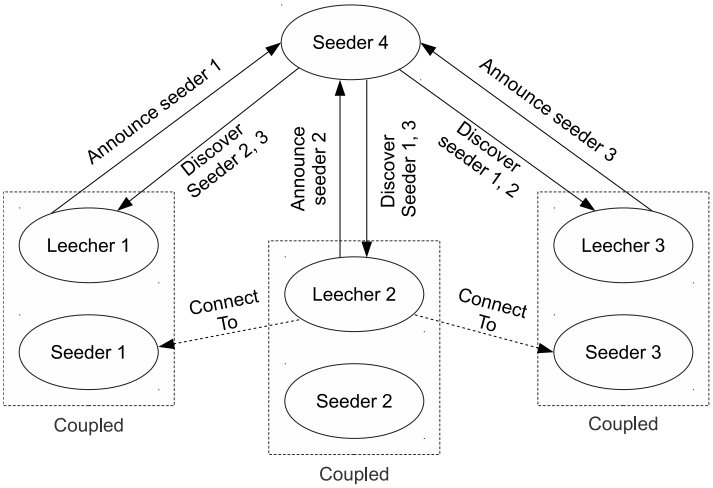


Figure 5.1: Seeder Discovery

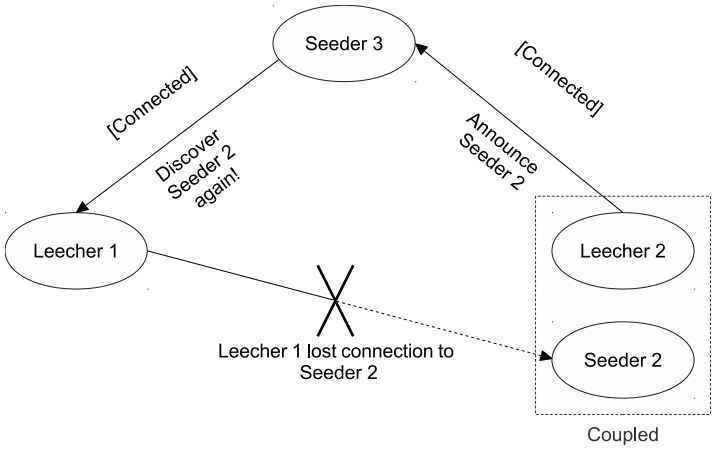


Figure 5.2: Reconnect

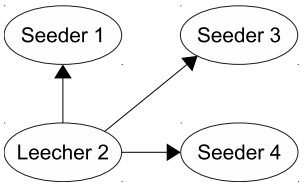


Figure 5.3: Leecher / Seeder mesh

5.1.1.1 Automatic Connect

Figure 5.1 shows in detail how the automatic connect mechanism works. Every leecher has to manually connect to a seeder for the first time, thereafter a leecher is able to automatically connect to other seeders. In order to provide this functionality every leecher, who is coupled with a seeder, announces the address of its coupled seeder to all other seeders it is currently connected with.

In this scenario leechers one, two and three announce the addresses of their seeders to seeder four, which broadcasts these addresses to all connected leechers. This way the leechers get to know each other and can connect to the remaining seeders. In figure 5.1 only the new connections of leecher two are illustrated, though leecher one and three connect to the other seeders as well. This works recursively, so if one of the seeders already has other leechers connected to it, these also will be

found. Finally, the topology is just a mesh with $(n-1) \cdot n$ connections, where n refers to the total number of coupled seeders and leechers and every leecher of each couple is connected to every seeder of all other couples. Figure 5.3 shows this from the point of view of leecher two, which is connected to all others seeders.

The mesh topology implies, that every peer knows exactly, what data sets other peers have and thus can request chunks efficiently, though it adds exponentially growing overhead and thus does scale indefinitely. To improve scalability, the number of connections per leecher can be limited, which also limits the knowledge of each leecher and thus the ability to make good decisions what to request from whom. This is explained later in section 6.1 in more detail.

If a leecher loses, for some reason, the connection to a seeder, the framework is able to reconnect, if the seeder is still online. This will be detected by periodic address broadcasts. So if, as shown in figure 5.2, leecher one lost the connection to seeder two but leecher one and two are also connected to seeder three, then seeder three will notify leecher one, that seeder two still exists.

5.1.1.2 Meta Data Announcements

After a leecher connected to a seeder, the leecher does not know, which data sets the seeder offers. As previously explained in section 5.1.1.1, leechers discover new seeders through seeders they are already connected with. The way a leecher collects knowledge about the data sets a seeder offers works in a similiar way. A seeder announces periodically its data sets to all connected leechers. It basically transfers a list of meta data, which is explained further in section 5.2.0.3, so every leecher can update its knowledge about this seeder. The list of meta data can be modified by the used distribution algorithm before it is announced to model a specific distribution behaviour. Section 5.3 outlines this mechanism.

An optimization uses a lazy transmission approach, thus a seeder will only transfer meta data, if it has changed. So if a data set does not change at all, a seeder will never transfer a second announcement of this data set. Another optimization causes the seeder not to transfer any announcements to leechers, which the seeder is uploading data to. This feature is further explained in the following section 5.1.1.3.

5.1.1.3 Download Requests

After a leecher successfully connected to one or more seeders and has already received meta data announcements, it can start to request chunks from the seeders. What to request from whom heavily

depends on the used distribution algorithm, which is explained in section 5.3 in more detail.

Since downloading more than one chunk at a time from the same seeder is not faster than downloading them sequentially, only one download per seeder at a time is allowed, which also reduces protocol complexity. Because of that, during a chunk download meta data and address announcements are not transferred, because these are obviously of no interest for the leecher. As soon as the chunk download completes, outstanding announcements will be bundled and transferred.

A seeder is allowed to decide whether a chunk request is valid or not, which means depending on the algorithm the seeder can reject a chunk request. If a seeder rejects a chunk request the leecher will not request the same chunk again. The seeder can tell the leecher, that a previously rejected chunk request is valid again by reannouncing this specific chunk. This way complex distribution behaviour can easily be implemented.

5.1.1.4 Nonblocking I/O

Network communication is based on *sockets*, which can either be datagram or stream based. Datagram sockets are connectionless, so they are of no importance for this thesis. Instead, the network implementation uses stream sockets. A stream socket is always connected to another socket. Programmatically, these two sockets behave like files. If data is written to a socket, it will be transferred to the other socket and can be read from it then. As with files, socket operations usually block the currently executing *thread* until the requested data is read or written, so this requires one thread per socket to handle multiple sockets concurrently. Unfortunately, threads are a limited resource, because they increase memory and *CPU* usage, this concept does not scale well.

To handle multiple sockets per thread, so called *nonblocking* sockets in combination with *event-polling* can be used. A nonblocking socket generally does not block on an operation, but returns an error instead, if it is not ready to execute this operation immediately. In combination with event-polling a thread can listen on sockets and be notified when these are ready. With this technique a single thread can handle tens of thousands of sockets.

Unfortunately, nonblocking sockets increase the complexity significantly, so the network implementation uses a framework to simplify the usage of them. The framework is called *Netty 5* and written in Java. Netty supports the separation of the transport procotol and the logic, so underlying transport protocols, like *In-VM*, *TCP*, *SCTP* or *UDT*, can be changed without affecting the logic of the application. By default, only In-VM and TCP is used, which are explained in section 5.1.1.5.

5.1.1.5 Transport: In-VM and TCP

Netty offers an In-VM, also called local, transport protocol, which does not involve network at all. It only works inside a single instance of the *JVM* and is perfectly suited for simulations or benchmarking. This is because, the local transport does not require (de-)serialization of data, which eliminates a huge amount of memory and CPU usage. Normally, the overhead of serialization is negligible, but the Benchmark module is able to simulate a large number of seeders and leechers, see section 6.1, so these overheads can sum up to a significant number. Therefore, the local transport seems to be the perfect fit for this purpose.

Netty also offers common transport protocols like TCP. It is used as a secondary option for the Benchmark module, which is useful to run distributed benchmarks on multiple machines, because local transport is limited to a single JVM instance.

5.1.1.6 Traffic-Shaping

In order to simulate and evaluate network applications, the ability to control the bandwidth to have comparable network conditions is needed. The problem with bandwidth limitation, which is also called traffic-shaping, is that the writer can not be throttled. The writer can either write as fast as possible or not write at all. In order to limit the bandwidth, the writer has to be started and stopped periodically. Since readers and writers are identical in terms of bandwidth limitation, the explanation always refers to writers but the same approach works for readers equally well.

If the limitation period is too long, the introduced bandwidth peaks will affect the measurements in a negative way, but if the period is too short instead, overhead will be increased. To perfectly limit the bandwidth an indefinitely short interval is needed. So, if the traffic-shaping shall work reliably the sweet spot between accuracy and minimal overhead has to be found. The network implementation uses a period of 250 ms, which offers enough precision, because the measurements are done every second. So every measurement contains the average of approximately four bandwidth limitation runs.

The complexity increases using multiple writers, who are not allowed to exceed a shared bandwidth. The main problem is fairness. It is not desired, that one writer gets 95 %, while the remaining shares only 5 % of the bandwidth. At the same time a single writer should be able to write at full speed, when no one else is writing. This complex behaviour can be achieved using a *leaky-bucket* algorithm and a *priority-queue*.

A leaky-bucket contains tokens, where a token equals one byte. If a writer wants to write a message, it will try to remove the tokens needed for the message to transfer. If the bucket does not have enough

tokens, the writer will take the remaining tokens and will decrease the size of the message it wants to write by the number of removed tokens. After this, the writer waits until the bucket gets refilled and repeats the procedure. Unfortunately, this does not fix the fairness problem.

To solve the fairness problem a writer is not allowed to remove tokens from the bucket directly. Instead it inserts the write request into a priority-queue. A writer job constantly polls the head of the queue and tries to process the write requests using the leaky-bucket. This time, the writer increases its number of written bytes according to the size of the written message. If the writer has more messages to write, it will insert itself into the queue according to the number of total written bytes compared to other writers. This way the head of the queue always is a write request belonging to a writer, which has written the fewest number of bytes in total. Thereby we get good fairness with almost no overhead, because the writer job runs only on demand in a thread pool and quits after all write requests are processed.

5.1.2 Concurrency

With the use of nonblocking sockets it is possible to handle a lot of sockets with only one thread. But this can be further improved, because modern CPUs always have multiple cores, whose quantity determines the number of threads able to run in parallel. For example, if a CPU has four cores and the application only uses one thread, the CPU will be utilized by only 25 %. Wherefore Netty uses more than one thread to increase the efficiency. In the next section 5.1.2.1 the utilization of multiple threads is explained in more detail.

5.1.2.1 Event-Loop

Netty has the notion of an *event loop*. An event loop is always running in its own thread and processes queued events one after the other. These events are mostly readiness events notifying a given socket is now readable, writable, acceptable or connectable.

To use more than one thread, Netty simply spawns more event loops, typically just as many as there are CPU cores. New sockets will then be distributed evenly among all running event loops. Even if the event loops get unbalanced, Netty will try to rearrange the sockets so that the event loops become balanced again.

These event loops can then run theoretically in parallel, because the underlying operating system distributes busy threads equally among all available cores. But using more than one thread is only the tip of the iceberg, multiple threads introduce a whole new category of problems and considerations, which have to be taken into account. The next section 5.1.2.2 discusses this a bit more in depth.

5.1.2.2 Lock-Free Progamming

The main problem using multiple threads is the question of how these threads interact which each other. If two threads access the same variable without any protection, the resulting behaviour is not always defined and can lead to untraceable bugs. The unprotected access of variables is called a *race condition* or *critical-section*.

A simple solution is to protect those sections with a mutually exclusive (mutex) lock, which simply forbids that more than one thread is entering the critical section at a time, but this decreases efficiency and increase thread context switching, which is expensive. One solution is to use lock-free algorithms, which exploit certain guarantees made by the Java memory model. The in-depth explanation of these

concepts would arguably exceed the boundries of this thesis, but it should be noted, that the network implemention heavily uses those algorithms.

The reason why those concepts were considered is that inter-thread communication was a key design choice in order to simplify the architecture of the implementation. For instance a leecher can be connected to a lot of seeders receiving a lot of meta data and address announcements. In order to effectively request chunks from the seeders, the leecher has to collect the information from all connections, which may run in different event loop threads and thus are not protected against concurrent access by default. Since this collection is a common task, it should be optimized as well. Using so called *compare and swap (CAS)* and volatile operations the necessary mutexes can be reduced to a minimum.

Usually, the overhead of mutexes can be ignored, but the simulation of hundreds of leechers and seeders on multiple cores means a lot of cross thread access. If every collection of meta data of each leecher would stop all event loop threads by using a mutex, the overhead should start to be noticeable. These impacts are hard to measure or quantify, though. But in principle the lock-free approach scales better. As a rule of thumb lock-free algorithms should always be preferred in performance critical sections. They can also help to prevent *dead locks*, which happen when two threads are waiting for resources locked by each other, and thus both wait forever.

5.2 Module: DataBase

The DataBase module represents a generic interface for storing binary data sets as chunks in any kind of storage. It has the responsibility to verify written chunks in terms of consistency and validation. But to do so the database has to know about the structure of the data set. Therefore, the meta data, which describes a given data set, is stored together with the data set as key-value pairs. The next section 5.2.0.3 describes the meta data format in more detail.

5.2.0.3 Meta Data

The meta data contains information about a given data set. These information are transferred between seeders and leechers to describe what data sets are available, and thus which chunks can be requested. The following information are stored in the meta data: *Name*, *Description*, *ID*, *Size*, *Chunks*, *Hash* and *ChunkedHashes*.

The Name and Description fields are used for human related tasks. The ID is used for streaming purposes, which will be explained later in section 6.3. The Size field determines the total size of the data set including all chunks, which leads to the next field. The Chunks field is a bit set, which simply stores one or zero bits for existing or missing chunks respectively. The length of the bit set determines the number of chunks a data set has. The Hash field contains a checksum generated by a hash algorithm like *SHA-1* from the whole data set. The ChunkHashes field does the same but for each chunk individually. This way the database can verify single chunks as well as the data set in total. The reason why there is an extra Hash field for the whole data set is to reduce the chance of hash collision. Two data sets are equal if all of their fields are also equal.

5.2.0.4 Backends: In-Memory and Filesystem

The *In-Memory* backend stores data sets in main memory and is not persistent. It is a useful backend for situations, where data sets are generated or consumed on the fly like video streaming or test scenarios. In case of streaming, one could simply add data set sources, which load chunks from a live camera or a similiar device. The Benchmark module uses this backend to reduce overhead introduced by the filesystem. This backend almost has no overhead other than memory usage. Future work may implement a persistent database based on a filesystem. This filesystem can also be virtual like a ZIP file. This backend is more rigid but also very important for file-sharing applications.

5.3 Module: Algorithm

This module contains the algorithms which determine the distribution behaviour. While the Core-Framework module takes care of all the low-level parts the real work is done here. This module defines two interfaces for both, a seeder and a leecher, which are called *SeederDistributionAlgorithm* and *LeecherDistributionAlgorithm* respectively.

A SeederDistributionAlgorithm can modify meta data, which is announced by a seeder and allows or rejects chunk requests. A LeecherDistributionAlgorithm is even simpler, it can only request chunks. A simplified implementation of both interfaces would look like this:

```
interface SeederDistributionAlgorithm {
    List of MetaData modifyMetaData(List of MetaData)
    Bool isChunkRequestAllowed(ChunkRequest)
}

interface LeecherDistributionAlgorithm {
    List of ChunkRequest requestChunks(AvailableDataSetsOfAllSeeders)
}
```

These interfaces are totally independend of the underlying network implementation. As a core concept, a leecher algorithm is only interested in requesting chunks as efficient as possible. The seeder can decide if a requested chunk request is allowed or not and what meta data it announces. With those two methods a seeder can model the behaviour of a seeder which uploads to every leecher in parallel, or only to one leecher in total, or even stop announcing meta data, if the associated data sets have already been transferred, which is used by the ChunkedSwarm model and explained in section 5.3.0.6. In the next sections all implemented distribution algorithms are compared and explained in detail.

5.3.0.5 Algorithm: Sequential and Logarithmic

The Sequential and Logarithmic model, explained in section 3.1 and 3.2 respectively, both share the same LeecherDistributionAlgorithm, which is the *OrderedSingleMostLeecherDistributionAlgorithm*. This algorithm chooses a seeder, which has the complete data set and requests it from this seeder. This algorithm does only work with a chunk count of one and never requests a data set from more than one seeder in parallel.

For seeding the Sequential model uses the *DefaultSeederDistributionAlgorithm*, which uploads data sets to every leecher asking for one. It is important to note, that this model has only one seeder and many leechers. All leechers are connected to this seeder and download data sets sequentially.

The Logarithmic model is fundamentally different in this regard, because every peer has a leecher coupled with a seeder and is connected to every other seeder of the network. This model uses the *LimitedSeederDistributionAlgorithm* for seeding and uploads data sets only to one leecher in parallel. During an upload other requests are rejected. The meta data of data sets which are being uploaded are also removed from the meta data announcement set, so that new leechers will not request these data sets. After an upload is complete, the meta data of the particular data set is announced again.

5.3.0.6 Algorithm: ChunkedSwarm

The ChunkedSwarm model allows multiple uploads in parallel and requires a chunk count equal to or greater than the number of peers. Every seeder uses the DefaultSeederDistributionAlgorithm and thus uploads chunks to every leecher. The leechers use the *OrderedChunkedSwarmLeecherDistributionAlgorithm*, which is similiar to the OrderedSingleMostLeecherDistributionAlgorithm, but requests single chunks instead of complete data sets and does download from any seeder the leecher is connected with in parallel, if this is possible.

To effectively download from as many seeders as possible, all seeders are sorted by the number of chunks they offer in ascending order, but without the chunks, which are currently being downloaded by the leecher and those, which are already downloaded. The leecher then traverses the sorted list of chunks and requests one chunk from each seeder. After each request the list is sorted and filtered again, so that the chunk the leecher just requested is not requested from an other seeder as well.

This way the leecher always chooses the best combination of chunk requests. If there are multiple chunks per seeder, which are equally well suited for downloading, the algorithm picks one chunk randomly. In consequence of this, chunk duplication cannot be prevented, though minimized if a high chunk count is choosen. Although, the efficiency of this model was determined with perfect chunk selection in mind.

Since random chunk selection can not always be perfect, an additional so called *SuperSeederDistributionAlgorithm* can be chosen. It is basically a DefaultSeederDistributionAlgorithm which remembers all chunks it has already uploaded and rejects any further attempts to download these chunks. It is typically used by only one seeder in the network, which is then called a *super seeder*. This way chunk duplication cannot occur, because leechers, whose chunk requests are rejected, simply request another chunk and request the rejected chunk later from normal seeders. The method guarantees that

every leecher requests a distinct chunk from the super seeder, which has the complete data set at the beginning. The other seeders still use the DefaultSeederDistributionAlgorithm. The problem is, that this technique is not safe against peer loss, which means that chunks uploaded to a peer, which goes offline, are not retransferred by the super seeder by default. This relates to future work and could be solved by periodic checks which peers are still online.

5.4 Module: Monitoring

To measure the quality and efficiency of the framework, the Monitoring module collects data like current bandwidth usage, total transferred bytes and chunk / data set completion timestamps from every peer. The current upload bandwidth usage and total uploaded bytes are only collected from seeders, while the current download bandwidth usage and total downloaded bytes are only collected from leechers. That is because leechers upload almost nothing, they only announce the address of their coupled seeder once and upload a chunk request for each chunk. These chunk requests are very small in terms of memory usage and usually are only transferred once every few seconds, depending on the chunk size. In consequence of that, the seeders only download those address announcements and chunk requests from their leechers, so seeders download almost nothing as well.

The collected data is then stored in csv files, which contain columns for every leecher and / or seeder, depending on the collected data. Because these files alone are not very meaningful, a small collection of python scripts helps to manage them. For instance, if there a multiple csv file sets from many runs, the merge script can calculate the mean and the confidence interval of all runs. Also, additional csv files are created from existing csv files, like the SortedTotalUploadedBandwidth.csv and SortedChunkCompletion.csv files which are, as the name let assume, sorted. In case of the SortedTotalUploadedBandwidth.csv file, the entries are sorted by the amount of uploaded bytes and the SortedChunkCompletion.file is sorted by the time needed to complete all chunks / data sets. Both files are sorted in descending order.

Besides the csv files, the Monitoring module can create a log file, which contains a chronological stream of events occured during a run. Amongst others, these events are chunk requests, starts of uploads and downloads and chunk / data set completions. The content of the log file can be rendered using the Record-Viewer module, which is explained in section 6.2. Using this module the process of the distribution can be observed at any point.

Chapter 6

Application Modules

6.1 Module: Benchmark

The Benchmark module

6.2 Module: Record-Viewer

6.3 Module: Streaming

Chapter 7

Evaluation

Chapter 8

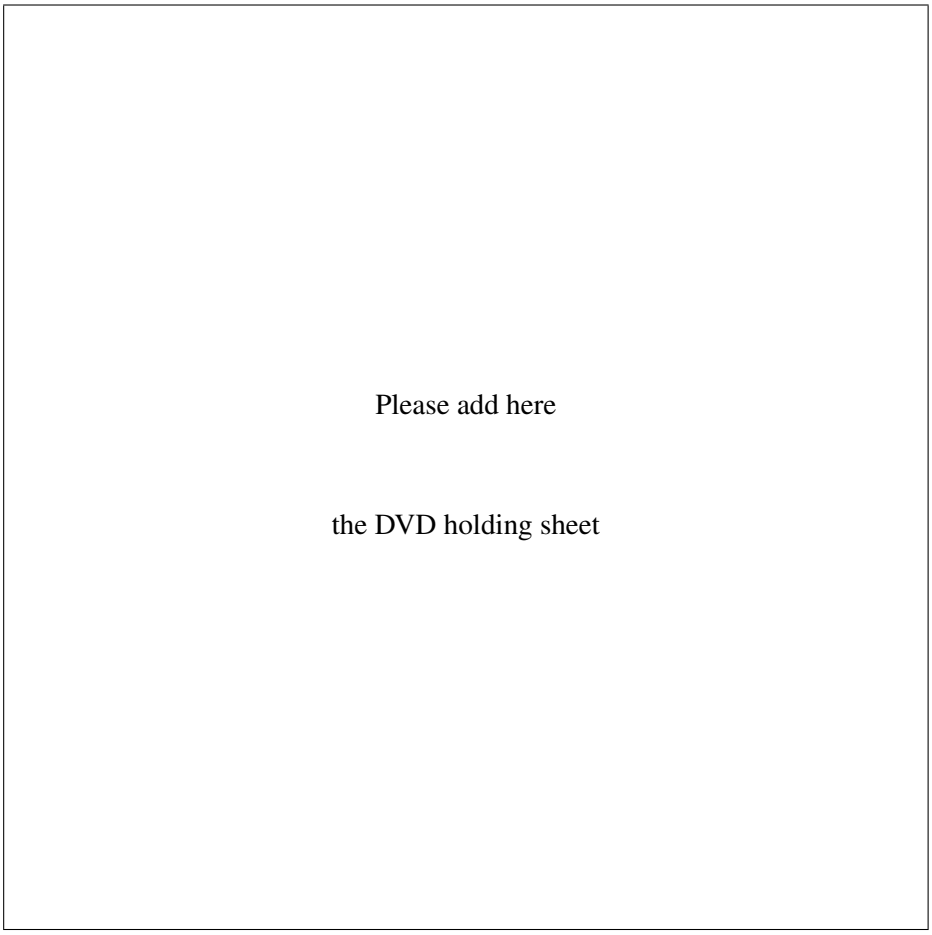
Future work

Bibliography

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 1.October 2014 Christopher Michael Probst



This DVD contains:

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers