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 peer-to-peer networks, especially for incremental transfers, which yields the possibillity for peer-to-peer 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 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.

Peer-to-peer 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 load specific chunks from the peer with the whole data set and distribute those chunks among themselves. 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.

In the course of this thesis an application was developed which implements a variation of the BitTorrent protocol, I call this variation Chunked-Swarm-Distribution (CSD), which is better suited for the needs of this thesis in terms of performance and time, a logarithmic distribution algorithm and a sequential client/server disitribution algorithm for comparison. The application is implemented in Java using the Netty framework, a high performance network library. This framework offers the possibillity to run simulated network connections which is great for monitoring and evaluating different distribution algorithms since real networks add too much overhead for testing. The application implements a couple of comparing benchmarks to evaluate the quality of the algorithms used.

The benchmark results show that the CSD algorithm is able to almost fully load all participating peers which means that if a transfer from peer A to peer B takes T seconds, the time taken for the complete distribution for N peers also is about T seconds, in practice it is often a bit over T seconds, which is caused by meta data overhead. This means that the number of participating peers is, theoretically, of no importance. In practice more peers means more meta data, but with compression and efficient data formats those influences can be minimised.

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 ix

List of Tables xi

1 Introduction 1
- 1.1 Problem and Motivation . . . 1
- 1.2 Objective . . . 1
- 1.3 Structure . . . 2

2 Related work 3

3 Theory 5
- 3.1 Model: Client/Server . . . 5
- 3.2 Model: Logarithmic . . . 7
- 3.3 Model: Chunked-Swarm . . . 8

4 Architecture 11

5 Framework Modules 13
- 5.1 Module: Core-Framework . . . 13
  - 5.1.1 Network . . . 13
    - 5.1.1.1 Automatic Connect . . . 14
    - 5.1.1.2 Meta-Data Announcements . . . 15
    - 5.1.1.3 Download Requests . . . 15
    - 5.1.1.4 Nonblocking I/O . . . 16
    - 5.1.1.5 Transport: Local and TCP . . . 17
    - 5.1.1.6 Traffic-Shaping . . . 17
  - 5.1.2 Concurrency . . . 19
    - 5.1.2.1 Event-Loop . . . 19
    - 5.1.2.2 Lock-Free-Progamming . . . 19
- 5.2 Module: DataBase . . . 21
    - 5.2.0.3 Meta-Data . . . 21

5.2.0.4 Backends: In-Memory and Filesystem 22
5.3 Module: Algorithm 23
5.3.0.5 Algorithm: Sequential 23
5.3.0.6 Algorithm: Logarithmic 23
5.3.0.7 Algorithm: ChunkedSwarm 23
5.3.0.8 Algorithm: SuperSeederChunkedSwarm 23
5.4 Module: Monitoring 25

6 Application Modules **27**
6.1 Module: Benchmark 27
6.2 Module: Record-Viewer 27
6.3 Module: Streaming 27

7 Evaluation **29**

8 Future work **31**

Bibliography **33**

List of Figures

3.1 Client/Server distribution 5
3.2 Peer mesh 7
3.3 Logarithmic distribution 8
3.4 Chunked-Swarm distribution 9
3.5 Chunk completion diagramm of the Chunked-Swarm distribution 1 9
3.6 Chunk completion diagramm of the Chunked-Swarm distribution 2 10

4.1 Software architecture 11

5.1 Seeder Discovery 14
5.2 Leecher/Seeder mesh 14
5.3 Reconnect 15

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, a higher number of clients means a higher 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. The consequence of that is, 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 unbalance the clients could use their own upload capacity to help the server distributing a given data set. Those networks are P2P 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 P2P network, which is able to distribute a given data set evenly among all peers in a fixed time relative to the number of participating peers.

1.2 Objective

This thesis concentrates on implementing a P2P network, which is able to distribute a given data set among any number of clients and never exceeds $2 \cdot T_0$ where T_0 is the time needed for a single transfer between the server and a client measured in seconds.

To reach this goal different distribution algorithms are implemented and evaluated to compare advantages and disadvantages. The resulting software is able to measure those 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 evaluating existing concepts in chapter 2 and comparing 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: Client/Server

In the 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 and thus all clients 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.

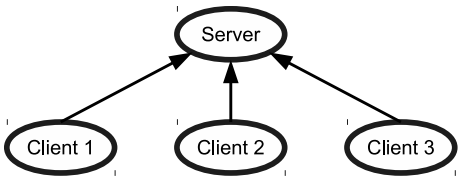


Figure 3.1: Client/Server distribution

If b is the bandwidth of the server in bytes/second, n the number of clients connected to the server and s the size of the whole data set, then $T = n \cdot T_0$ is the time in seconds it would take to transfer the data set to all clients, where $T_0 = \frac{s}{b}$ is the time in seconds for a single tranfser from the server to a client. This formula assumes, that the download bandwidth of each client is bigger than the shared upload bandwidth. This assumption applies to all models presented in this chapter. In reality this makes sense, because clients often have a much higher download bandwidth than upload bandwidth. Without adding servers or using the upload bandwidth of the clients this linear relationship cannot be removed. This traditional model is implemented in the Sequential algorithm, which can be found in section 5.3.0.5.

From the very first beginning it was clear, that this model is not able to keep the limit of $2 \cdot T_0$ seconds but it is implemented to show the immense difference. The next section 3.2 presents a much better

model.

3.2 Model: Logarithmic

If a P2P network is used instead of a client/server model, where every participant is called a peer and has a connection to one or more other peers in the network. The number of connections depends 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.

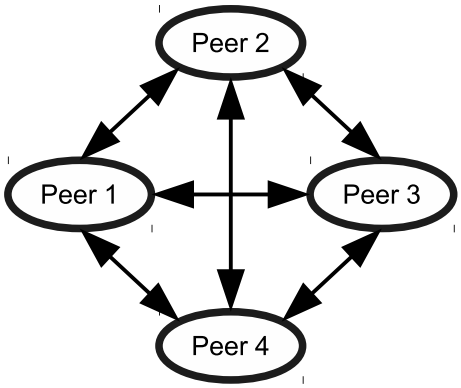


Figure 3.2: Peer mesh

In the logarithmic model every peer can do both uploading and downloading data sets. The only restriction is, that it can only upload to one other peer in parallel. So if a new data set is announced by one peer in the P2P network, all peers try to download this data set from the given peer. Only one peer, basically the first one, is allowed to download the data set. The rest of the requesting peers are rejected and do nothing after this. When the peer completed the download, there are two peers in the network containing the data set. This time another two peers will be allowed to download the data set. This way the number of downloading peers is doubled every $T_0 = \frac{s}{u}$ seconds, where u is the upload bandwidth of each peer, which is considered exponential growth.

At first, this concept might seem even worse than the client/server model, but it turns out, that this model guarantees, that in $T = \log(n) \cdot T_0$ seconds every peer has the complete data set. This formula assumes that every peer has the same upload bandwidth u, which is not 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 4 peers, where peer 1 is the only peer with the data set at first. It transfers the data set to peer 2 in T_0 seconds. Then peer 1 and 2 transfers this data set to peer 3 and 4 respectively in T_0 seconds. So the whole transfer takes $2 \cdot T_0$ seconds.

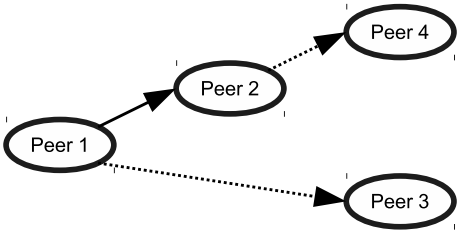


Figure 3.3: Logarithmic distribution

The implementation of this model increases the complexity only marginal, but the efficiency is significantly higher. The more peers there are participating the higher is the efficiency. The overhead is also very manageable because during the transfers the idle peers do nothing except waiting. The problem of this model is that never more than 50% of all peers are uploading in parallel. Also at the beginning only one peer uploads the data set. This means that in the beginning a majority of all peers do not use their upload bandwidth. This model is implemented in the Logarithmic algorithm, which can be found in section 5.3.0.6.

At the end this model cannot provide the $2 \cdot T_0$ limit we are looking for. To do so this model has to be improved, which is done in the next section 3.3.

3.3 Model: Chunked-Swarm

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. So peers cannot download the whole data set at once, but have to request each chunk individually instead. Since the size of those chunks is smaller than the size of the whole data set, the peers can help to distribute the data set more quickly. The figure 3.4 shows 4 peers distributing 3 chunks. At first peer 1 has the data set completely. Peer 2, 3 and 4 each request one chunk in parallel. Those peers will then distribute their chunks among themselves. This means that those peers announce their own chunks to the other peers, which can then request those chunks. It is important to note, that this model is always pull-based. A chunk is only uploaded if a peer explicitly requests it.

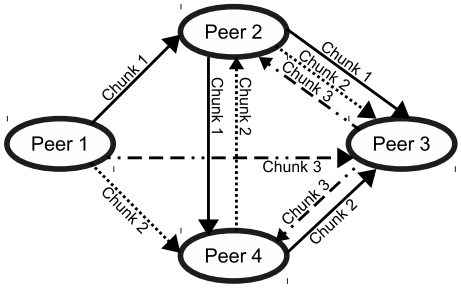


Figure 3.4: Chunked-Swarm distribution

The implementation of this model is not as easy as the logarithmic one, because the efficiency totally depends on the choice of the chunks the peer 2, 3 and 4 request from peer 1. Since this approach is pull-based, the peers do not know what the other peers request. This could lead to chunk duplication which decreases the efficiency considerably. Chunk duplication means that two peers request the same chunk from peer 1 and therefore cannot exchange those chunks afterwards. An implementation of this model has to make strong considerations about this issue. This model is implemented in the ChunkedSwarm and the SuperSeederChunkedSwarm algorithm, which can be found in section 5.3.0.7 and 5.3.0.8 respectively. Those sections also explain the problem of chunk selection in more detail.

To determine the efficiency of this model, it is assumed, that the chunk selection works perfectly, which means that all peers request distinct chunks from peer 1. This also implies that there have to be at least as many chunks as there are peers. Less chunks would also work, but then chunk duplication cannot be prevented.

For simplicity it is also assumed, that there are just as many chunks as there are peers. It this case, it would take 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. After this, the peers distribute their chunks among themselves, which takes $\frac{2}{3} \cdot T_0$ seconds, as shown in figure 3.5.

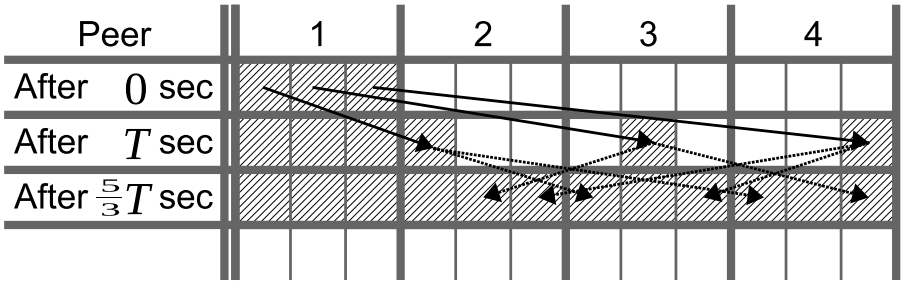


Figure 3.5: Chunk completion diagramm of the Chunked-Swarm distribution 1

This means that after $T = T_0 + \frac{2}{3} \cdot T_0 = \frac{5}{3} \cdot T_0$ seconds every peer has all chunks available. Similiary a variable number of peers need $T(n) = T_0 + \frac{n-1}{n} \cdot T_0 = \frac{2n-1}{n} \cdot T_0 = (2 - \frac{1}{n}) \cdot T_0$ seconds. A very nice property of this formula is that the result is always smaller than $2 \cdot T_0$. 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.

If the chunk count is doubled, the formula gets even better. Figure 3.6 illustrates this scenario. The efficiency rises because the peers can start to upload their own chunks earlier. After $\frac{1}{2} \cdot T_0$ seconds the peers received the first three distinct chunks from peer 1. At this point the peers can exchange their chunks, which takes $\frac{1}{3} \cdot T_0$ seconds. So after $\frac{5}{6} \cdot T_0$ seconds all peers have the first 50% of the data set. At T seconds the last three distinct chunks arrive which will also be exchanged afterwards. So after $T = T_0 + \frac{1}{3} \cdot T_0 = \frac{4}{3} \cdot T_0$ seconds all peers have the data set.

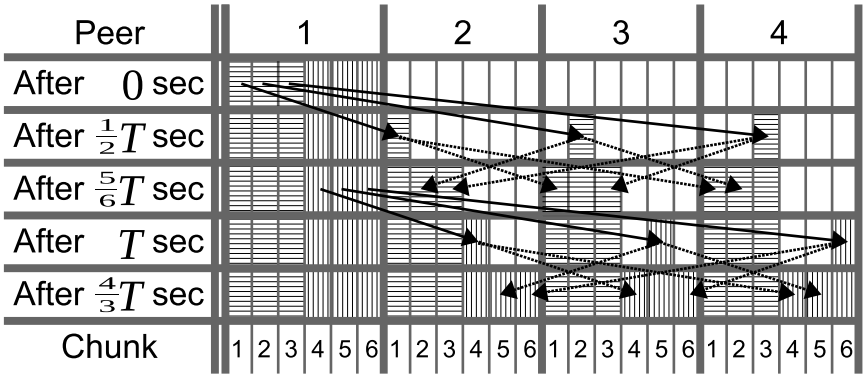


Figure 3.6: Chunk completion diagramm of the Chunked-Swarm distribution 2

Chapter 4

Architecture

This chapter explains the architecture of the software in detail. From the very first beginning it was one of the main goals to achive high modularity so that individual parts can be enhanced or replaced easily. The software is seperated into the framework and the application part which in turn consist of multiple modules. Figure 4.1 shows the architecture.

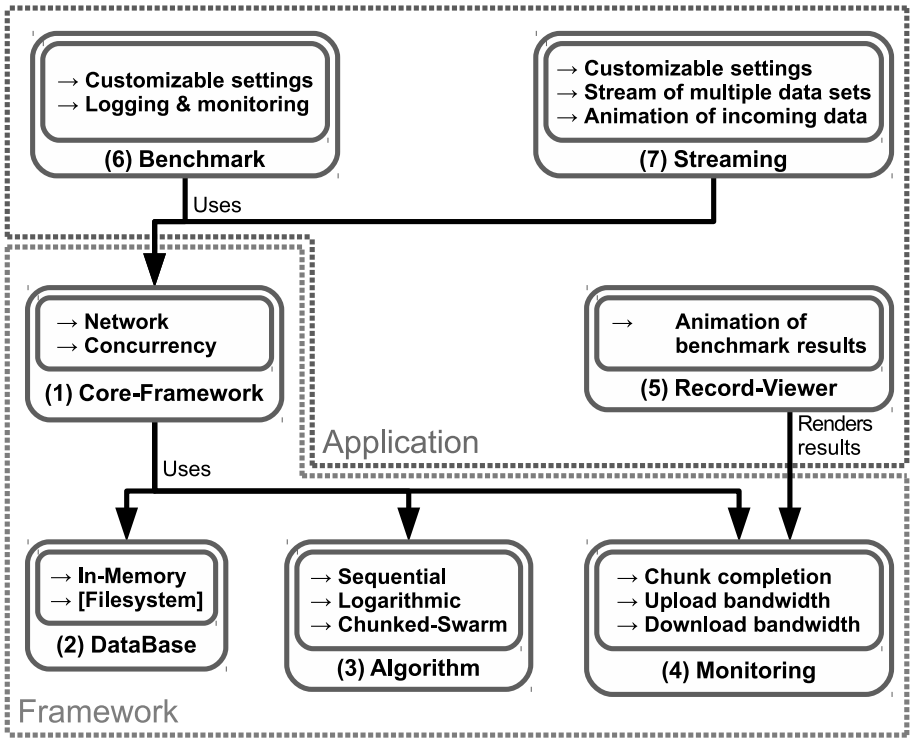


Figure 4.1: Software architecture

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 filesystem or even completely in-memory. The Algorithm module contains the implementations of the concepts presented in chapter 3, which include a sequential, logarithmic and chunked-swarm algorithm. 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 points for evaluating 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 happend 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 and after this 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 has the concept of leechers and seeders. A leecher connects to one or more seeders and requests chunks from them. A seeder has multiple leechers connected to it and answers to chunk requests. Every chunk transfer is pull-based, which means it can only be requested by the leecher while every non-chunk transfer like Meta-Data or address announcements is always push-based, which means that a seeder transfers them without being asked to do so. This has the advantage that 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 chunks in its database and a seeder uploads chunks from its database. It is possible that both a leecher and a seeder reference the same database instance in which case the seeder is able to upload every downloaded chunk from the leecher. The DataBase module and all related concepts are explained in section 5.2.

A leecher and a seeder both are associated with one of the multiple distribution algorithms defined in the Algorithm module. Those algorithms determine the distribution behaviour, which is further described in section 5.3.

5.1.1.1 Automatic Connect

While every leecher has to manually connect to a seeder for the first time, after this a leecher is able to automatically connect to other seeders. This is because every leecher who is coupled with a seeder announces the address of its coupled seeder to all other seeders it is currently connected with. The figure 5.1 show this in detail.

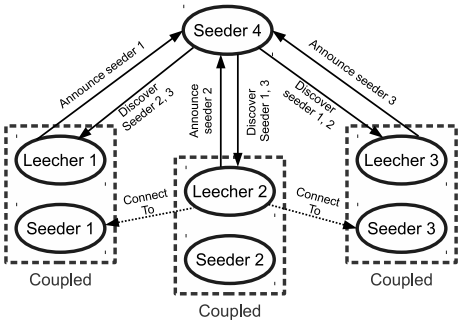


Figure 5.1: Seeder Discovery

Here leecher 1, 2 and 3 announce the address of seeder 1, 2 and 3 respectively to seeder 4 which then broadcasts those addresses to all connected leechers. This way the leechers get to know each other and can connect to the remaining seeders. Though in figure 5.1 only leecher 2 is illustrated, leecher 1 and 3 connect to the other seeders as well. Of course, this works recursively, so if one of the seeders 1, 2 or 3 already has other leechers connected to it those will also be found. So in the end the topology is just a mesh with $(n-1)*n$ 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. The figure 5.2 shows this from the point of view of leecher 1, which is connected to seeder 2, 3 and 4.

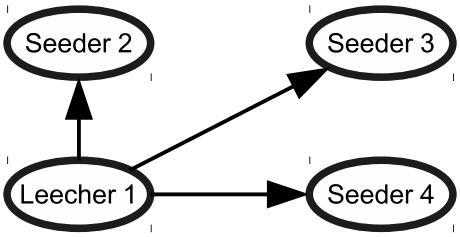


Figure 5.2: Leecher/Seeder mesh

While this topology is really great in terms of knowledge where every peer knows exactly what data sets the other peers have and thus can request chunks efficiently, it is also quite expensive and would not scale indefinitely. To improve scalability and reduce efficiency the number of connections per leecher can be limited which 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 what ever 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, for instance, leecher 1 lost the connection to seeder 2 but leecher 1 and 2 are also connected to seeder 3 then seeder 3 will notify leecher 1 that seeder 2 still exists. The figure 5.3 illustrates this.

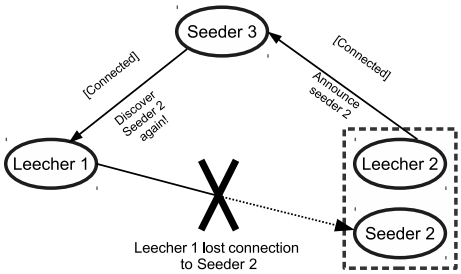


Figure 5.3: Reconnect

5.1.1.2 Meta-Data Announcements

After a leecher connects to a seeder the leecher does not know anything about the seeder. As previously explained in section 5.1.1.1 the leecher discovers new seeders through the seeders it is currently connected to. The way the leecher gets knowlege about what data sets a seeder offers works a similiar way. A seeder announces to all connected leechers periodically what data sets it has to offer. It basically transfers a list of Meta-Data, which is explained further in section 5.2.0.3, so that every leecher can update its knowledge about the seeder. The list of Meta-Data can be modified before it is announced by the used distribution algorithm to model a specific distribution behaviour, see section 5.3 for further information.

As an optimization the seeder only transfers new Meta-Data. So if a data set does not change at all, a seeder will never transfer a second announcement of this data set. Another optimization is that while a leecher downloads a chunk from a seeder, the seeder will also not transfer any announcements during this period, which is explained further in the next section 5.1.1.3.

5.1.1.3 Download Requests

After a leecher successfully connected to one or more seeders and 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 a seeder is not faster than downloading them sequentially only one download at a time is allowed which also reduces protocol complexity. Because of that Meta-Data and address announcements are also not transferred during a chunk download because those announcements are then obviously of no interest for the leecher. As soon as the chunk download completes outstanding announcements are bundled and transferred.

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

5.1.1.4 Nonblocking I/O

Network communication is based on sockets. Sockets can either be datagram or stream based. Datagram sockets have no notion of connections and are of no importance for this thesis. Instead the network implementation uses stream sockets. A stream socket is always connected to an other socket. Those two sockets behave like files. If you write data to a socket it is transferred to the other socket and can then be read from it. As with files, socket operations usually block the currently executing thread until the requested data is read or written. This implies that you basically need one thread per socket to handle multiple sockets concurrently. Unfortunately threads are a limited resource and thus this concept does not scale well.

To improve this situation nonblocking sockets in combination with event-polling were introduced, where you are notified when a socket operation can be executed without blocking. This way you can theoretically serve tens of thousands of sockets in a single thread. Since this thesis concentrates on large-scale this technique is a must-have.

The problem with nonblocking sockets is that it increases the complexity significantly so the network implementation uses a framework to simplify the usage. The framework is called Netty 5 and written in Java. One great thing about Netty is that it totally separates the transport protocol oriented code from the logic oriented one. This way it does not matter which transport protocol you use as long as this protocol is stream based.

5.1.1.5 Transport: Local and TCP

Netty offers a Local (also called In-VM) 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. That is because the Local transport does not require you to serialize and deserialize the data which eliminates a huge amount of memory and CPU usage. In normal usage the overhead of serialization is negligible but since the Benchmark module is able to simulate a large number of seeders and leechers, see section 6.1, those overheads can start to sum up. The Local transport seems to be the perfect fit for this purpose.

Netty also offers usual transport protocols like TCP. It is used as secondary option for the Benchmark module which is useful if you want to run distributed benchmark on multiple machines since Local transport is limited to a single JVM instance.

5.1.1.6 Traffic-Shaping

To simulate and evaluate network applications you need the ability to control the bandwidth in some way to have comparable network conditions. The problem with bandwidth limitation, which is also called traffic-shaping, is that most times you cannot throttle the writer. The writer can either be writing as fast as possible or not writing at all. So in order to limit the bandwidth you have to start and stop the writer periodically. Because 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 period is too long the introduced bandwidth peaks can affect the measurements in a negative way but if the period is too short instead, you introduce a lot of overhead. To perfectly limit the bandwidth you would need to use a period which is indefinitely short. So if you want to implement traffic-shaping reliably, you must find the sweet spot between accuracy and minimal overhead. The network implementation uses a period of 250 ms which is good enough because the measurements are done every second. So every measurement contains the average of approximately 4 bandwidth limitation runs.

The complexity increases if multiple writers together are not allowed to exceed a given bandwidth. The main problem is fairness. You do not want that one writer gets 95% of the bandwidth while the remaining writes get only 5% together. But at the same time you want that a single writer is able to write at full speed if no other writer is writing. This complex behaviour can be achieved using a Leaky-Bucket algorithm and a Priority-Queue.

The Leaky-Bucket contains tokens, where one token equals one byte. If a writer wants to write a

message it tries to remove the tokens needed for the message to transfer. If the bucket does not have enough tokens, the writer takes the remaining tokens and decreases the size of the message it want to write by the number of removed tokens. After this it waits until the bucket gets refilled and repeats the action. 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 request using the Leaky-Bucket. The difference is that after writing the message the writer increases its number of total written bytes. If the writer has more messages to write it inserts 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 least. This way you 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. If your CPU has 4 cores and you are using only one thread then you cannot use more than 25% of your CPU capacity. Because of that 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. Those events are mostly readiness events which say that 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 tries to rearrange the sockets to become balanced again.

Those 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 when using multiple threads is the question of how those 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 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. To explain those concepts in detail 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 threads and thus are not protected against concurrent access by default. Since this collection is a very common task it should be optimized as well. Using CAS and volatile operations the necessary mutexes can be reduced to a minimum.

In normal situations the overhead of mutexes can be ignored, but when simulating hundreds of leechers and seeders on multiple cores this 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. Those 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.

5.2 Module: DataBase

The DataBase module represents a generic interface for storing binary data in any kind of storage. While the Core-Framework transfers data between peers the data has to be read from and written to some kind of database. The data managed by the database is not just plain data but chunked data instead. Because of that a database stores so called data sets which consist of multiple chunks.

The database has the responsibility to verify stored chunks in terms of consistency and validation. But to do so the storage has to know about the structure of the data set. Because of that 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. Those information are transferred between seeders and leechers to describe what data sets are avaiable for download. Those information are:

- Name
- Description
- ID
- Size
- Chunks
- Hash
- ChunkHashes

The Name and Description fields are used for human-related tasks. The ID has to be globally unique and 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 strong hash algorithm like SHA from the whole data set. The ChunkHashes field does the same but for each chunk individually. This way the storage 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 you generate or consume data sets on the fly like video streaming or test scenarios.

In case of streaming you could simply add fake data set sources which loads chunks from a live camera or similiar devices.

The Benchmark module uses this backend to reduce overhead introduced by the filesystem. This backend almost has no overhead other than memory usage.

Related to future work is the filesystem backend which implements a persistent database based on a filesystem. The 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 logic part of the implementation. 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)
}
```

Those interfaces are totally independend of the underlying network implementation. The core concept is, that 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 SuperSeederChunkedSwarm algorithm, which is explained in section 5.3.0.8. In the next sections all implemented distribution aögorithms are compared and explained in detail.

5.3.0.5 Algorithm: Sequential

5.3.0.6 Algorithm: Logarithmic

5.3.0.7 Algorithm: ChunkedSwarm

5.3.0.8 Algorithm: SuperSeederChunkedSwarm

5.4 Module: Monitoring

Chapter 6

Application Modules

6.1 Module: Benchmark

6.2 Module: Record-Viewer

6.3 Module: Streaming

Chapter 7

Evaluation

Chapter 8

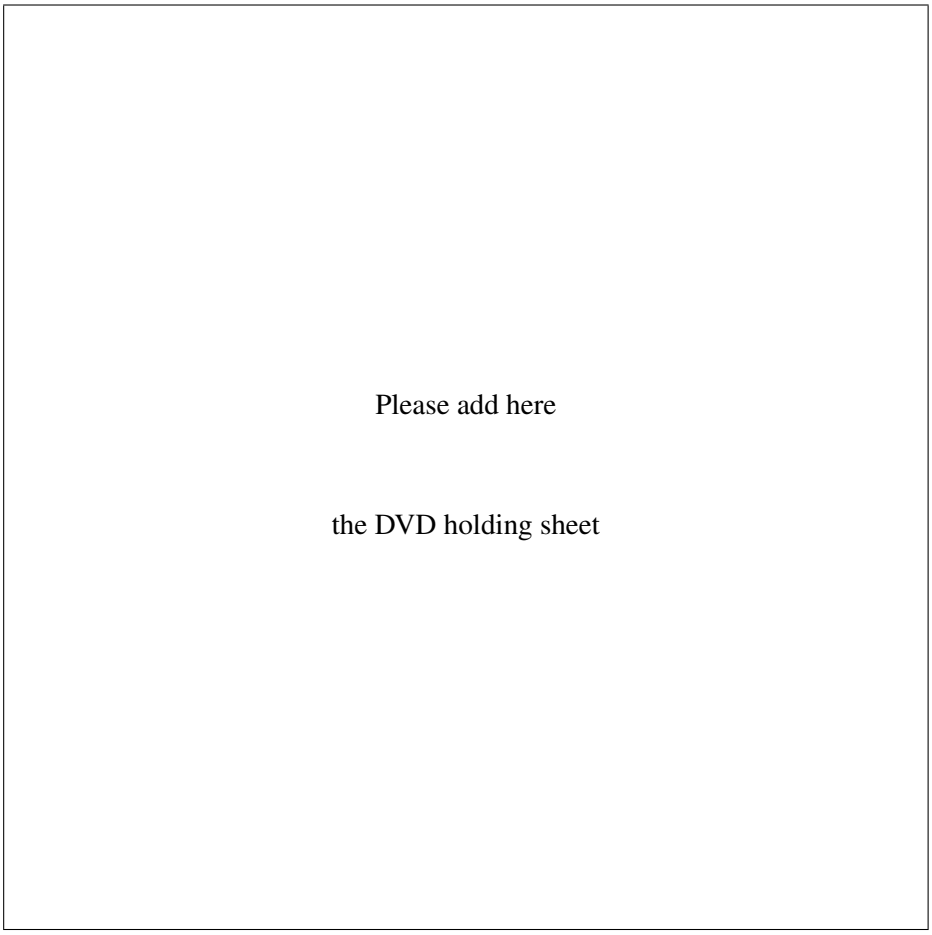
Future work

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