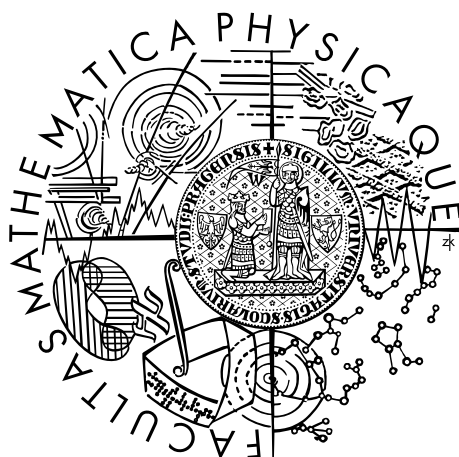Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Vojtěch Hudeček

Distributed video compression in the peer to peer network

Department of Distributed and Depandable Systems

Supervisor of the bachelor thesis: JUDr., Mgr. Antonín Steinhauser

Study programme: Computer Science

Study branch: General Computer Science

Prague 2015

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....   signature of the author

Název práce: Distribuovaná komprese videa v peer to peer sítích

Autor: Vojtěch Hudeček

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí: JUDr., Mgr. Antonín Steinhauser, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Navzdory vysokému a stále rostoucímu výkonu dnešních počítačů se stále setkáváme s úkoly, které jsou velmi časově náročné. Některé z nich lze rozdělit na menší podúkoly, které mohou být zpracovány paralelně. Běžnou technikou je rozdělení práce mezi více počítačů za účelem zrychlení celého procesu. Většinou se nicméně setkáváme s přístupy, které jsou založeny na architekturách typu klient-server. V této práci představujeme čistě peer to peer řešení, které umožňuje škálovatelnost, zotavení z chyb a snadné spravování. V našem frameworku není vyčleněna žádná speciální role a kterýkoli výpočetní uzel se může kdykoli připojit nebo odpojit. Systém se také dokáže vyrovnat se selháními uzlů při zachování dobrého výpočetního času. Testování ukázalo, že v lokálních sítích můžeme dosáhnout několikanásobného zrychlení.

Klíčová slova: Paralelizace Peer to Peer Distribuované výpočty Kódování videa

Title: Distributed video compression in the peer to peer network

Author: Vojtěch Hudeček

Department: Department of Distributed and Dependable Systems

Supervisor: JUDr., Mgr. Antonín Steinhauser, Department of Distributed and Dependable Systems

Abstract: Despite today's computers' performance there still exist some tasks that are quite time demanding. Nature of some of these tasks allows to split them into smaller parts that can be processed in parallel. Distributing work among more computers in order to speed up such processes is a common technique. However, most of the approaches use client-server architecture to achieve this goal. We provide purely peer-to-peer solution which allows high level of scalability, error recovery and easy maintaining. No special role is needed in our framework and each node can join the network at any time. Also the system is able to deal with node failures, keeping the overall computation time reasonable. Tests showed that significant improvement can be achieved in local area networks.

Keywords: Parallelization Peer to Peer Distributed computing Video encoding

I would like to thank to Mr. Marek and Mr. Steinhauser for valuable advice and patient supervising. I would not be able to finish this work without them. My thanks also belong to my family and friends for support and patience, not only during the work on the thesis but during all my studies.

Contents

Introduction

Motivation

Although the Moore's law[1] still applies and the performance of computers is now greater than it ever was, there still exist some tasks which require a lot of computer power. Despite parallelism, their accomplishment takes a non-trivial amount of time. Among these tasks are for example computations of some special prime numbers, processing large data sets and many others.

It's natural to ask, how we can speed up such computations. Obvious possibility is to get better hardware, however this is limited by technical progress and can be quite expensive. Widely used approach is to parallelize the computation, that is different parts of the job are run concurrently (in parallel) which results in reduction of the processing time. Generally, tasks can be parallelized on different levels. We will talk about parallelism on the whole computers level, i.e. each computing node is represented by a standalone machine. These nodes are independent and their physical distance can be arbitrary big. The only condition which has to be fulfilled is, that the nodes are connected together and form a network. It means that they can communicate and exchange data. This approach makes sense only with sufficiently big tasks, because the network communication is relatively slow and for many tasks it would be a bottleneck. There are also different ways how the general term "parallelism" can be understood. We will consider so-called data parallelism. That is, each computing node runs the same program, only the data differs. This is the main difference from the task parallelism concept in which different nodes can run completely different tasks.

Obviously, not all tasks can be parallelized easily or even are not possible to parallelize at all. On the other hand, there exist also jobs that can be split into smaller parts, each of which can be processed independently and the results then can be joined together in the end. Concurrent processing of such jobs is quite straightforward and allows to use of the data parallelism. These jobs are suitable for our framework.

Goal of the thesis

The main goal is to implement a robust framework which should be able to split the given job and distribute the work among the participating nodes. Each node then processes the assigned parts and the results are eventually collected and joined together. The domain of suitable jobs is restricted to those that are described in the preceding paragraph.

The concept of parallelization has been known for many years and many different approaches have been introduced. Some of them are discussed in chapter 1. Nevertheless, the great majority of them use some kind of hierarchy or need some control node, whose functionality differs from the others. Our main motivation was to provide solution, in which all the nodes are equivalent. Furthermore, each can initiate the process or serve as a computing node. There also exists one node which controls the process while the rest is working. But the point is, that

---

[1] https://en.wikipedia.org/wiki/Moore's_law

every single node can become the initiator (master) or serve as a slave or even both at the same time. Our framework was implemented with the application for the video encoding in mind. However its essence is generic, so it could be easily modified to be used with an arbitrary job which fulfills the conditions mentioned above.

The purpose is to implement framework which conforms to the peer to peer paradigm as much as possible. We also want to keep the logic in the node which have demands to be processed in order to ensure better control and more deterministic behavior. Another important goal is to minimize occurrences of errors in the system and provide good ability to recover from them. The system is also intended to use personal computers as computing nodes, so the program typically runs alongside other applications.

Application to the video encoding

Video files are basically sequences of images, so-called frames. When the file is played, the images are showed sequentially one by one. They are changed many times per second. so the illusion of smooth video is made. There are many different ways how to code the video into the digital form. Because of this, the need arises to convert the files from one format to another. The program which codes the analog video data to digital or decodes it the other way around is called codec. Some of the well known codecs are for example *H.264*[2] or *MPEG-2*[3]. In order to convert the file, each frame has to be re-encoded, which takes quite a large amount of time. This makes the video encoding ideal task for our framework.

Before we use it, some more details have to be revealed. Because we usually want not only to watch the video but also listen to some sound, the video files are usually accompanied by one or more audio files. All of them are packed together in some container. When we want to convert the video, the container has to be opened, the video extracted, processed and stored again. The audio files are not our concern and we will simply copy them. Common container formats are *avi*[4] or *mkv*[5].

There is another important thing. In order to reduce file size and decrease the requirements of rendering the video, the video file actually contains only few frames which carry the whole information. These are usually referred to as the key frames. The rest of them contains only information how the frame differs from the previous one. This approach greatly reduces the file size and preserves good ability to recover from errors, e.g. when some part of the file is damaged. Consequence of this fact is, that we cannot split the video file at an arbitrary place. This is because if the chunk started with some non-key frame, the conversion would not be done properly.

Issues connected with video encoding are quite complex and used algorithms are very sophisticated and advanced. We did not reinvent the wheel and used third party software for the video encoding, namely the ffmpeg and ffprobe from

---

[2] https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC
[3] https://en.wikipedia.org/wiki/MPEG-2
[4] https://en.wikipedia.org/wiki/Audio_Video_Interleave
[5] https://en.wikipedia.org/wiki/Matroska

the FFmpeg[6] project. For the sake of simplicity, the program outputs always video files packed in the Matroska container. The choice of the codec is arbitrary, however it is restricted by the list of options. All of this software is free and published under the GNU General Public License[7] or GNU Lesser General Public License[8] in case of FFmpeg.

Terminology

To prevent misunderstandings there is a short list of frequently used terms:

- *master* (or *initiating node, initiator*) is a node which has job to be done, i.e. has file to be encoded. It initiates and controls the process.

- *task* (*job*) refers to work that should be done, specified by the initiator.

- *chunk* denotes one part of the split file, which is supposed to be encoded independently.

- *(computing) node* is one particular entity which communicates with the master and encodes chunks for it. Theoretically, there can be more such entities on one physical computer, because each node is determined by its communicating port and address.

- *neighbor* is called every node which is in the list of the node we are considering.

Thesis organization

In the first chapter is given overview of the existing technologies and approaches and the description of our framework. Second chapter describes implementation details. Third chapter introduces some experiments and summarizes the results. The fourth chapter is dedicated to alternatives and possible future improvements. In the appendix is then described installation and use of the program.

---

[6]`https://www.ffmpeg.org/`
[7]`https://en.wikipedia.org/wiki/GNU_General_Public_License`
[8]`https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License`

1. Analysis

1.1 Known approaches

The idea of distributed computing has been known since the 70's. Different approaches have been tried since then. Some of them are described in this section in order to show different concepts.

1.1.1 Client/server architecture

It's very common example of centralized system. In this pattern is one central node, known as the server. The rest of nodes, clients, are all connected to it. All the information is stored at the server. Also all the important computations happen there. The communication is typically initialized by the client, which sends a request. The server processes it and returns the desired result. Pros of this approach are relatively simple implementation and easy control of the traffic, because all the data passes through server. Also the data flow can be controlled easily. However the huge disadvantage is, that the server is by definition a single point of failure, i.e. if it goes down, the whole system becomes unusable. Also, thanks to the asymmetric nature of the system, there usually has to be more parts of the software, one to run on the server and the other to run on the clients. The system is dependent on the server's performance too.

1.1.2 Peer to peer architectures

Pure peer to peer network

Peer to peer paradigm is virtually the opposite of the previous model. Nodes are completely equivalent to each other. Any of them can join the network at any time, as well as leave it. This approach means great scalability and fault tolerance. On the other hand, it requires more complex design and can encounter problems with security and management of the network. Our framework implements this model.

Hybrid peer to peer network

This approach combines the previous two. There exists one node which is dedicated to serve as a control server. Each node contacts this server and all the control information pass through it while the data are exchanged in the same manner as in the pure peer to peer networks. This approach removes possible problems with the control of the data flow and provides information coherence while scalability is preserved. However, the issue with the single point of failure is reintroduced here.

Super peer architecture

Another try to improve the architecture introduces the concept of the so-called super peer. It is a computer, usually with slightly better performance than other

nodes, that plays the role of the control server, but only for a restricted group of nodes. Autonomous groups (clusters) are thus formed. It has the same advantages as the hybrid network but the potential super peer failure is not so crucial. Also each super peer can have backup so the robustness of the network can be very good.[1]

1.1.3 Summary

We showed different implementations of the idea of distributing the work. Each has its own specifics and may be suitable for some application. We have chosen to implement our framework to conform to the pure peer to peer paradigm as much as possible. Although the super peer architecture offers better control, we decided not to use it for several reasons. Mainly because our framework is supposed to be used in rather small networks where this paradigm could be quite exaggerated. Also we did not use the hybrid network because we definitely want to avoid presence of the single point of failure.

1.2 Existing solutions

Since encoding of the video files is quite reasonable task, there was a few implementations of the similar issue. Some of them served for academic purposes only, but others found good use case in practice.

1.2.1 DVE (Distributed Video Encoding)

D. Hughes and J. Walkerdine from the Lancaster university published in their paper a solution which is using Lancaster's P2P Application Framework. They implemented a Java plug-in for this framework which uses Microsoft Windows Media Encoder SDK[1]. Their approach was quite similar to ours and they achieved quite persuasive results. However their solution was usable in the specific environment only.[2]

1.2.2 Apache Hadoop

It is sophisticated distributed framework written for the Java platform. It serves for processing large amount of data and also as a storage. It provides very good ability to deal with hardware failures. The framework consists of the HDFS (Hadoop Distributed File System) and the processing part. The HDFS is a distributed file system which uses TCP/IP[2] sockets for communication between nodes. It uses replication of data to achieve reliability. It is optimized to store large immutable files (range of gigabytes or even terabytes). The framework could use different file system, however, one of its main advantages is the knowledge about the data locality, hence unnecessary data transfers can be avoided. This feature may not be available when using some different file system. The processing part is based on the MapReduce engine, based on functional programming concept. It basically consists of three steps.

---

[1] https://en.wikipedia.org/wiki/Windows_Media_Encoder
[2] https://en.wikipedia.org/wiki/Internet_protocol_suite

1. Map - A special function is applied to the local data on each worker node, data are associated with a key. This step can be processed in parallel.

2. Shuffle - The data are redistributed, according to keys. Each worker node obtains data sets with the same key

3. Reduce - The data are processed, in parallel.

A special master node is needed which ensures effective task scheduling. Generally, the framework is suitable for processing large data sets. It needs dedicated infrastructure and can achieve very good results. It can be used in many different applications including image processing, marketing analysis or data mining. It differs a lot from our solution - among its complexity and suitability for wide range of tasks, it needs some nodes with special functionality and also the network of specialized nodes. In contrast, our solution can operate in almost any computer network and the computing nodes can serve for other purposes.[3] [4]

1.2.3 BOINC (Berkeley Open Infrastructure for Network Computing)

It is a distributed system, developed on the Berkeley university, which utilizes computing power of personal computers around the world and interconnects them in the huge network. It supports multiple operating systems (the client part). Anyone can voluntarily join the network and scale it up by granting his computer's performance. After installing the client application, the system is able to exploit free CPU or GPU capacity of the computer. The server provides him with a portion of the given task. After the computation, the results are verified and uploaded. The system is used mainly for scientific computations. For example it is used by the well known project SETI@home. The framework is based on the client-server architecture. The server runs on UNIX OS[3], using common technologies such as the Apache web server and the MySQL database server. A PHP language is used.There are also CGI programs and daemons running on the server. The obvious difference is in use of the client-server model, nevertheless the philosophy of using regular computers as the computing nodes corresponds to ours. [5]

1.2.4 Gnutella

Gnutella is a peer to peer framework used for file sharing. The nodes use flood technique with limited number of hops for searching of the desired data. The positive acknowledgement message is sent when this data was found, employing the UDP protocol. The file transfer can be negotiated then. The system is based on the super peer architecture, which makes it possible to reduce maximum number of hops. Though it is not used for distributed computation, the data transfers are used which makes it comparable with our system.[6]

---

[3]`https://en.wikipedia.org/wiki/Unix`

1.2.5   BitTorrent protocol

Another interesting file sharing technique is introduced in the BitTorrent network. The files offered by the node are stored in special list and a single file download can (and usually do) use more nodes. This is possible, because the files are divided into pieces and when the node downloads some piece, it becomes source of this piece for the others. The pieces are marked so they can be downloaded in random order. The system provides good redundancy. Another advantage is, that as the file spreads through the network, the requirements on original distributor's bandwidth decrease. Eventually the node could not be needed at all. This protocol is used for file sharing by the BOINC framework, among many others. It is certainly an interesting alternative to handle the file transfers. In our framework, the use of this protocol would not be very convenient and useful, since once the chunk is processed, it becomes useless.[7]

1.3   Framework description

1.3.1   Basic overview

The heart of the program is one executable file, that should be accompanied by the configuration file. This is discussed in detail in appendix A. Main options that should be set are IP address and a number of port which the program should listen on. It is also essential to provide credentials (i.e. address and port) of some node which should be contacted by default. When more nodes are spawned, the network is formed and the computation may be initialized. Note that the information about neighbors spread in a nondeterministic manner, it matters who is contacted by whom. So it's good to have one or more nodes which are online most of the time and the rest contacts only these nodes initially. Otherwise the distribution of the neighborhood knowledge may be quite slow. Once the network is established, the computation can begin. If one (or more) nodes have tasks to be done, it can start the process. The file is then processed and divided into chunks in the initiating node. These chunks are distributed among neighbors, processed and returned back. Once the initiator has all chunks back, it joins them together and the process ends.

C++ language has been chosen for the implementation. It allows us to use standard POSIX socket [4] API (Application Programming Interface) which is widely used and has been proven by thousands of applications. Also it provides convenient functionality thanks to its Standard Template Library while allowing to use C library functions so it can cooperate well with the OS. Because it compiles to the native code, it does not need any interpreter, so its requirements are lower than other languages' such as Java and C#. This fact is positive, because it means, that if the application runs in background and computes nothing, it does not spend many system resources. The program is supporting Linux operating system, it has been tested on several Linux distributions. Nevertheless, it conforms to the POSIX standard [5], so after some modifications it could be used with any UNIX or UNIX-like operating system.

---

[4]`https://en.wikipedia.org/wiki/Berkeley_sockets`
[5]`https://en.wikipedia.org/wiki/POSIX#POSIX.1-2001`

1.3.2 Neighbor maintaining

The diagram which describes the process can be seen in the Figure 1.1. When the node wants to use other computer's computation capacity, it naturally has to contact them first. But the node has no prior explicit knowledge of the network's infrastructure, moreover, this infrastructure can change dynamically. To solve this, each node maintains the list of its neighbors. It contains addresses of the nodes which it has contacted successfully in the past. The list is refreshed periodically, so the node keeps track of the current network state. Besides this main list exists another list, which contains potential neighbors. Generally, each node that has communicated with the given one sometime in the past is added to the list of potential neighbors. The purpose of this list is to reduce the amount of time spent with maintaining the neighbors list. That is, no more than required count of neighbors is maintained, but if needed, suggestions can be found in this backup list.

Neighbors are uniquely identified by the pair consisting of address and communicating port. This is sufficient for the potential neighbors, because before the neighbor is added to the main list, it has to be contacted. Additional information is then added, so more complex structure is needed for storing neighbors. This structure contains information about neighbor's quality, last known state etc. The quality of the neighbor helps to prefer one neighbor to another while picking the one to contact. It is updated after each chunk is delivered from the given neighbor and reflects the neighbors computation power together with the speed of the connection.

Initialization and discovery

This may be the most important part of the process. Its overview is given in Figure 1.1. The node is provided with the address and port of the neighbor which should be contacted by default, that is, when the neighbors list is empty. Each node has minimum count of neighbors which should be in its list. This number is checked periodically. In case that the count of confirmed neighbors is too small, the list of potential neighbors is checked. If it's empty, the node has to obtain more neighbors so it picks one of its neighbors, contacts it and receives some suggestions. If it has no neighbors, the default node is contacted. Once the suggestions are received, the node adds the new addresses to the list of potential neighbors. When the list contains some potential neighbors, the node can contact them, confirm, and add as regular neighbors.

Gathering neighbors

When the initiating node has not enough neighbors and it needs more neighbors with free computation power, it can use another mechanism to collect them. This mechanism uses a flood technique to spread the request among the nodes in the network. The request is send to each neighbor, which spreads it further in the same manner. However, the request is equipped with time to live value that is decreased after each hop. Because of this, it does not spread forever nor too far. Once the node receives a request and is not busy, it contacts the initiator directly.
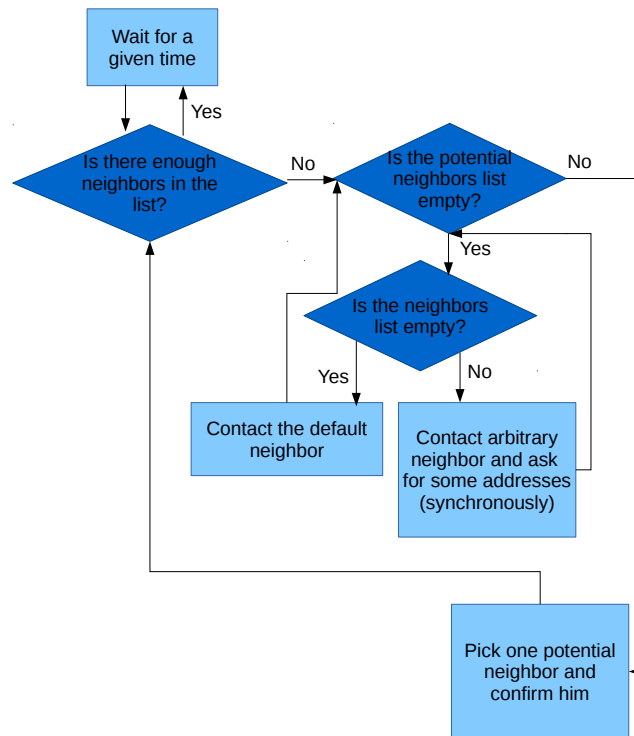
Figure 1.1: Maintaining the neighbors

It can then add him to the list of potential neighbors and later possibly as the regular neighbor.

Withdrawing from the network

When the node wants to leave the network it has to abort the process, if it is the initiator. Then a special message is sent to each neighbor, which informs them so they can react accordingly. That is, if the neighbor has tasks to be processed for the leaving one, it removes them from the queue and then removes the neighbor itself. Removed neighbors are completely lost, they are no longer stored in either of the lists. The process of saying goodbye to neighbors is asynchronous, the withdrawing node does not wait for any confirmation. There are several reasons for this behavior. Waiting for the response could cause a deadlock under some specific circumstances, e.g. if the contacted neighbor went down unexpectedly. Also, there is no need for it, since there is no possibility to stop the node from withdrawing.

Neighbor's failure

There is no guarantee that all the neighbors leave the network properly. The program itself can encounter error or be terminated violently. Another possibility is some unpredictable error of physical character, for example power failure, net-

work problem etc. In those cases it's essential for the other nodes in the network to be informed about this fact. Especially it's very important for the initiator who had some tasks processed by this node. In order to ensure handling of this possibility, the neighbors list is checked periodically. If some neighbor does not respond, it is removed from the list and all the data connected with it are treated accordingly. Namely the chunks assigned to it are resent.

1.3.3 Distribution of chunks

Once the file is split, the chunks has to be distributed, processed and finally collected. To achieve this, we must deal with several issues, which are described further.

Life cycle of the chunk

Its description is displayed in the Figure 1.2. Firstly, we have to keep track of every chunk's state. That is, we have to know whether the chunk is waiting in the queue, has been sent to be processed or has returned already. Each chunk is represented by the dedicated structure, which holds information about it. It also carry information essential for the transfer. This structure is further described in chapter 2. From now on we will use the term chunk for both the physical file and the reference.

Typical chunk's life cycle looks like this: The chunk is created and pushed to the waiting queue. Later it's popped out and transfered to the processing node. There it is enqueued for processing, then processed and sent back. Meanwhile the initiator holds the reference in the list of tasks being processed. In case of failure of the processing node, the chunk is pushed to the waiting queue again. Also when the chunk waits for return, it's checked periodically and if the computation takes too long, it's resent too. This happens because the respective chunk's encoding could fail or there were some problems with it. There is a possibility, that it will be computed sooner by another neighbor. Note that this can cause the situation, when one task is being processed by more than one node. However it's not a problem at all, because if the chunk returns more than once, it simply is not accepted. Furthermore, when the chunk returns, all neighbors that have it assigned are notified, so this situation should not happen at all. Once the chunk returns successfully, the reference is moved to another list, where it waits for completion of the task. When all the chunks are collected, the joining process may begin and the task execution ends.

Storing files

Tightly coupled with this process is the problem of storing the files. Four files have to be created during the processing of every chunk. The first one is created when the original file is split. This file can't be removed until the processed chunk returns, because it has to be available in case that the conversion fails for some reason. Another two files are created at the processing node, one for the input and one to store the output. The last file is created at the initiating node again to hold the processed chunk. This means, that the initiator has to have free disk space at least twice larger than the resulting file.
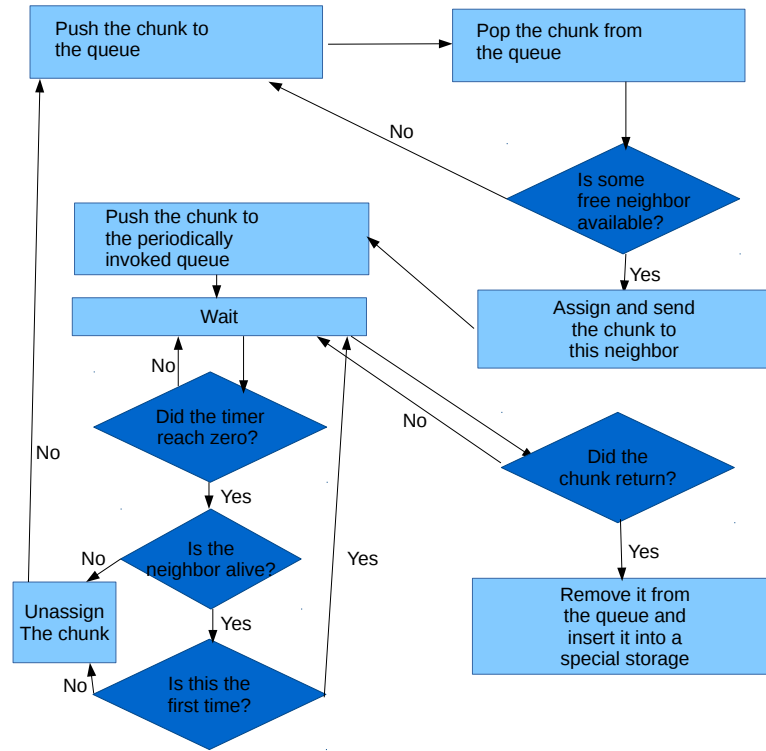
Figure 1.2: Processing a chunk - initiator part

Picking neighbors

Last but not least we have to choose policy to whom the chunks are distributed. We want to achieve as big speedup as possible, while preserving rather small list of neighbors. When the chunk is popped out from the queue, the initiator looks for suitable neighbor. That is the neighbor which has free status in the corresponding structure. If no such neighbor is found, the chunk is re-queued and another try is postponed. Also the gathering process described in the previous section begins. If some neighbor is available, the chunk is assigned to it and the transfer may begin. The initiator keeps track how many chunks were sent to the particular neighbor and it does not send more than specified count to one neighbor because it could potentially cause delay. The flag indicating whether the neighbor is free helps to control the flow. Each time chunk is assigned to the neighbor, the flag is set to false value to prevent sending more chunks in parallel. It's set to true again after the successful completion of the transfer. When the neighbor is too busy, it can express it in the communication, so the flag is set to false to prevent overloading of the neighbor. The flag is also refreshed during every periodic check.

1.3.4   Security issues

The present implementation is possibly vulnerable to some security threat. This is caused partly by the pure peer to peer nature, because it is difficult to control the traffic and authorize all nodes in dynamic environments like this. It also was

not our aim to solve this issue. The framework is supposed to be used mostly in LANs (Local Area Network) where all the peers are trustworthy. Otherwise it could be compromised easily. For example when the encoded chunk arrives, it is not checked whether it has been sent to this node or not. So a malicious chunk could be infiltrated causing bad output or even failure of the joining process.

1.3.5 Networking handling

The network communication is the most important part of the framework. Standard C library functions and structures were used which are conforming to POSIX.1-2001[6] standard. Although the program is intended to be used on the UNIX[7] or UNIX-like operating systems, it should be portable to the Microsoft Windows systems as well thanks to the use of this standard. To preserve simplicity, all the network communication makes use of the TCP (Transmission Control Protocol) [8]. The system primarily uses the IPv6 \footnote{`https://en.wikipedia.org/wiki/Internet_Protocol` addresses, but it can be run in the mode which uses addresses of the IPv4 family only. Additional and more detailed information can be found in chapter 2.

1.3.6 User interface

To provide interaction with the user, the curses[9] library is used. This library makes it possible to control the terminal screen. That means, the application does not require any special GUI (Graphical User Interface) libraries and is able to run interactively even on machines without the X server. The control is rather simple, offering possibilities to load the file, start or abort the process, show information about neighbors and so on. The interaction require only a keyboard, no mouse is needed at all. Concrete information together with some examples can be found in appendix A. Detailed information about the implementation, namely the synchronization problems are discussed in chapter 2.

---

[6]`https://en.wikipedia.org/wiki/POSIX#POSIX.1-2001`
[7]`https://en.wikipedia.org/wiki/Unix`
[8]`https://en.wikipedia.org/wiki/Transmission_Control_Protocol`
[9]`https://en.wikipedia.org/wiki/Curses_(programming_library)`

2. Implementation

This chapter describes some of the used mechanisms in more detail. It also introduces some classes and methods, but it is not supposed to serve as a detailed and full documentation. The Doxygen[1] software has been used to generate documentation so the complete overview of the code can be found in the attached HTML[2] documents. The program is implemented in the C++ language. This allows us to use the standard C library functions. Also the functionality provided by the C++ STL\footnote{`https://en.wikipedia.org/wiki/Standard_Template_Library` is exploited. Some of its containers are used as a base for containers with synchronized access implemented in the framework. Some of the functionality from the C++11[3] standard is also used, so the use of the program is limited to computers with compiler supporting this standard.

2.1   Networking

One of the most important issue is how to handle the networking. The chosen approach will be described in this section. As it has been said already, the program uses TCP for network communication. This is certainly good option when we want to handle data transfers, however, it can be considered unnecessarily demanding for simple tasks. To preserve the implementation simple, we chose not to use UDP. To utilize the possibilities of the operating system, standard socket API is used. Information about addresses are stored in the *sockaddr_storage* structures which are suitable for storing both IPv4 and IPv6 addresses. There are also some helper functions to work with address structures which are generic in the use of an address family. To provide easy manipulation with addresses, the structure *MyAddr* was created which groups the related functionality together. This approach makes it possible to switch between both IP versions easily.

2.1.1   Spawning connections

The ultimate class to handle the connections is the *NetworkHandler* class. It provides all the necessary functionality. Each instance of the program binds to the given listening port and starts accepting connections. When the connection is accepted, new thread is spawned to handle the connection. When the program wants to make the connection, it provides structure referring to a given neighbor together with the commands to be executed to the *NetworkHandler* instance. The connection is then spawned and handled.

Here we encounter the topic of commands. Each action is represented by a set of commands that implements it. Commands are instances of a class which inherits the class *Commnand*. Each command consists of two parts. When one node initiates the connection, it provides the vector of commands to be executed. They are processed in the loop in the following manner: The command's *execute* method is invoked. It communicates over the network. First it sends name of the

---

[1] `https://en.wikipedia.org/wiki/Doxygen`
[2] `https://en.wikipedia.org/wiki/HTML`
[3] `https://en.wikipedia.org/wiki/C++`

command to be invoked in the peer node. The peer's thread loops too. First it reads the name of the command and then it invokes it. In that moment there are command methods running in both nodes and they can communicate. When they end, the receiving node waits for another action. Meanwhile the initiating node spawns next command from the vector. If the vector is empty, the initiator ends the connection. This mechanism is used to handle all the network communication. What happens in case of problems is described in the section about handling errors.

Incoming connections are always handled asynchronously. Nevertheless, the outgoing connections may be handled in the synchronous way. It can be essential sometimes. For example if the node needs to obtain some potential neighbors because his list is empty, it has to wait until the action ends, because if it just sent the request and continued, it would probably find the list still empty.

2.1.2 Protocol

As it was said, the communication uses commands. Thread that handles the incoming connection is provided with respective file descriptor and address of the communicating node. First data that appears are considered as the listening port number of the node, so it can be identified in the neighbors list or added to the list of potential neighbors. Then the name of the command is sent, which is represented by the *enum* type. If no error occurs, the confirmation is send and the appropriate command's method is invoked. After returning from the method, another command is read. If there are no data left, the connection is closed. The most important commands are listed below.

1. Commands for maintaining

   - **Confirm** - Confirms the potential neighbor, adds the node to the neighbors list.
   - **Ask** - Asks the neighbor for the list of addresses, receives the list and adds the addresses to the potential neighbors list.
   - **Ping** - Verifies that the neighbor is alive, refreshes its status.
   - **Cancel** - Cancels the request for particular chunk's encoding.
   - **Goodbye** - Notifies the neighbor that the node itself is withdrawing from the network.

2. Commands regarding transfers

   - **Distribute** - Sends both the reference and the chunk itself during distribution.
   - **Return** - Returns the encoded chunk back, together with the referencing structure, which is updated with the data about encoding and transfers.
   - **Gather** - Spreads the request of the initiator to obtain more neighbors.

2.1.3 Transferring the data

First problem every network application has to deal with considers the byte order. The POSIX sockets API provides set of functions to deal with it. Namely it's *htons* and *ntohs*, or *htonl* and *ntohl* respectively. These functions convert the native representation of short (long) data types to the network byte order. The framework uses these function.

Basic transfer functions

Each of the functions mentioned in this paragraph has basically two parts. One for sending and its receiving counterpart. Integers are stored as the *int64_t* type which ensures correct communication even between 32 and 64 bit nodes. They are sent using the mentioned converting functions. When the string is transfered, first is sent the length of the string, followed by the appropriate number of characters. Commands are sent as numbers, wrapper functions are used which converts between the *enum* type and *int32_t* explicitly. Sending the structures containing the addresses is managed by another special function. It converts the address to string and sends it in this form, followed by the port number. This allows to handle both IPv4 and IPv6 addresses, the format is recognized during the reversed conversion.

Transfering files

The most delicate network task is to transfer the files. The file is first check, and its size is determined. Then it is sent and then the function repeatedly reads part of the file to buffer and sends it until whole file is processed. The count of sent bytes is compared with the actual file size in the end. The receiving side accepts the bytes and writes it to the file continuously, the file size is checked in the end. The data are first written to a temporary file which is renamed after the successful transfer. This mechanism prevents inconsistency of the received files. Each file is referenced by the structure. Among other things this structure stores counter of unsuccessful sent tries. If the counter exceeds given limit, the neighbor is treated as invalid and his state is set to non-free. The file is also checked, if it is not valid for some reason (the splitting process has encountered some error), the whole process has to be aborted because there is no way how to fix one specific chunk file.

2.1.4 Handling errors

Unfortunately the network environment is quite error prone and all the action has uncertain results. Moreover, the communication can be interrupted at any time. Because of this it is important for the network application to be able to deal with different error situations.

Errors during the connection handling

Almost all the functions indicates error state by the negative return code. These codes are checked so the error can propagate. If some error happens during the

control communication, the loop that handles the connection simply brakes, so the connection is closed. The commands are invoked in the try-catch block, so if the data have been corrupted or the synchronization has been lost, the next invalid command name raises an exception and the error is handled. The loss of synchronization may be detected thanks to the obligatory confirmation of every command.

Other errors

If an error happens during the file transfer, the receiving side detects inconsistency thanks to checking of the file size, so the bad file can be removed. Generally, if any error is encountered during the communication, the corresponding execute method indicates it by its return value, so it can be propagated further. The signal handler also has to be set to cover situations when the connection is destroyed unexpectedly and the SIGPIPE[4] is delivered.

2.2   Structures' overview

This section describes two main structures that are used in the framework. They common sign is, that they inherit from the *Listener* class, so they have to implement the invoke method. This fact makes it possible to use them as periodic listeners.

2.2.1   The TransferInfo structure

This structure serves for referencing the chunk. It contains flags used for transfer, addresses (source and destination), information about the video and path locating the physical position of the file. This field is important because it makes it possible to reference the file. It is changed several time during the process; as the state of the chunk changes, it it located in various directories and it is important to keep the value of the field actual. The structure also contains information which help to determine the encoding process and some statistics which describes the result. These are used to compute and update the quality of the neighbor. The structure is also equipped with a pair of methods that make it able to transfer it over the network. This simplifies the usage of the structure. When the referenced chunk is waiting for return, the corresponding method is invoked periodically. It decreases the timer. For the first time the timer reaches zero the neighbor which has this chunk assigned is checked. If it is alive, the timer is set one more time. If the neighbor doesn't respond or the timer reaches zero for the second time, the chunk is resent.

2.2.2   The NeighborInfo structure

Instances of this structure are kept in the NeighborStorage class. They represent the neighbor, that is its address and listening port. It also keeps the information about quality of the neighbor. The time elapsed from the last check is stored

---

[4]`https://en.wikipedia.org/wiki/Unix_signal`

too. Periodic invocation causes the timer to decrease and possibly contact the neighbor to refresh the state.

2.3 Important classes

2.3.1 The NeighborStorage class

It is used for storing the *NeighborInfo* structures. It provides several methods to maintain neighbors list while preserving synchronization. This is crucial, because the information about neighbor can change any time but it's desirable to keep our knowledge consistent. The class exists in one instance and helps to keep the information about neighbors in one place, so the manipulation can be controlled.

2.3.2 The NetworkHandler class

This class is used to handle all the networking issues. It provides functionality to spawn the connections or contact neighbors. It also holds the list of potential neighbors. This list differs from the neighbors list, because besides address and port, which are necessary, no further info is stored about potential neighbors. Also, most of the other classes have not got a notion about this list. When the lack of neighbors appears, simply the function *obtainNeighbors* is invoked, which uses the list internally. This class also handles adding of the new neighbors.

2.3.3 The Data class

It is a singleton class which helps to keep all the data at one place. It also makes the data accessible from anywhere in the program. It holds the instances of the *NeighborStorage*, *State* and all the queues that are used during the transfer and the encoding process.

There are more significant classes such as TaskHandler and WindowPrinter, which will be discussed in the corresponding sections.

2.4 Periodic actions

The framework uses a mechanism which invokes some actions periodically. Pros and cons of this approach are described in the respective sections, namely the alternatives. To implement this mechanism, separate thread runs that loops and once after each time quantum it invokes methods of structures that inherit from the *Listener* abstract class and are stored in the special queue. The time quantum is defined as a constant, so all timers actually express count of the quanta left. Obvious disadvantage of this approach is busy waiting that is used in the loop. Alternative approach could use a signal handler and setting an alarm. However, this could lead to not necessary asynchronous interrupts. Since the mutexes are used to avoid race conditions, a problem could occur if the signal interrupted some method holding a "bad" mutex.

2.5 Queues

As it is described in the corresponding section, the references to chunks can appear in different queues. The chunk's placement depends on the state in which it is. Since the whole process is nondeterministic, different conditions may occur and more than one thread could need to work with the queue at one moment. This means, that some way of serialization has to be provided. Because of this, the *SynchronizedQueue structure* has been created. Basically it provides usual functionality that could be requested from the queue, but ensures handling race conditions because only one thread at a time can access the underlying data. This is ensured by the *mutex*. The *pop* method also uses the conditional variable, so if there are no data that can be popped at the moment of invocation, it blocks and waits for a signal.

2.6 Working with input and output

All the file operations are customized for use on the UNIX operating system. Nevertheless, there is possibility to implement respective functions to work on different operating systems easily.

File operations

Each chunk is stored in a separate file. For this reason several functions were implemented to allow easier work with the file system. These include functions for manipulation with the filenames, controlling files and working with directories. Also there is a generic function which spawns an external process. This function uses process forking, spawns the desired process and returns contents of its standard output and standard error. The function also accepts value of timeout after which it kills the process. This ensures, that it will not hang. The result of the process' run propagates in the function's return value so the caller can react accordingly. This mechanism is used to work with video. Especially for splitting, encoding and joining it.

Working with the video

The video processing is secured by the *TaskHandler class*. When it is loaded, some useful piece of information is obtained thanks to the *ffprobe* program. To allow easy processing, the output is in the JSON[5] format which is then parsed with the help of the *rapidjson*[6] library. This library consists of header files only and is distributed as a part of the source code. It is available under the MIT license which makes it suitable for our usage. Parsed values can be showed using the F6 key. More importantly, they are used to compute the number of chunks that will be created. Note, that the number of chunks can change slightly after the splitting process. It is caused by the fact, that the theoretically computed count does not consider the positions of key frames. So the chunks can actually have different sizes and thus their count differs. This fact also implies, that each

---

[5]https://en.wikipedia.org/wiki/JSON
[6]https://github.com/miloyip/rapidjson

chunk has got slightly different size. When the task execution starts, the instance of *ffmpeg* program is spawned which splits the file. The result files are then stored in the special subdirectory of the working directory. For easy identification, each process has a unique code assigned to it. This code is generated from the time stamp. The chunks are numbered in increasing order. Chunk names are stored in the referencing structures that are created after the split finishes. Then the chunks are distributed as described in the section about distributing. Each processing neighbor encodes the chunks using the *ffmpeg* and then sends it back. The information about the chunk, such as the level of the encoding quality is stored in the referencing structure. When all the chunks are collected, a list of files to be joined is created. Then the *ffmpeg* is used again to join all these files to the output file. The process can be aborted at any time. This action stops the distribution, cleans the storages and notifies neighbors which are processing some chunks so they can trash it.

An alternative approach to splitting the file was used during the development. Firstly the position of every split was computed. Then it was spawned one process per each chunk. The advantage was, that the distribution could begin after the first chunk was created and therefore some time was saved. However, this approach turned out to be bad because of the existence of key frames. It does not allow to split the file at the arbitrary position so certain shifts were observable in the result file.

User interaction

To provide interaction with the user, the *curses* library is used. User can provide input from the keyboard. There is set no delay of the input, so the buffering is disabled. Because of this, we need to handle the user input manually. On the other hand, it also makes it possible to control the input and accept the commands immediately. The output is provided using the *WindowPrinter* class. This class stores the queue of records and provides functionality to add or remove some of them. Each record holds the line to be outputted together with the style of the line. So each line can be displayed differently than the others.

The screen is divided into four parts. Each part spans the whole width. There is a line displaying available commands at the top. The biggest portion of the vertical space belongs to two windows of equal height. The first one displays different information about processing, neighbors or file properties. The second window displays the status changes, notifications and potentially some debugging messages. The bottom part shows a prompt when user input is required.

Because there is usually a lot of threads that can cause the screen to refresh (this means the particular *WindowPrinter* instance is updated), it is important to allow only one graphical update at the moment, otherwise it could cause inconsistency of the graphical data. This is ensured by a mutex assigned to each *WindowPrinter* instance.

2.7 Synchronization

Because of the nondeterministic nature of the application, it is necessary to provide some kind of synchronization to ensure data and information consistency.

Mutex and conditional variable templates, available in the C++ standard library, helps to deal with this issue. We created implementations of queue and map like structures which ensure serialized access. These classes use containers from the STL and the synchronization primitives mentioned above. Namely they are *SynchronizedMap* and *SynchronizedQueue* and are used to store chunk references. Operations with the list of neighbors have to be synchronized too. For example a race condition could occur, when one thread would be working with the neighbor's reference while other would want to remove the same neighbor from the list.

Another area which have to deal with some race conditions is output which is displayed on the screen. The output is handled by the *curses* library which provides practically raw access to the graphical data. This means, that if more threads try to work with the screen at one moment, there is high possibility that they would compromise each other and nonsensical data would be displayed at the output. Moreover, this situation can also possibly result in the segmentation fault. To avoid these situations, the data that are supposed to appear on the screen are stored in the respective instances of the *WindowPrinter*. Mutexes are used to allow only one thread to change the content of the storage or refresh the screen. This approach has a disadvantage, that each call of the routine that produces some output could be blocking.

2.8   Error detection and recovery

During the process, various types of errors can occur. Errors connected with the networking are discussed in the respective section. Here we will describe other errors that could possibly happen.

Neighbor failure

If an unexpected failure of some node occurs, the other nodes which has it in their lists must react. If the communication is interrupted in the middle, there is no way how the other node can recognize the failure, so the command's execution just fails. Nevertheless, this usually leads to repetition of the command. This is able to register the failure. Generally, the failure is noted when the try to establish a connection with the given neighbor fails. It can occur during processing a command, checking a neighbor or when the timer assigned with some chunk reaches zero. In every of these situations the same function is used, so the situation is always treated in the same way. The unresponsive neighbor is removed from the list. If it has some chunks assigned, i.e. they has been sent to it already, they are queued for send to another neighbor. If it has sent some chunks to be processed by the current node, those chunks are trashed, because there will not be any use for them, since there is no neighbor they should be returned to.

Chunk disappearance

After the chunk is sent to be processed, it is pushed into the queue which invokes its members periodically. When the time is up for the first time, the respective neighbor is checked. If it responses, the timeout is set again. If it does not

respond, or the timer reaches zero for the second time, the chunk is queued for sent. Also, in case of the neighbor failure, the neighbor is removed. This can lead to a situation, when one chunk is being processed by two different neighbors at the same time. However, after it returns for the first time, it is put into the dedicated storage, so if it returns afterwards, it is simply rejected. But this situation should not occur, because when the chunk returns, all neighbors that have it assigned are notified. They can cancel the computation then. It can also occur the situation, when one node receives by accident one chunk more times. The files are checked and what is more, the transfer uses temporary files so the worst scenario involves wasteful encoding of the chunk for the second time. For this reason, each node remembers all the chunks it has processed, so this can be avoided. It is important to store only the successfully processed ones, because if the chunk was damaged during the transfer, the initiator could ask for its repeated encoding and it would be valid in this case. Another issue which has to be solved is how to set the timeout. When the neighbor is involved in the computation for the first time, we have no information about its performance. The timeout is thus set respectively to the size of the chunk, default multiplication factor is used. When the neighbor has already quality factor assigned, it is used to compute the timeout. So the quality coefficient can be seen as time needed to encode and transfer some unit of data. This coefficient is recomputed with every chunk delivered by the respective neighbor.

Other errors

Errors can also be encountered during the manipulation with the video. Because all the video related problems are manipulated by the external programs, the mechanism is used, which can control the process. The approximate upper bounds are set for each task that is supposed to be executed and if the process' execution takes too long, the signal is sent that terminates the process. Then the error code is returned.

3. Experiments

The purpose of the application is to speed up the computation process. Thus it should be verified, whether the improvement makes sense or do not. The improvement should correspond to the number of nodes involved in the computation. Our wish is, that the dependence is of some linear form, that is, the computation gets faster with every additional node and it improves by the same steps. In this hypothetical ideal case two nodes means two times faster computation and one hundred nodes means one hundred times faster achievement of the result. However, this is impossible for several reasons. At first, we must consider time that is taken by the division process. Additional time is consumed by the transfers and final join operation. Another problem arises from the fact, that the transfers are quite demanding themselves. So when more transfers are ongoing at a particular moment, the initiator is more utilized and the process can be slowed down due to this fact. This also implies that the improvement does not raise linearly when adding more nodes. Finally, we must consider delays which can appear due to technical reasons, network congestion or node failures.

3.1   Approach to testing

If we want to obtain reasonable data, the measurements must be repeated several times to prevent deviations. Also we want to keep the measurements independent to make its statistical processing easier. Our approach to the testing and gathering results is described in this chapter. Our main goal is to measure the improvement, but we would also like to measure the impact of the particular setting on the result.

The tests were run in the school laboratory. The network consists of several computers connected together with common ethernet twisted pair cables. Each computer has currently installed 64 bit Gentoo Linux[1] with the Linux kernel version 3.18. The machines are equipped with Intel Core i7 processors and 6 GB of operation memory. The MTU[2] is set to 1500B and the network uses Gigabit Ethernet[3].

Because of the number of tests, it is desirable for the testing process to be automated. Special Bash script was created for this reason. The script is tailored to be used at the testing laboratory, so it may need little modifications to work in some different environment. It is distributed with the source code of the framework. To allow automated and robust execution of the tests, special functionality was added to the program. It is invokable by option given at the start time and causes the program to run in a non-interactive mode. No input is accepted in this mode, the program just processes given file and ends. This options assumes all the essential data are given at the start time of the program. To keep the measurements independent, all the instances (on every node) of the program are started when the test begins and they are killed in the end. Communication

---

[1] https://www.gentoo.org/
[2] https://en.wikipedia.org/wiki/Maximum_Transmission_Unit
[3] https://en.wikipedia.org/wiki/Ethernet

with the remote nodes is handled by the ssh program[4]. The testing script uses a special file which describes the particular run. Working example of such file together with explanations of the values is given below.

```
v6 // use IPv6
/afs/ms/u/h/hudecekv/futu.avi // location of the file to be re-encoded
2 // run the whole scenario twice
slower // quality of the encoding
10000 // chunk size [KByte]
2048576 // transfer buffer size
spawn u-pl1 2221 // spawn the program on the machine 'u-pl1', use port 2221
spawn u-pl2 2222
spawn u-pl4 2224
spawn u-pl5 2225
spawn u-pl6 2226
spawn u-pl7 2227
wait 10 // wait for ten seconds before next action
kill u-pl4 // kill the instance of program running on machine 'u-pl4'
spawn u-pl8 2228
spawn u-pl9 2229
spawn u-pl10 2230
```

Thanks to this mechanism, various scenarios can be run easily without the need of human interaction.

The data were collected by running each test ten times for the given configuration. The count of involved nodes varied from one to ten. Each test was run once with chunks of 40 000 kB in size and once with 10 000 kB chunks. The same file was used each time as well as the encoding quality. The sample input file was packed in the *avi* container, encoded with the *msmpeg* codec. It was re-encoded with *H264* codec and stored in the *mkv* container. Each test gathered various results, among others the average times needed for transfer and encoding, number of chunks, quality and count of involved nodes. Because we had not the chance to run the tests in some dedicated network, the computation times may vary for the given settings. It depends on the conditions during the test. This is obviously a problem, because we can't compare such results. The tests showed, that if we multiply the average time needed to encode one chunk by the count of chunks, the product corresponds to the time that the serial encoding process would take. This allows us to deal with the problem, because we can use this computed estimation to obtain the improvement and the error will be minimal.

The desired values have been gathered in two ways. Some of them, for example average transfer and encoding times, are measured directly in the program and then outputted to a special file. The testing script just reads it from this file. The rest of the values is measured by the script.

---

[4]https://en.wikipedia.org/wiki/Secure_Shell

3.2 Results

3.2.1 Interpreting the results

In the Figure 3.1 are showed the achieved results, interpreted with respect to time needed by the single node. The x-axis shows count of nodes, the y-axis the portion of time needed by the distributed process. The blue dashed line represents the estimate which is based on the model which used hyperbolic function to predict the data. The obtained data are visualized as black crosses, red squares show respective mean values.
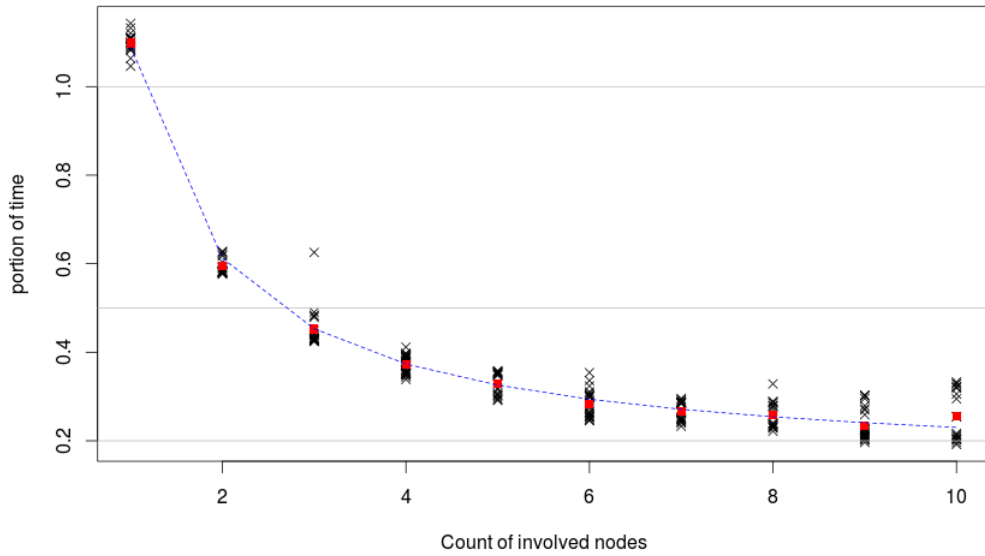

Figure 3.1: Achieved improvement - all measurements

The figures 3.2 - 3.4 shows the achieved speedup. The y-axis shows the obtained speedup. Model that was used is further described in the next section. The data are represented by gray crosses, the estimation based on the model is visualized by the blue dashed line. The red line shows the linear function, which would be the ideal case. This linear function is of the form

$$y = q * x$$

where coefficient q represents the influence of the time consumed by the data transfers. The Figure 3.4 shows, that for this chunk size the time spent with the distribution causes latency when more nodes are employed. This is caused by the fact, that too many transfers are processed in parallel. Consequently, some of the nodes have to wait for the chunks for too long.

In the Figures 3.5 and 3.6 are displayed ratios between particular operations performed during the process. The ratios are displayed with respect to the time that would be taken by the serial execution. Each column represents one measurement (process). The first image displays data for chunks with 10 MB in size,
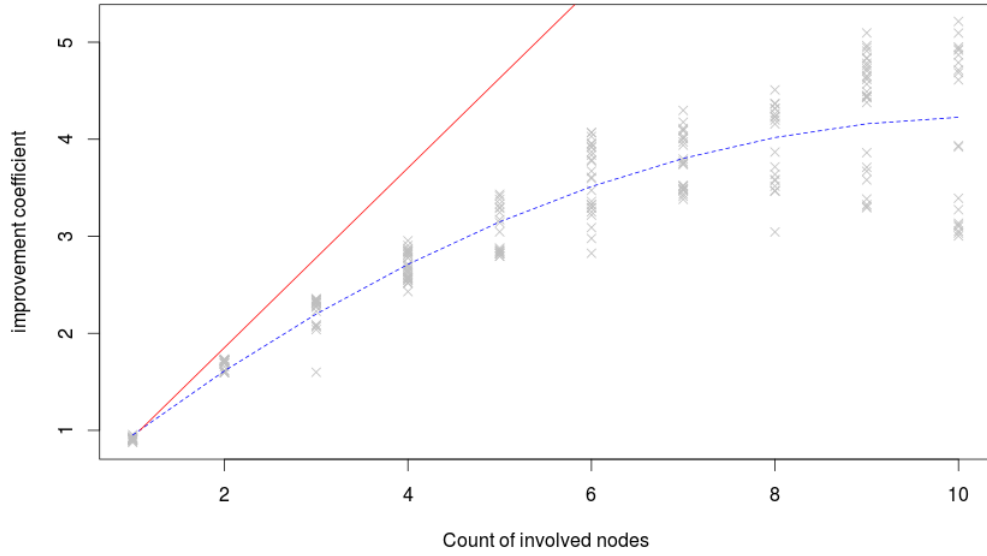
26

Figure 3.2: Achieved speedup - all measurements

the second shows 40 MB chunks. The data are sorted according to the number of neighbors used in the process. We can see that portion of time spent with network transfers is relatively small in our case. These charts were generated using the LibreOffice package[5]

Figure 3.7 shows results of the experiment, in which one of the nodes was killed during the process and then spawned again. As a result, several chunks were sent more times, depending on the conditions in the network. The plot shows average number of chunk sent and achieved improvement. In this experiment, 40 MB chunks were used, displayed in the left side. Also, some of the experiments were intentionally run with no failure, so we can see results achieved with normal run in the left down corner and we can compare it. Those values are highlighted. We can see, that resending of chunks has great impact on the result. This problem could be reduced by using smaller chunks. When we used 5 MB chunks, the results improved significantly as showed in the right image in the Figure. This is caused by the fact, that with the same number of faults, the amount of work that needs to be done again is smaller when using smaller chunks. However, disadvantage of this approach is, that the split and join operations take slightly more time.
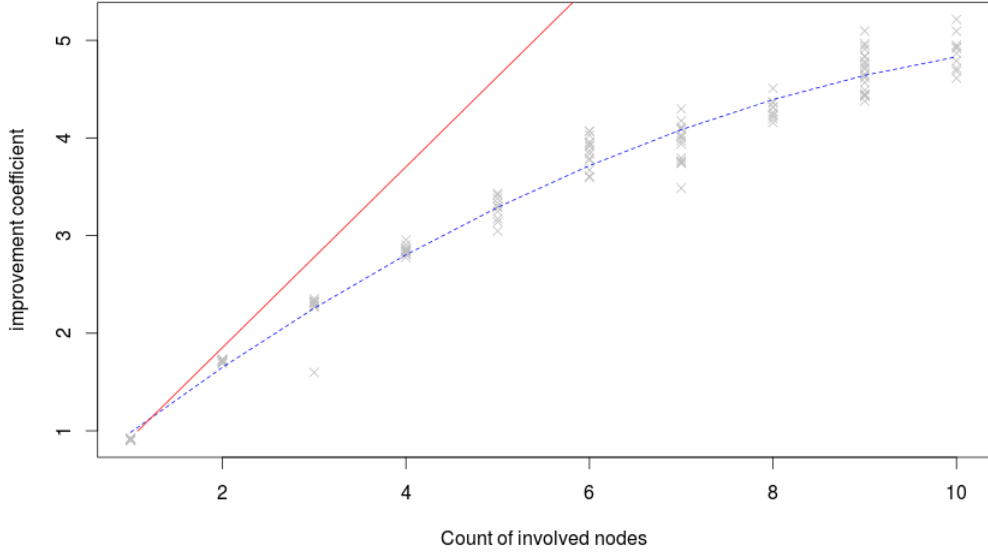
---

[5]http://www.libreoffice.org/

Figure 3.3: Achieved speedup - 10 MB chunks

3.2.2 Linear Model

Work with the data and the model was performed in the R Studio program[6]. To evaluate the data, simple linear regression model was used. Specifically, subsequent formula was used:

$$\frac{single\_node\_time}{distributed\_time} = \beta_0 + \beta_1 \times neighbor\_count + \beta_2 \times neighbor\_count^2 + \epsilon_i$$

Higher powers were not used in order to not over fit the model. The analysis of the model showed, that it makes sense to use this model. The assumptions such as homoscedasticity (constant variance) and independence of errors were explored using plots and results given by the R Studio. Some of the mentioned outputs are given in the Figures 3.8 and 3.9. We can see, that according to the p-values corresponding to coefficients, all of them are significant for the model. Residual standard error shows, that the variance is not too big. In the plots we can see that the residuals unfortunately has not constant variance. The second plot also suggests, that they probably are not distributed normally. However, citation says: "heteroscedasticity has never been a reason to throw out an otherwise good model."[8] So we used this model in our modeling. It may be questionable, whether we had enough measurements, however, the model seems to be good enough to describe the data and predict the behavior for more nodes. We can also notice, that the β_2 coefficient is negative and since the squared value rises faster, there is some point at which the improvement stops raising, which corresponds to reality.
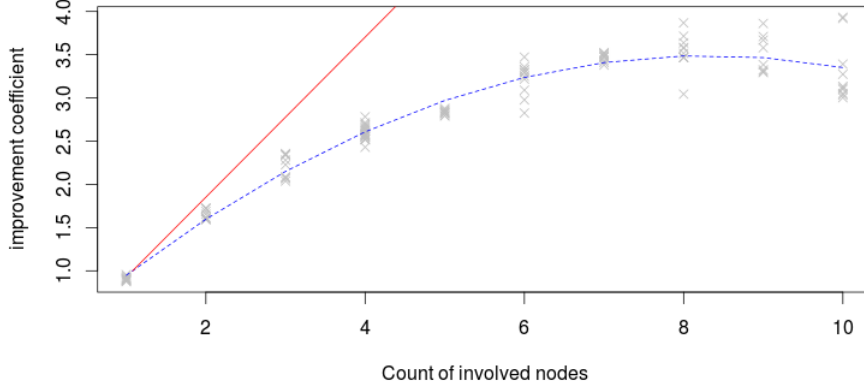
---

[6]https://www.rstudio.com/

Figure 3.4: Achieved speedup - 40 MB chunks

3.2.3 Upper Bounds

Overall time of the process can be divided into operations split, send, encode, receive and join (in this order). It is also influenced by searching appropriate neighbors, but it will not be taken into account in this analysis. Also, based on the data we can observe, that the time taken by the split operation is not very significant, therefore we can omit it. If we ask how big speedup can be achieved, we can use the Amdahl's law[7] to model the situation. The join operation can not be parallelized. The encoding itself is parallelized completely. The data transfer operations are performed theoretically in parallel, however, in practice we are limited by the network throughput and the system resources of the initiator, i.e. how many data transfers it can handle in parallel. This can be influenced by the OS, disk speed or even operation memory. Because this number is limited, the initiator can not employ arbitrary number of nodes effectively. If we want to deal with this problem, we have to consider several facts. The speed of the distribution is influenced by the ability of the nodes to receive the data with no delay, so we assume that buffering and disk operations do not slow down the process. Also, the full network capacity cannot be used because of other ongoing transfers. Some portion of the capacity is consumed by the redundant information used by the TCP/IP too. For the sake of simplicity, we will make the assumption, that the capacity corresponds to the actual amount of useful data transfered. We also must not forgot, that the sizes of original and received chunks differs. Finally, we treat the parallel transfer over the network as it took the same time as the serial, which does not have to be always true. We will consider the size of received chunks to be k times bigger than the size of the send ones and count with this size. Let c be the amount of data that can be transfered over the network per one second, s chunk size and t time needed to encode one chunk (here we assume the encoding times are the same for all the nodes). Then maximum number n of effectively employed nodes must fulfill the inequality:

$$\frac{n \times s}{c} \leq t$$

---

[7] https://en.wikipedia.org/wiki/Amdahl's_law
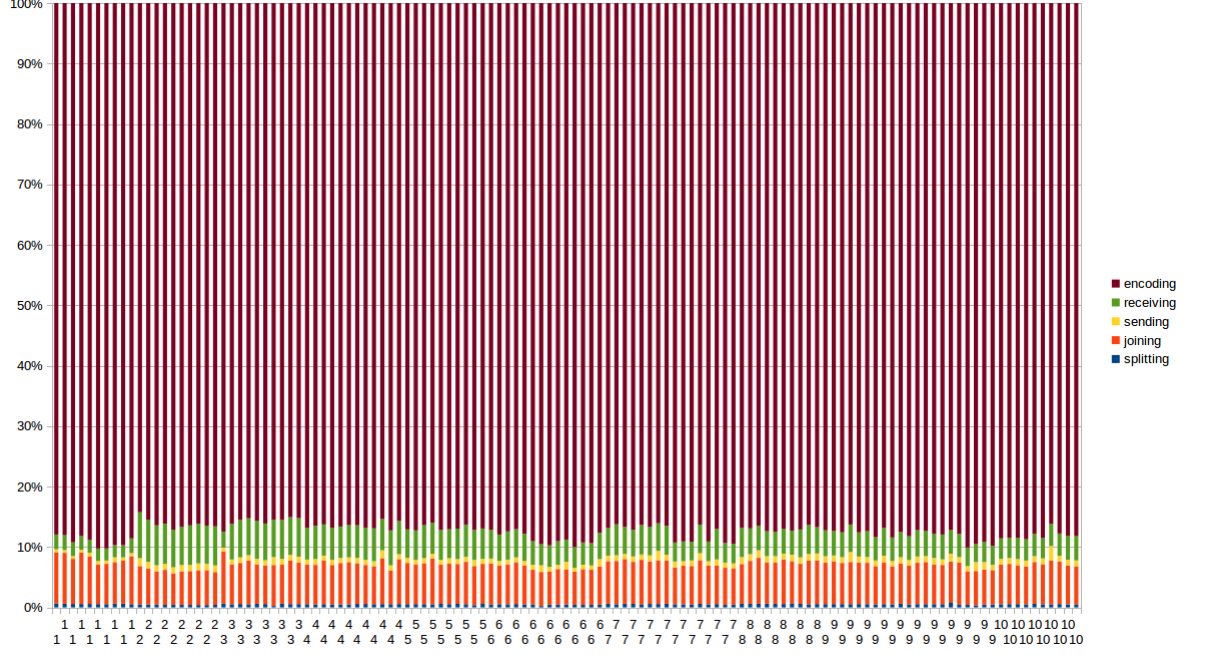
Figure 3.5: Comparison of operation times - 10 MB chunks

since otherwise the transfer would take more time than the encoding, so the encoded chunks would have to wait.

To applicate the Amdahl's law, we must determine the fraction of the algorithm that is strictly serial. This involves the splitting and joining. We consider the data transfers to be parallelized with the preceding paragraph in mind. According to the results, the split and join operations take approximately 7.5% of time in average. So according to the basic form of the law, the theoretical maximum speedup should be obtained from the following formula:

$$S = \lim_{n \to +\infty} \frac{1}{B + \frac{1}{n}(1-B)} = \frac{1}{B} = \frac{1}{0.075} = 13.33$$

Where B represents the serial fraction. This result is strictly theoretical and does not correspond to real situation, mainly because of the problems mentioned above. We can obtain more realistic results by using the formula to estimate the parallel fraction of time.

$$P_{est} = \frac{\frac{1}{speedup} - 1}{\frac{1}{nodecount} - 1}$$

For our data the estimation equals 0.89, which gives us theoretical maximum speedup 9.09.
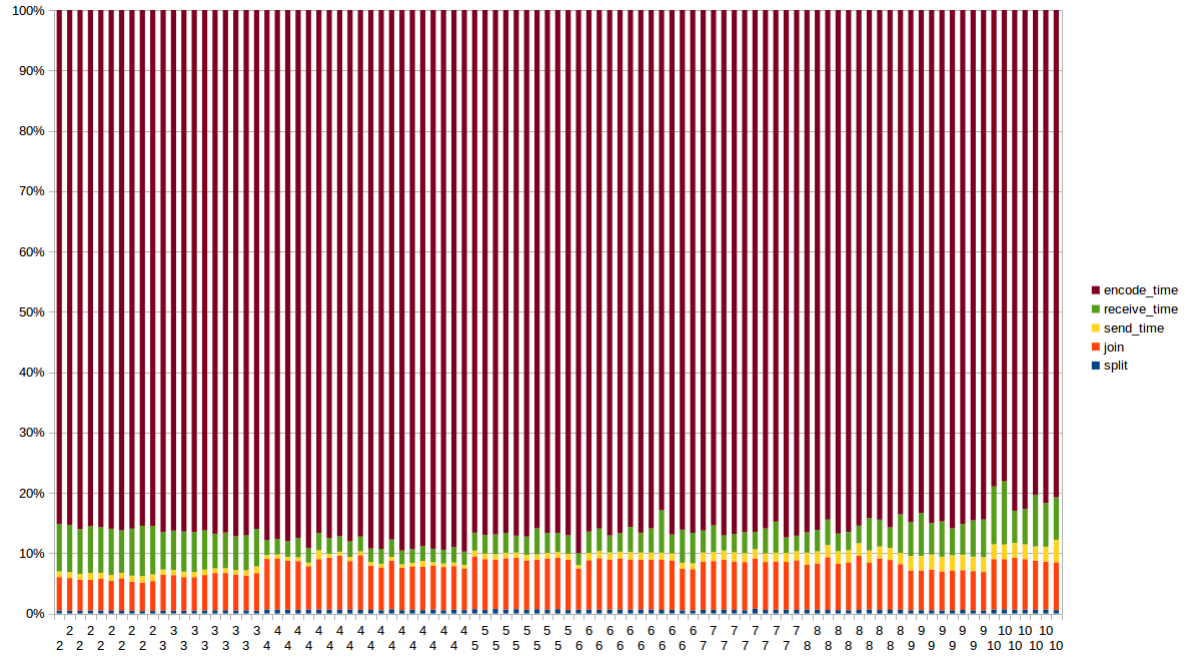
Figure 3.6: Comparison of operation times - 40 MB chunks
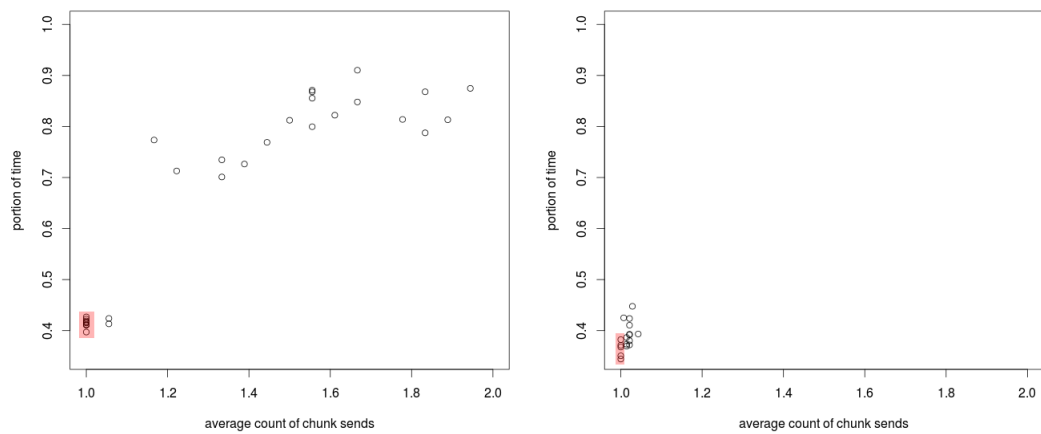
Figure 3.7: Impact of resending on the result

```
Console ~/ 

Residuals:
     Min      1Q   Median      3Q      Max
-1.22315 -0.19181  0.01452  0.21881  0.98769

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.216463   0.103147   2.099   0.0369 *
I(ncount)    0.771986   0.041967  18.395   <2e-16 ***
I(ncount^2) -0.037098   0.003687 -10.062   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3971 on 229 degrees of freedom
Multiple R-squared:  0.8736,	Adjusted R-squared:  0.8725
F-statistic: 791.4 on 2 and 229 DF,  p-value: < 2.2e-16

> |
```

Figure 3.8: Summary of the model
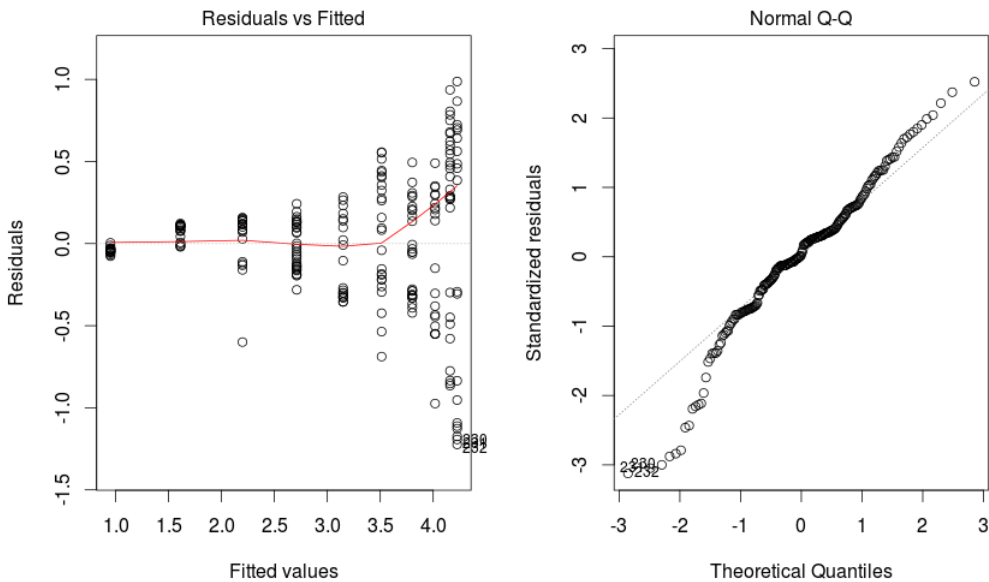
Figure 3.9: Graphical representation of the model data

4. Problems, alternatives and possible improvements

Some alternative approaches were also considered during the designing. One of them was not to use periodical checking at all. It was based on the idea, that there would exist permanent connection with each neighbor and the state changes would be indicated by the events related to this connection. However it was rejected due to the requirements connected with keeping the connection. Furthermore, the changes of ready state of the node would have to be checked either periodically or the node would have to inform all its neighbors (even potential) about each change which would lead to another problems to deal with. It is in contrast with passivity of the slaves too.

Another issue was related to more sophisticated way of distributing the chunks. Namely some kind of hierarchy was considered in which the chunk references would be distributed to neighbors in packs. At the neighbor it would be further split and distributed among neighbor's neighbors and so on. A kind of a tree structure would be formed this way. The transfer of the file would be processed directly between the initiator and the leaf node. However this approach turned out to be complicated and brings many problems. For example in case of failure of some node which is high in the hierarchy a lot of chunks would have to be re-distributed. Also the initiator's ability to control the distribution would be reduced. Moreover, the advantages of this approach are not so significant at all, because the biggest portion of time is spent during transfers and processing the chunks. The time spent with distribution of references is not important at all.

Interesting alternative would be use of the anycast[1] mechanism provided by the IPv6 protocol. The nodes would be addressed with an anycast address and each chunk would be simply sent to this address. Obvious disadvantage of this approach is the loss of the control of the distribution. More sophisticated variant would use special node which would maintain the list of free nodes and schedule the distribution. However this would break the peer to peer paradigm because of the centralization of control to one specific node.

Maybe the biggest unnecessary delay could appear when the whole process waits for some lost chunk. This is partially solved by resending chunks after the timeout. Also some form of redundancy could help, which on the other hand would certainly affect the effectiveness. If some node is processing more tasks sequentially while using approximately the same set of neighbors, the framework could also determine optimal chunk size to achieve good ratio of transfer and computing times. The chunks should not be too small because of the delays tied with its distribution. Optimal chunk size could significantly reduce the possible delay caused by waiting for re-encoding of some lost chunk. We could see at the end of the chapter 3, that use of smaller chunk size leads to better performance, especially when the failure of some node occurs, the difference can be quite big.

Also the current implementation creates a separate connection for each data transfer. Alternatively, each chunk could be delivered and returned using the same connection which would lessen the demands of the communication. The

---

[1]https://en.wikipedia.org/wiki/Anycast

connection's termination would also indicate problem with the chunk's processing or the neighbor itself. But the connection termination does not necessarily mean the failure of the process. Because it is desirable to avoid needless re-encoding of the chunks, this situation would has to be treated specially which would introduce additional problems. Also this approach does not fit very well to the current design in which each logical action is executed as the sequence of commands and for each sequence there is a special connection.

Conclusion

Based on the results of the experiments it can be said, that the framework works fine and it can be successfully used to speed up computation in the local network. Although the achieved speed up does not grow linearly with the increasing number of nodes, it can be quite significant. It was not tested in WAN environments, however, according to the results, the transfers of the data takes indispensable portion of the whole processing time, so the improvement depends on the network throughput. We have also shown, that for some count of involved nodes the optimum is reached. Adding more nodes slows the process down instead of speeding it up. This count depends on conditions in the network and chunk size.

As is being discussed in chapter 4, the framework could be further improved. To achieve more effective distribution of the work, more sophisticated scheduling could be employed that would take into account network topology or each node's performance and possibly create chunks of different sizes etc. It would require some better network knowledge. Also it was showed, that some redundancy for prevention of re-computing all chunks from one particular node in the case of failure could be useful. However, this would cause worse performance and also the advantages are quite unsure. Because of the speed of the network the data transfers generally seems to be a bottleneck. Thus good scheduling algorithm appears to be very important, together with optimal choice of the chunk size. This leads us to an idea, that in reliable network environments the logic could be centralized to a special node in order to achieve better performance. Potential malfunctions caused by the control node failure would have to be accepted. The control node would schedule the process for each client that would ask. Another advantage of this approach is the fact, that this node could use a knowledge of the current network state, that is, which nodes are employed and how.

Nevertheless, a lot of computer networks can suffer from unreliability and there always is a possibility of node failure. That is the reason, why our framework can be very useful. It is able to deal with error situations and does not require any special nodes. Among video encoding, it could be easily modified to process different tasks, such as processing large data sets or images. Although the efficiency is dependent on the current conditions, mainly the network speed, the achieved speedup can be quite significant.

Bibliography

[1] Fiorano Software. Super-peer architectures for distributed computing. `http://www.fiorano.com/whitepapers/superpeer.pdf`.

[2] Daniel Hughes James and James Walkerdine. Distributed video encoding over a peer-to-peer network. In *In the proceedings of PREP 2005*, 2005.

[3] Apache hadoop. `https://en.wikipedia.org/wiki/Apache_Hadoop`. Accessed: 2015-07-05.

[4] Mapreduce. `https://en.wikipedia.org/wiki/MapReduce`. Accessed: 2015-07-05.

[5] Boinc client–server technology. `https://en.wikipedia.org/wiki/BOINC_client-server_technology`. Accessed: 2015-07-05.

[6] Gnutella. `https://en.wikipedia.org/wiki/Gnutella`. Accessed: 2015-07-05.

[7] Bittorrent. `https://en.wikipedia.org/wiki/BitTorrent`. Accessed: 2015-07-05.

[8] D. N. Gujarati and D. C. Porter. 2009.

List of Figures

List of Tables

List of Abbreviations

- **LAN** Local Area Network

- **TCP** Transmission control protocol

- **IPv4 (IPv6)** Internet Protocol version 4 (version 6)

- **API** Application programming interface

- **GUI** Graphical User Interface

- **HTML** Hypertext Markup Language

- **OS** Operating System

- **STL** Standard Template Library

- **JSON** JavaScript Object Notation

- **MTU** Maximum Transmission Unit

- **B, kB, MB, GB** Byte, kiloByte, megaByte, gigaByte

Attachments

Appendix A - Installation and use

Download

First it's essential to get the source code. You can clone the code directly from the git repository using command

```
# git clone https://github.com/vojtsek/VideoCompression.git
```

Alternatively you can download the zip file and unpack it in some directory.

Requirements, installation and first run

You must have *ffmpeg* and *ffprobe* installed on your computer, if you want to run the program successfully. Although technically it doesn't matter which codec is used, the program currently uses H.264 standard as a default, so it assumes the *ffmpeg* has been compiled with the *x264* codec support. Otherwise the program does not have any special requirements except standard libraries which should be available on all UNIX systems. When you install these programs, you can change to the directory containing the source code and run the installation script:

```
# cd VideoCompression
# ./install.sh
```

The installation script is a regular Bash script, so the Bourne again shell interpreter is required to run it successfully. It uses utilities that are common part of every Linux distribution. If some of it is not present, you can either install it or do the preparation yourself. The script explores your computer, i.e. gets the IP address, finds location of *ffmpeg* binaries etc. Then it creates home directory for the program. The home directory contains data of the program's run. These include intermediate results as well as the final result and log files. The installation continues with generating the configuration file. This file is crucial for the framework. Before setting each option, the script prompts you for confirmation of the value. If you type nothing and just press the *Enter* key, the suggested value is used, otherwise the script uses your input. The result configuration file is stored in the *bin/* directory which is created during the installation. It's a plain text file so you can edit it anytime in the future. The script then continues with compilation of the program. If everything is all right, you can change to the newly created *bin/* directory and continue. The directory *bin/lists/* contains some supporting files that should not be changed. Otherwise the program could behave improperly.

The last step before you can run the program is to check the configuration. The configuration is saved in the *bin/CONF* file, which is created by the installation script. It's important to provide valid path to the *ffmpeg* and *ffprobe* executables and address with port of the neighbor that should be contacted initially. Otherwise you won't be able to join the network. The field *MY_IP* is not

essential as long as the initial neighbor is alive - it will be recognized automatically.

Then you can finally run the program. Some of the settings can be changed by providing options, the available ones are listed in the table 4.1.

-s	no address will be contacted initially
-n [address]:port_number	node to contact initially
-a [address]:port_number	address and port to bound to
-h	directory path to the home directory
-i file	file to encode
-p port_number	listening port
-d level	debug level
-q quality	quality of encoding

Table 4.1: Table of the possible options.

If the string *IPv4* appears among parameters, the program will use only the IPv4 addresses[2], in which case should be the *CONF* file changed appropriately.

Using the program

When you run the program, the initial screen appears. You can perform desired action using function keys. You can see the initial screen in the Figure 4.1. Available options are highlighted.

Figure 4.1: Initial screen after joining.

The important key bindings are listed in the table 4.2.

---

[2]https://en.wikipedia.org/wiki/IPv4

F6	Show information
F7	Start the process
F8	Load the file
F9	Set values
F10	Abort the process
F12	Quit the program
Up, Down	Traverse available options
Enter	Confirm the input

Table 4.2: Table of the control keys.

First you should load the video file. When the corresponding function key is pressed, the program prompts you for the file location. You can type in the absolute path of the file. Once you use the file, it is stored in history which you can browse using up and down arrow keys. This can be seen in the Figure 4.2.


```
|Enough neighbors gained.                                                              |
+--------------------------------------------------------------------------------------+



Enter a file path:
>/data/futu.avi
```

Figure 4.2: Loading the file.

Then you can set some parameters or show different information using F6 respectively F9 function keys. These keys provides set of options which you can choose from. When you are satisfied with the settings, you can start the process. The program then starts splitting the file and distributing the chunks. It also keeps informing you about the progress. When the process is done, the file is joined and you can do further actions. You can find the result in the provided work directory. There should appear new folder with a timestamp of the current job. It contains the file named *orig_output.mkv*, where *orig* is the base name of the input file. Some screenshots from the ongoing process are displayed in the Figures 4.3, 4.4 and 4.5


```
+--------------------------------------------------------------------------------------+
|      My IP address:   Address: ::ffff:127.0.0.1 : 2500                                |
|      Neighbor count:                                 2                                |
|      encoded chunks:                               0/19                               |
|Chunks to send:                                                                       |
|job_1435327641747_6                                                                   |
|job_1435327641747_7                                                                   |
|job_1435327641747_8                                                                   |
|job_1435327641747_9                                                                   |
|                                                                                      |
+--------------------------------------------------------------------------------------+
```

Figure 4.3: Overview of the process.

To obtain more information about what is going on, the *-d* option may be used which allows to specify level of debug messages that will be showed.

```
+----------------------------------------------------------------------------------+
|      My IP address:    Address: ::ffff:127.0.0.1 : 2500                           |
|    Neighbor count:                                2                               |
|Chunks that I am processing:                                                      |
|job_1435327733953_9 (10349618 B); Address: ::ffff:127.0.0.1 : 4444                |
|job_1435327733953_5 (8959194 B); Address: ::ffff:127.0.0.1 : 4444                 |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
+----------------------------------------------------------------------------------+
```

Figure 4.4: Processing of the tasks.

```
Distributed video compression tool.
Commands:    F6 show  F7 start   F8 load    F9 set F10 Abort  F12 quit


+----------------------------------------------------------------------------------+
|      My IP address:    Address: ::ffff:127.0.0.1 : 2500                           |
|    Neighbor count:                                2                               |
|    encoded chunks:                              19/19                             |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
|                                                                                  |
+----------------------------------------------------------------------------------+
+----------------------------------------------------------------------------------+
|                                                                                  |
|                                                                                  |
|                                                                                  |
|Enough neighbors gained.                                                          |
|/data/futu.avi loaded.                                                            |
|Splitting file: /data/futu.avi                                                    |
|The operation took 5 seconds.                                                     |
|Joining the file: /data/WD/2500/futu_output.mkv                                   |
+----------------------------------------------------------------------------------+

(11%)███████
```

Figure 4.5: Joining the file.

Appendix B - Attached software

All the attached software can be found on the attached CD. It contains:

1. Sources of the framework, together with the installation script in the *Video-Compression/* directory. This directory also contains the *rapidjson* library and license agreements.

2. Documentation in the *doc/* directory. To view this documentation, it is recommended to open the index.html file in your favorite html browser.

3. FFmpeg and x264 codec sources in the *ffmpeg/* and *x264/* directory.