DPS assignment 2

Preon, a peer-to-peer volunteer computing platform

Elgar van der Zande
s1485873

Luc de Jonckheere
s1685538

1 Introduction

For this assignment, we decided to create a volunteer computing network. Volunteer computing is a type of distributed computing were people can voluntarily offer their computers' resources to help others with computations. Normally, these platforms employ a client-server model. A famous example of such system is BOINC. In a client-server architecture, only the server can create jobs to be executed. Also, the server may become a bottleneck, as all clients have to interact with it. By employing a peer-to-peer architecture, we do not need a centralized server and it will allow for any peer to start a job.

2 Design

The system we will design has to be able to create jobs. Such a job will consist of files which can, together with an execution command, generate answers in the form of other files. These jobs have to be shared on a network of peers. Also, the generated files should be able to be uploaded to other peers. To control jobs, we will design a command line interface tool. The tool should be easy to use, especially for people who only volunteer their computer power. This means that job sharing and execution should all be performed without user intervention.

Peers cannot find other peers on their own, so some sort of tracker server is required. As this server only handles very small and simple requests and is not involved in sharing and executing jobs, it is not likely to become a bottleneck. The tracker only has to keep track of which peers are working on a job. When a peer wants to join a job, it receives the IP addresses of the other peers from the tracker and has to communicate with the other peers to start working on it. This minimizes the trackers involvement.

A job needs a master. Usually, this is the peer who initially started the job. Because the tracker only does so little, the master is responsible for recruiting peers for a job. Peers who are looking for a job work on the virtual job "idle" so that the master can identify them. From the trackers' perspective, the idle job is the same as any other job. As peers cannot distinguish normal peers from the master peer, the master peer will actively have to try to download the answers from other peers. Our design allows for multiple masters in one job, so multiple peers can download the answers.

By splitting files up in blocks, we can concurrently download blocks of one file from multiple peers. This allows for a torrent-like file sharing. Such a design is necessary as most peers will have slow upload speeds compared to download speed. With this design, multiple slow upload speeds can be aggregated to achieve a decent download speed at one peer.

As with any network, faults are bound to occur. For instance, network packets may not arrive correctly or something may go wrong during the writing of blocks. Because of this, hashes of all files are also shared.

A peer may drop out at any moment without prior notice. This should not only keep the network in a consistent state, but also the job status on disk. For this reason, job files are verified at startup. Also, the status updates are written to a file after a block update is written to disk. This process helps maintain a consistent state.

On top of this, there may also be malicious actors which intentionally try to spread incorrect files. As the answer files are not known a priori, a possible solution would be to have multiple workers calculate the same job. Then, the answers of the different peers can be compared. By increasing the number of redundant workers, the confidence in the answer can be increased. Instead of uploading the full results multiple times, we can require one peer to upload the file and others to only send a checksum, which minimizes network usage. If verifying the answer is trivial, such as in NP problems, only a single worker needs to execute the job, as the master can verify the result itself. All of these design ideas can be summarized in a list of functional and non-functional requirements.

Functional requirements:

- Create new jobs
- Sharing and executing jobs
- Upload answers of a job
- Command line interface tool to interact with jobs

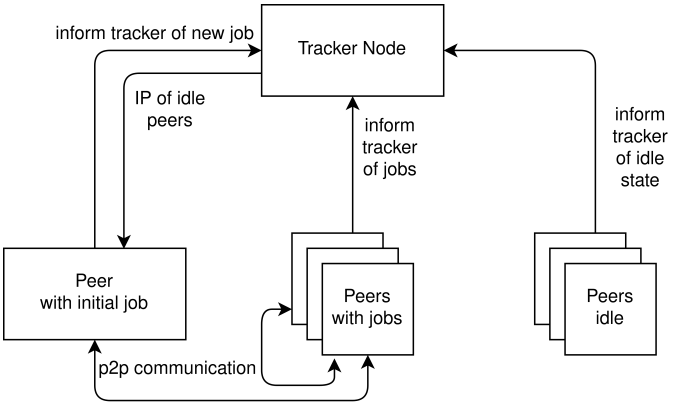

Figure 1: Diagram of interaction between different actors

Non-functional requirements:

- Peer-to-peer network
- Tracker server
- Automatically download, execute and upload answer of jobs
- Scalability with number of peers (master and tracker)
- File transfer in blocks
- Fault checking
- Sandboxed job execution
- Answer verification

3 Implementation

In order to test if such a system is feasible, we implemented a prototype called Preon. The prototype is developed in C++ and is designed to run on a Linux system. It has no dependencies on other libraries. All the files used for this project can be found at `https://github.com/lucmans/dps_ass2`.

Unfortunately, due to time constrains, we could not implement every feature we designed. However, because we designed our project with these features in mind, they could still be added easily. It is not possible to download multiple block from different peers concurrently. Also, letting multiple peers compute the same answer and comparing these answers is not yet implemented.

Our prototype consists of two parts. A tracker and peers. When a peer starts, it notifies the tracker it is idle. Peers who have a job which needs to be computed can request any of the idle workers to work for him. When a peer starts to work on a job, it notifies the tracker of this and the peer receives a list of other peers which are also working on this job. With this information, it can start working on the job with the others. All of this is summarized in Figure 1.

3.1 Job

Preon executes jobs, which are defined by manifest files. The manifest contains all information needed to download and execute a job and to upload the results. A job can be created using the command line interface of Preon (use `./preon -h` for more information). This will generate a job directory with a manifest. The job directory is named after its job ID, which is the SHA256 hash of `manifest.txt`. This prevents tempering with the job content and also generates a practically unique ID. The manifest has three sections. A section which lists all files to run a job, indicated by "[files]" in the manifest. An entry in this section consists of the filename, size and the SHA256 hash separated by spaces. A section which contains the execution command for a job, indicated by "[exec]". This can only be one command. It is customary, to provide a shell script with a job which starts a job. Using a shell script, more than one command can be executed. Lastly is the section where output files are defined which is indicated by "[dynamic]". Entries in this section only consist of a filename, as the hash and size may not be known.

The status of a job is saved in a file called `status.txt`. This file contains information such as the download state of every block and execution status of the job. The master status is also saved in this file, which will be discussed later. Only after a block has been written to disk, this file will be updated. This way, when a peer unexpectedly shuts down, the block will at worst be re-downloaded.

3.2 Communication

Peers and the tracker communicate using a simple protocol. The protocol is implemented on top of TCP. The body of a message is plain text. A message starts with its message type. For instance, a query message starts with "QUERY". Any arguments have to be space separated and a message has to end with a newline. The different messages will be explained in their respective section. The tracker always responds to a message so that a peer knows its message was handled. This can be the requested information, or simply "OK" if there is no information to send.

3.3 Tracker

The tracker only does one thing, keeping track of which peers are working on which jobs. It does this by creating a map in RAM which maps a job ID to a set of peers. A peer is defined by its IP address and port on which it is listening. The information relating to a peer is managed by the peer itself. Joining a job is done through inform messages and leaving a job is done through delete messages. Both these messages require the port on which a peer is listening and a job ID as arguments. The query message is used to get a list of all peers working on a specific job. A query message

requires a job ID as an argument. In response, it sends a newline separated list of all peers working on that job, were every peer is described as an IP address and port separated by a space.

There is one special job, the "idle" job. All peers start "working" on this idle job. Using this job, other peers can find peers which are looking for a job.

All code to run a tracker is in the directory `tracker`. Communication with the tracker is abstracted for a peer through the Tracker class (`preon/tracker.h`).

3.4 Peer

A peer has several functions it has to perform. It has to listen to other peers and respond to them, it has to download jobs and execute them and it has to check if new jobs are added. Every function and job run on their own thread. This way, different functions do not block each other and multiple jobs can download at the same time.

To check if new jobs are added, the job download directory is periodically checked for new jobs. If a new directory is found, a new job will be added to the job list.

The main function of a peer is downloading and executing jobs. If a peer wants to download the files containing the job results, it has to assign itself the master role. Other than downloading the results, the master is responsible for recruiting idle peers. As mentioned before, we choose this design to minimize the involvement of a central server. However, this does mean that if the master goes offline, no new workers are recruited.

Working on a job consists of a few steps. First, a peer checks if there is a manifest file. If it is missing, it is downloaded. Then, the manifest file is verified and parsed. If a peer is a master, it will try to download the meta data of dynamic files from other peers. This blocks until a peer is done executing its job. Using the obtained information, we can start downloading blocks from other peers. We do this until every file is downloaded and verified. Non-master peers execute the downloaded job instead of downloading the results.

A peer can be configured to run a specific number of job threads. In every thread, a job can run, which allows for multiple jobs to run concurrently. By changing the number of job threads, the user can choose how much compute power they volunteer to the network.

In order to balance the load between peers, we download each block from a random peer. This will eventually even out the requests between all peers. We also request a random block from every peer. This is to prevent many peers requesting the same block at once, which would happen if many peers join a job and all would start with downloading the first block.

4 Experiments

To test if our system has the key features of a distributed system, we ran some experiments. We will be showing that the system is scalable and that it performs well. All experiments were run on the DAS-5 cluster.

4.1 Scalability - Peer scaling

When adding more peers to the system, everything becomes more complex and individual executing times may go up. To test the overhead of adding extra peers, we let an increasing number of peers work in a job pool. This job pool will consist of 256 jobs. Each peer will have 16 worker threads and the job pool will have one master. The job will compute the Mandelbrot fractal using our Mandelbrot tool. The settings are 4096 iterations and a 4096 by 4096 resolution. Every test is performed three times. The time to finish all jobs is compared to running the program locally a total of 256 times with 32 threads at the time. This is then divided by the number of peers for each point. This number serves as a lower bound on the computation time. The average computation time of the local operation is 506.9 seconds with a standard deviation of 3.45 seconds.

The results are shown in Figure 2. Computing a job on our network is much slower than expected. The job did have to copy 65 MiB of data every job, but this should only take a second. However, because there many jobs running on the same network. As all communication goes through this network, the 65 MiB has to be send 256 times. This equates to about 16 GiB, which would take proximately 250 seconds. If we add this time to the local computations, we see the results lie much closer together. This shows the network is built for longer jobs, as with longer jobs, the execution time becomes significant compared to the file transfer time. Also, the total time of the jobs does not scale perfectly $\frac{1}{n}$. This is because there is some extra overhead in managing extra peers.

4.2 Scalability - Tracker bottleneck

The only central point of communication is the tracker. Since a peer can only execute a job after communication with a tracker, a peer should never have to wait on the tracker.

To estimate how many clients a tracker could handle worst case, we run as many jobs on as many nodes as possible. By measuring the time spend on handling requests and measuring the time spend waiting on requests in the tracker, we can estimate the load on the tracker. To accurately measure the tracker, we designed a job which has minimal download/execution time. This will maximize the interaction with the tracker, as communication with the tracker occurs before starting and after completing a

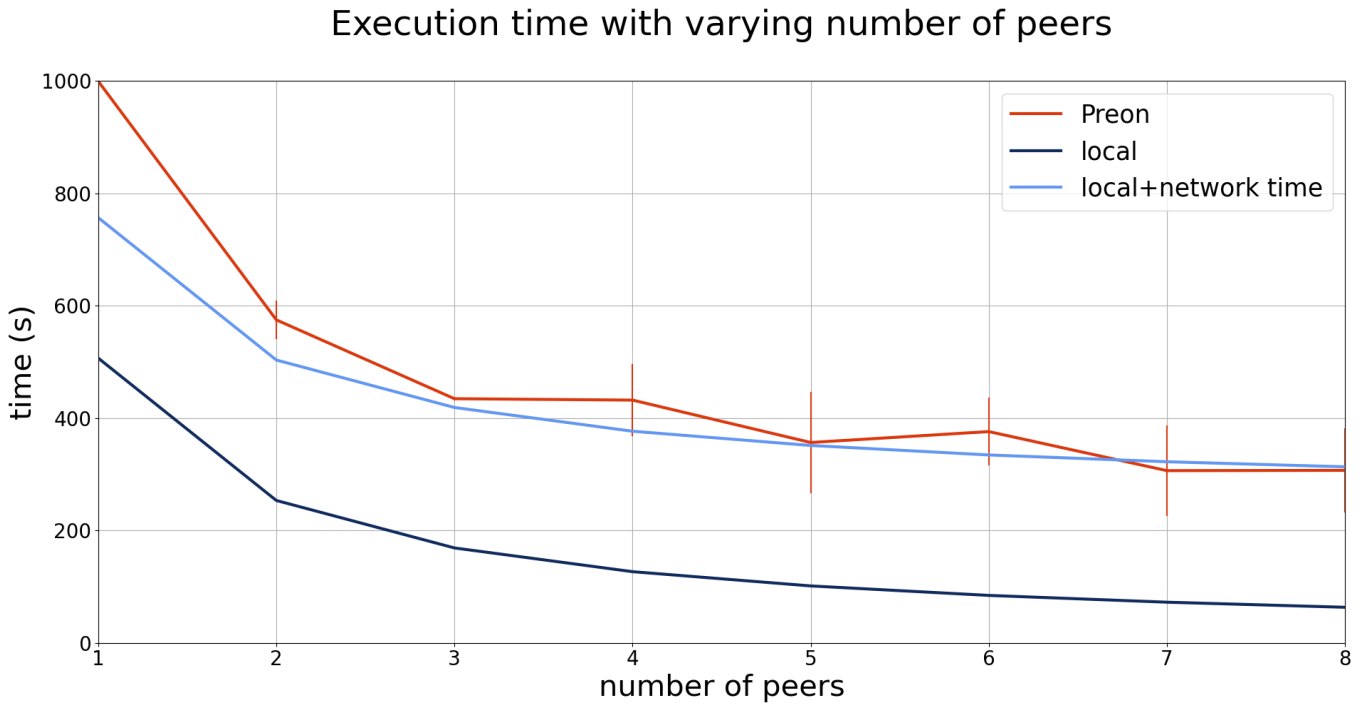

Figure 2: The total time to complete 256 jobs and upload the results back to the master for a varying number of peers in the network.

job. The job consist of one file, a shell script which only contains the line "echo hello".

We will run one master and 16 normal peers. Each peer will run 32 worker threads. The master will run with 256 worker threads. In total, 2048 echo jobs will be created.

In Figure 3, we can see the tracker usage over time as a percentage of its maximum usage. The usage is determined by dividing the time spent working by the total time between two requests (working time plus waiting on request time). As the tracker usage fluctuates too much to plot, a rolling average of the last 25 datapoints is plotted instead.

In the first 15 seconds, the master has not yet noticed all the new jobs. This is because the jobs were still being generated and the filesystem watcher only checks the filesystem once per second. The tracker traffic comes from peers notifying they are idle. This kind of traffic is always present, however, we did not optimize this traffic much. After the first 15 seconds, the jobs start being noticed and are executed rapidly. The spikes in the graph are because jobs are discovered in batches. This results in much traffic as they all get started together. However, handling 2048 jobs in 20 seconds is a very high amount, especially given that normal jobs take much longer so do not have to start as often. Also, the tracker server is single threaded at the moment. Handling each request on a different thread will improve the tracker throughput significantly, as most time is spent waiting on blocked network calls. After 36 seconds, the last jobs have started. The rest of the requests are peers going back to idle again.

4.3 Performance - Job distribution

The program has some overhead in sending out jobs to other peers. To test how quickly jobs are distributed, we let the master start 200 jobs. Then, we let ten peers work on twenty jobs at once. The echo job from the previous experiment was used again. This experiment was performed ten times. The average amount of time it took for a job to finish being executed by a peer after being dispatched by the master was 10.07 milliseconds with a standard deviation of 2.93 milliseconds. In Figure 4, we can see the duration of each job. The jobs are sorted on start time, so a point left of another point was dispatched earlier. In the graph, we can see that the job time fluctuates after dispatching many jobs, however, it stabilizes quickly. This is because at the start, a peer may accept too many jobs if too many are started at once in a network. To finish the last job after dispatching the first takes on average 1.96 seconds with a standard deviation of 0.03 seconds.

The delay between dispatching a job and it being executed are plenty fast for our network, as most jobs will have a runtime significantly larger than this delay. Thus the runtime will overshadow the distribution times.

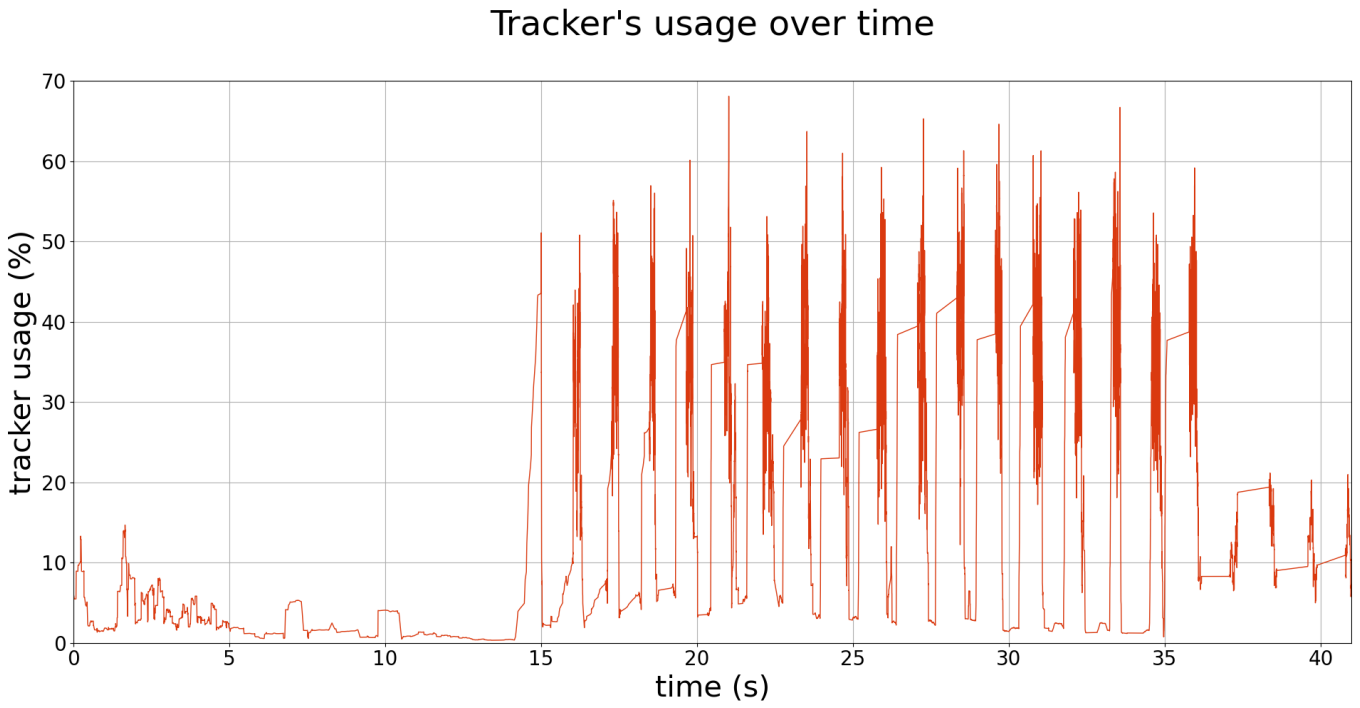

Figure 3: Usage is defined by the percentage of time spend working between two requests. After 15 seconds, the master dispatches the first jobs.

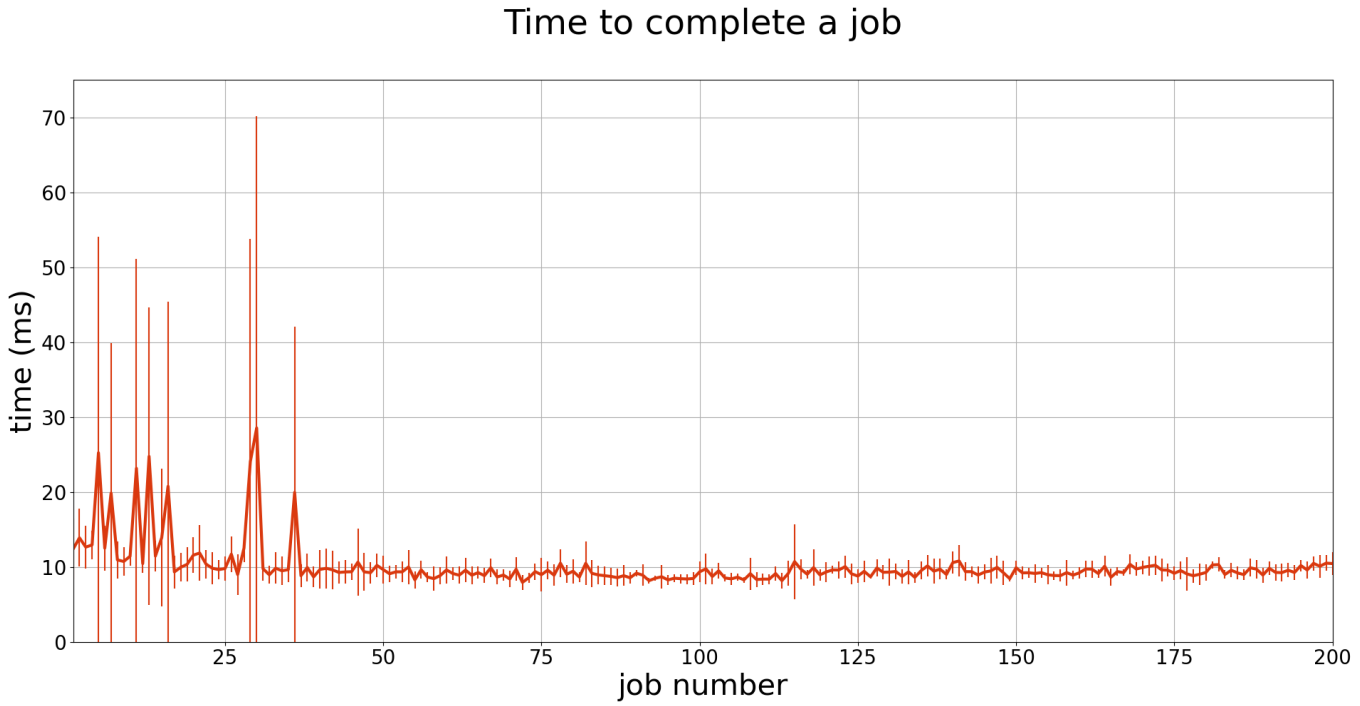

Figure 4: The time from dispatching a job until worker has finished execution. The job number indicates the start order of a job.

4.4 Performance - File transfer

To test the overhead of our file transfer algorithm, we copy over a large file (1 GiB) through Preon and through rsync. We choose rsync as it is often used for fast file copying over a network connection. Every transfer was performed ten times. The average results and standard deviations are shown in Table 1. Also, the calculated transfer speed is noted. The theoretic limit is based on the network bandwidth between two nodes in the DAS-5 cluster. Note that compression was explicitly disabled in rsync as Preon does not compress blocks. With compression enabled, rsync performed about 0.1-0.2 seconds faster. The improvement is likely this small because random data cannot be compressed much.

Program	average	st. dev.	speed
Preon	13.95 s	0.274 s	73.42 MiB/s
rsync	9.768 s	0.037 s	104.8 MiB/s
Theoretic	8 s	-	125 MiB/s

Table 1: Transfer time and speed

The table shows that rsync performs about 1.5 times better than Preon. This is, however, for a 1-to-1 download. Preon's block transfer could share multiple blocks concurrently, which is not implemented yet. Preon can only obtain a higher speed than rsync if there are multiple nodes which downloaded from a single master. In this case, Preon can also start downloading from the other nodes, hence alleviating the network usage from the original source peer.

5 Conclusion

In this assignment, we have created a peer-to-peer volunteer computing platform called Preon. By using a peer-to-peer design, we can alleviate the bottlenecks that normally occur in similar systems with a client-server model. Preon consists of two programs, a peer and tracker, which together can distribute and compute jobs. We have made our own protocol for the communication. This protocol allows files to be downloaded in blocks, so a file can be downloaded from multiple peers at the same time. The integrity of all files is verified by comparing the checksum of the file to the checksum in the manifest. The checksum for the manifest is shared through the job ID. Moreover, the tracker is designed to have little involvement in the whole process, so it is not likely to become a bottleneck.

Our experiments show that Preon satisfies distributed system requirements such as scalability and performance. The trackers is fast enough to handle 512 workers. At this workload, the tracker was only working less than 50% of the time. Job do have a significant overhead compared to computing it locally. However, given enough peers, job execution times do become faster than doing it all locally. Jobs are quickly distributed within the network and file transfer speeds are sufficient, especially considering it could gain a significant speedup by allowing downloading downloads from multiple peers concurrently. The network can handle sudden failures of peers.

Most of the requirement were met in the end. For the functional requirements, we can create, share and execute jobs. Afterwards, the answers can be uploaded back. There is a CLI tool to manage all of this.
For the non-functional requirement, we did not manage to fulfill all. We do have a peer-to-peer network with a tracker server. When starting the application, it automatically starts working on jobs. From a peers perfective, the results are also automatically uploaded. Both the tracker and the master seem scalable according to our tests. The file transfer is done in blocks, however, concurrent block downloads is not yet possible. Every file is checked, so faults are detected. We unfortunately do not yet execute a job in a sandboxed environment, so malicious jobs can interfere with someones computer. The answers are also not verified yet by letting multiple peers compute the same job.

6 Future work

In this report, we have mentioned some things which can be improved or which we did not have the time for to finish. We will summarize them here.

The tracker runs entirely on one thread. The performance could be increased by handling multiple requests on different threads. This could greatly increase the performance of the trackers, especially when it tracks many different jobs.

We planned on answer verification by recruiting multiple peers for the same job to prevent malicious actors from uploading incorrect results. We can then compare the results from the individual peers to gain confidence in the validity of the answer. The larger the answer redundancy, the more confident we become.

Moreover, blocks are currently only downloaded from one peer at the time. This does already improve on downloading all blocks from the master. Now, when a few peers have a job, the total upload speed of a job on the network could be the aggregate upload speed of all these peers, but the download speed is still limited by an individual upload speed. However, when we can download from multiple peers at once, we can achieve the download speed of the aggregated upload speed of individual peers.

Something we have not mentioned before is job dependencies. Often, multiple jobs share the same data. By introducing job dependencies, we could make one job with all data and make it a dependency of many other differing

jobs which use this data. This prevent much data from being send multiple times over the network.

Appendix: Work distribution

The following table shows how we ended up dividing the work over the both of us. Note that most working time was spend together by using VoIP and we divided all tasks by just doing something that needed to be done, so the table is only an approximation.

Elgar		**Luc**	
Task	Time	Task	Time
Writing peer	9 days	Writing peer	5 day
Writing tracker	1 day	Write misc software	1 day
Proofreading report	1 day	Write report	2 days
		Perform experiments	2 days