Lasse Malmberg

Development of a Client-Server Chat Application

Thesis
Kajaani University of Applied Sciences
Bachelor of Business Administration
Business Information Technology
2017

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ
TIIVISTELMÄ

Koulutusala Luonnontieteiden ala	Koulutusohjelma Tietojenkäsittely
Tekijä(t) Lasse Malmberg	
Työn nimi Asiakas-Palvelin-Keskusteluohjelman Kehitys	
Vaihtoehtoiset ammattiopinnot Datacenter-ratkaisut	Ohjaaja(t) Deepak K.C. Toimeksiantaja
Aika 2017-05-04	Sivumäärä ja liitteet 45

Opinnäytetyön tavoitteena oli kehittää yksinkertainen asiakas-palvelin-keskusteluohjelma. Opinnäytetyö keskittyy pääasiassa ohjelman kehitysprosessiin ja siinä käytettyihin työkaluihin.

Ohjelman kehityksessä käytettiin C++-ohjelmointikieltä ja SDL-ohjelmistokehityskirjastoa. Kehitysympäristö koostui enimmäkseen komentorivityökaluista.

Keskusteluohjelma saatiin kehitettyä työn aikana. Ohjelma on työpöytäapplikaatio, joka on testattu Windows 10 -käyttöjärjestelmällä. Ohjelmaa hallitaan komennoilla. Ohjelma sisältää sekä asiakas- että palvelinominaisuudet samassa suoritettavassa tiedostossa.

Lähdekoodi on luettavissa osoitteessa https://github.com/LasseMalmberg/chat. Säilytyspaikka sisältää Windows 10 -käyttöjärjestelmälle valmiiksi käännetyn version ohjelmasta.

Kieli	Englanti
Asiasanat	keskusteluohjelma, asiakas-palvelin, c++, sdl, ohjelmointi, ohjelmistokehitys
Säilytyspaikka	☒ Verkkokirjasto Theseus ☐ Kajaanin ammattikorkeakoulun kirjasto

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

THESIS
ABSTRACT

School Natural Sciences	Degree Programme Business Information Technology
Author(s) Lasse Malmberg	
Title Development of a Client-Server Chat Application	
Optional Professional Studies Datacenter solutions	Instructor(s) Deepak K.C. Commissioned by
Date 2017-05-04	Total Number of Pages and Appendices 45

The goal of the Bachelor's thesis was to develop a simple client-server chat application. The focus was on the development process and the underlying technologies that were utilized in the development of the chat application.

The chat application was developed with the C++ programming language and the SDL software development library. The development environment consisted primarily of command-line operated tools.

The thesis resulted in a functional chat application. The program is a desktop application that has been tested on the Windows 10 operating system. The program is controlled with commands. The same executable file includes both client and server functionality.

The source code is available at https://github.com/LasseMalmberg/chat. The repository includes a precompiled version of the program for the Windows 10 operating system.

Language of Thesis	English
Keywords	chat, client-server, c++, sdl, programming, software development
Deposited at	☒ Electronic library Theseus ☐ Library of Kajaani University of Applied Sciences

TABLE OF CONTENTS

1 INTRODUCTION 1

2 NETWORKS 2

2.1 The Internet 2

2.2 TCP/IP suite 3

2.2.1 Application layer 4

2.2.2 Transport layer 5

2.2.3 Network layer 5

2.2.4 Link layer 6

2.2.5 Physical layer 7

2.3 Application architectures 7

2.3.1 Client-server 7

2.3.2 Peer-to-peer 8

2.4 Sockets 9

2.4.1 TCP 11

2.4.2 UDP 11

2.5 NAT 12

3 TOOLS OF THE TRADE 16

3.1 Programming language 16

3.1.1 Considerations 16

3.1.2 C++ 17

3.2 Language processor 18

3.2.1 Compiler 19

3.2.2 GCC 19

3.3 Debugger 20

3.3.1 Debugging 20

3.3.2 GDB 21

3.4 Libraries 22

3.4.1 C++ standard library 22

3.4.2 SDL 23

3.5 Source code editor 23

3.6 Build automation tool 24

3.7 Version control system 25
3.8 Modeling language 26

4 CHAT APPLICATION 28
4.1 Planning 28
4.1.1 Overview 28
4.1.2 Requirements 29
4.1.3 User interface 30
4.2 Development environment 31
4.3 Coding conventions 32
4.4 Program structure 33
4.4.1 Engine 34
4.4.2 Application 36
4.5 Build system 38
4.6 Testing 38
4.7 Demonstration 39
4.8 Analysis 41

5 CONCLUSION 43

REFERENCES 44

LIST OF SYMBOLS

API	Application Programming Interface
AWS	Amazon Web Services
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamic-Link Library
DNS	Domain Name System
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
GNU	GNU’s Not Unix
GUI	Graphical User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IPv4	Internet Protocol Version 4
ISO	International Organization for Standardization
ISP	Internet Service Provider
MAC	Media Access Control
MinGW	Minimalist GNU for Windows
NAT	Network Address Translation
NIC	Network Interface Controller

P2P	Peer-to-Peer
SDL	Simple DirectMedia Layer
SMTP	Simple Mail Transfer Protocol
STL	Standard Template Library
STUN	Simple Traversal of UDP through NAT
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
VCS	Version Control System

1 INTRODUCTION

The goal of the thesis was to develop a simple client-server chat application. The focus was on the development process and the underlying technologies that were utilized in the development of the chat application. A chat was chosen as the topic for the application because it requires little domain-specific knowledge to understand. Thus, more time could be spent on developing the software instead of trying to grasp what the problem is. A chat is also very flexible in terms of scope; features could be added or removed easily.

The chat application was developed with the C++ programming language and the SDL software development library. The development environment consisted primarily of command-line operated tools.

Chapters two and three cover the preliminary theoretical knowledge that forms the basis for the chat application. Chapter two introduces the most fundamental computer networking concepts, while chapter three presents the tools that make up the toolkit of a software developer. Chapter four describes all the facets of the chat application's development process in detail.

The project resulted in a functional client-server chat application. The program is a desktop application that has been tested on the Windows 10 operating system. The application is operated with commands. The same executable file includes both client and server functionality.

The source code is available at https://github.com/LasseMalmberg/chat. The repository includes a precompiled version of the program for the Windows 10 operating system.

2 NETWORKS

A network can be defined as a group of two or more computers that exchange data (Microsoft Official Academic Course 2011, 2). Computer networking is a vast and complex subject area. Networks play a major role in the development of a networked chat application. This chapter covers the most fundamental and relevant networking concepts.

2.1 The Internet

The Internet is a computer network connecting hundreds of millions of computing devices from all around the world. These devices are traditional desktop computers and servers, and nontraditional end systems, such as gaming consoles, smartphones, and TVs. Devices connected to the Internet are referred to as hosts or end systems. (Kurose & Ross 2012, 2.)

The Internet started as a four-node network in late 1969. It was originally known as ARPANET, and developed by the United States Advanced Research Projects Agency. The goal was to allow scientists working from different locations to have access to powerful remote computers. Circuit switching was the first method used to transmit information. Circuit switching created a consistent circuit by dedicating and connecting smaller circuits into a longer path. Systems used the circuit to send information. Circuit switching provided a high-quality service. It was limited in usability, because the dedicated circuits could only be used for one purpose at a time. (Glazer & Madhav 2015, 16.)

Packet switching replaced circuit switching. It removed the requirement of having to dedicate a circuit to a single transmission. Packet switching divides transmissions into small chunks called packets. The packets are sent through shared lines using a process called store and forward. Nodes in the network are connected to other nodes using a line that can carry packets between the nodes. A node can store incoming packets and forward the packets to another node closer to the packets' destination. Packet switching allows multiple transmissions to happen at the same time, using the same lines. (Glazer & Madhav 2015, 16 - 17.)

Packet switching requires a formal protocol to define how data should be packaged into packets and forwarded through the network. ARPANET used the 1822 protocol. ARPANET

grew over the years, and eventually became part of a larger network now known as the Internet. The 1822 protocol and other protocols of the time evolved into the protocols that now form the backbone of the Internet. The protocols form a collection known as the TCP/IP suite. (Glazer & Madhav 2015, 16 - 17.)

2.2 TCP/IP suite

The TCP/IP suite consists of a tower of independent and abstracted layers. Each layer is supported by a host of interchangeable protocols. (Glazer & Madhav 2015, 17 – 18.) The Internet protocol stack is a model that explains the interactions of the layers used for Internet communication. It consists of five layers: the physical layer, the link layer, the network layer, the transport layer, and the application layer. (Kurose & Ross 2012, 50.) Figure 1 illustrates the Internet protocol stack. The arrows depict the flow of data from the sending host to the receiving host.

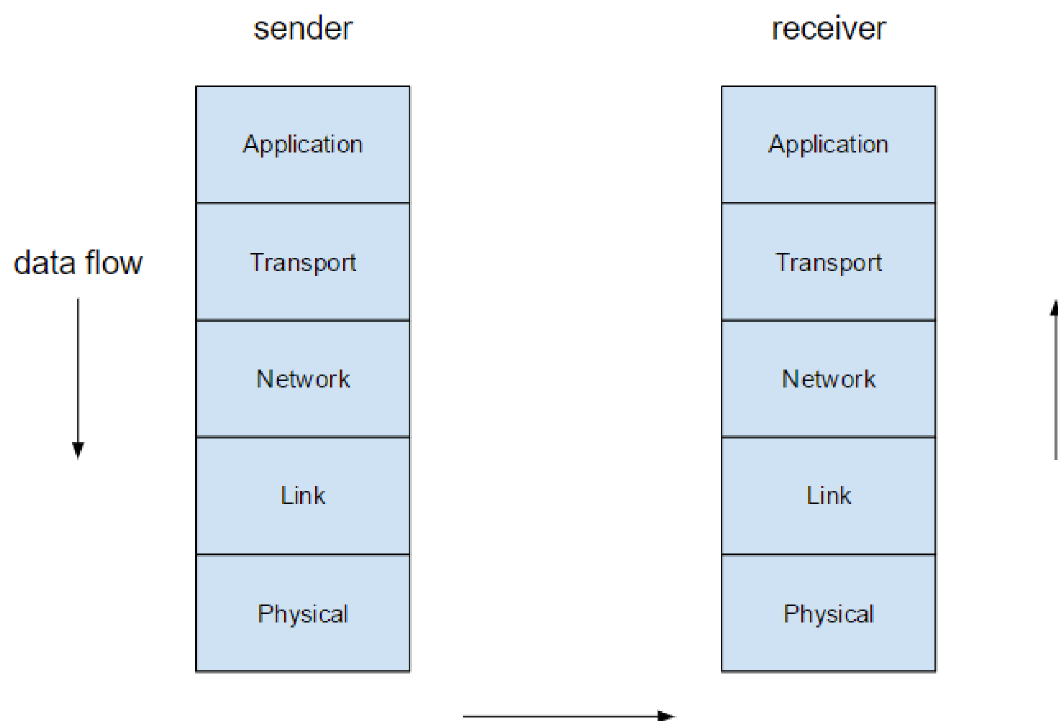

Figure 1. Internet protocol stack.

Each layer supports the layer above it and has its own purpose. All layers share six basic tasks. On a sending host, layers accept data from a higher layer, package the data, and forward it to the layer below. On a receiving host, layers accept data from a lower layer, unpackage the data, and forward it to the layer above. (Glazer & Madhav 2015, 18.)

A layer's definition does not specify how the layer should perform its tasks. All layers use various protocols to perform their tasks. A layer can be thought of as an interface, and each protocol or group of protocols as an implementation of the interface. (Glazer & Madhav 2015, 18 - 19.)

2.2.1 Application layer

The application layer starts the message creation process. FTP, SMTP, and Telnet are examples of end-user protocols that work at this layer. (Microsoft Official Academic Course 2011, 32.) Many of the Internet's fundamental protocols, such as DNS and DHCP, reside at this layer (Glazer & Madhav 2015, 52).

The DHCP or dynamic host configuration protocol allows a host to automatically request configuration information when it attaches to a network. The host broadcasts a DHCPDISCOVER message when it connects to a network. DHCP servers that are part of the network receive the message. If a DHCP server has an available IP address, it replies the host with an offer. If the host accepts the offer, it responds to the server requesting the IP address. If the offer is still available, the server responds to the host confirming the assignment of IP the address. The server also sends the host other relevant network information, such as the subnet mask. (Glazer & Madhav 2015, 52.)

The DNS or domain name system protocol enables the translation of domain and subdomain names into IP addresses. Users do not need to remember the IP address of a website. They can use its domain name. The domain name system translates the domain name into an IP address. The host queries a name server, which stores mappings from domain names to IP addresses. The name server responds to the client with the IP address matching the domain name. Name servers form a hierarchy. Most name servers are only authoritative for a small subset of the Internet's domains and subdomains. Hosts and name servers typically cache the results of queries. (Glazer & Madhav 2015, 52 – 53.)

2.2.2 Transport layer

The transport layer's responsibility is to allow individual processes on distant hosts to communicate. A host needs to know which process a received packet should be passed on to. The transport layer uses ports to achieve this. A port represents a communication endpoint on a host. Processes bind to ports. A port is identified by a 16-bit unsigned number. Communication addressed to a port is forwarded to the process bound to that port. (Glazer & Madhav 2015, 39 – 40.)

TCP and UDP are the most important transport layer protocols. UDP or the user datagram protocol is a lightweight protocol. It wraps data and sends it from a port on one host to a port on another host. UDP is unreliable. Data is not guaranteed to be delivered in order or at all. TCP or the transmission control protocol is reliable. It creates a persistent connection between two hosts. (Glazer & Madhav 2015, 41 - 42.)

Transport-layer protocols provide logical communication for processes running on different hosts. The processes see the hosts running the processes as directly connected, even if the hosts are distant. The logical communication provided by the transport layer allows remote processes to exchange messages. The processes do not need to know anything about the physical infrastructure used to transfer the messages. (Kurose & Ross 2012, 186.)

The transport layer converts application-layer messages into transport layer packets. The packets are known as transport-layer segments. The transport layer breaks the application-layer messages into smaller pieces. It adds a transport-layer header to each piece to form segments. The transport layer passes the segments to the network layer. The network layer encapsulates the segments within network layer packets. (Kurose & Ross 2012, 186.)

2.2.3 Network layer

The network layer provides host-to-host communication. Each host and router in a network makes use of the network layer. The network layer uses forwarding and routing to transfer packets from one host to another. A router forwards the packets it receives to the next router on the path towards the packets' destination. The network layer uses routing algorithms to determine the path the packets should take. (Kurose & Ross 2012, 308.)

The network layer adds a logical address infrastructure on top of the link layer. This makes it easy to replace host hardware and segregate groups of hosts into subnetworks. The logical address infrastructure allows hosts using different link-layer protocols and different physical media to communicate. (Glazer & Madhav 2015, 24.)

The internet protocol version 4 or IPv4 is the most common protocol used to implement the features of the network layer. It has a logical addressing system, a subnet system, and a routing system. The logical addressing system allows individual hosts to be named. The subnet system is used for creating physical subnetworks from the logical subsections of the address space. The routing system forwards data between subnets. IPv4 uses IP addresses to identify hosts. An IPv4 IP address is a 32-bit number. It is typically displayed to humans as four 8-bit numbers separated by periods. (Glazer & Madhav 2015, 24.)

The IP or Internet protocol service model is a best-effort delivery service. It does not guarantee orderly delivery of segments, integrity of the data in the segments, or that the segments are delivered at all. It is an unreliable service. One of the main responsibilities of the UDP and TCP transport layer protocols is to extend the Internet protocol's delivery service between two end systems to a delivery service between two processes on the end systems. (Kurose & Ross 2012, 190.)

2.2.4 Link layer

The link layer provides physically connected hosts a method of communication. It allows a source host to package information and transmit it through the physical layer. The destination host receives the package and extracts the information. The unit of transmission at the link layer is a frame. (Glazer & Madhav 2015, 19.)

The link layer uses protocols that correspond to physical media. Twisted pair Cat 6 cables and radio waves are examples of physical media. The 1000BASET Ethernet protocol corresponds to twisted pair cables. Other protocols correspond to different physical media. (Glazer & Madhav 2015, 20.)

Ethernet is a group of link layer protocols defined under IEEE 802.3. Varieties of Ethernet exist for different physical media and different transmission speeds. Ethernet uses media

access control addresses or MAC addresses to identify hosts. Every network interface controller or NIC that can connect to an Ethernet network has a theoretically unique MAC address. A MAC address allows devices connected to the network to be identified at the link layer. (Glazer & Madhav 2015, 21.)

2.2.5 Physical layer

The physical layer is responsible for providing a physical connection between networked computers. A physical medium is necessary to transmit information. Twisted pair cables, coaxial cables, fiber optic cables, phone lines, and radio waves are examples of physical media. (Glazer & Madhav 2015, 19.) The unit of transmission at the physical layer is a bit. Any network element that you can touch is part of the physical layer. (Microsoft Official Academic Course 2011, 32.)

2.3 Application architectures

An application architecture dictates how an application is structured over various end systems. The application developer designs the application architecture. The predominant architectural paradigms used in modern network applications are the client-server architecture and the peer-to-peer (P2P) architecture. (Kurose & Ross 2012, 86.)

2.3.1 Client-server

A client-server architecture consists of an always-on host, called the server, which responds to requests from many other hosts, called clients. The clients communicate through the server. (Kurose & Ross 2012, 86.) Figure 2 shows an example of a client-server architecture, where three clients (A – C) are connected to a server.

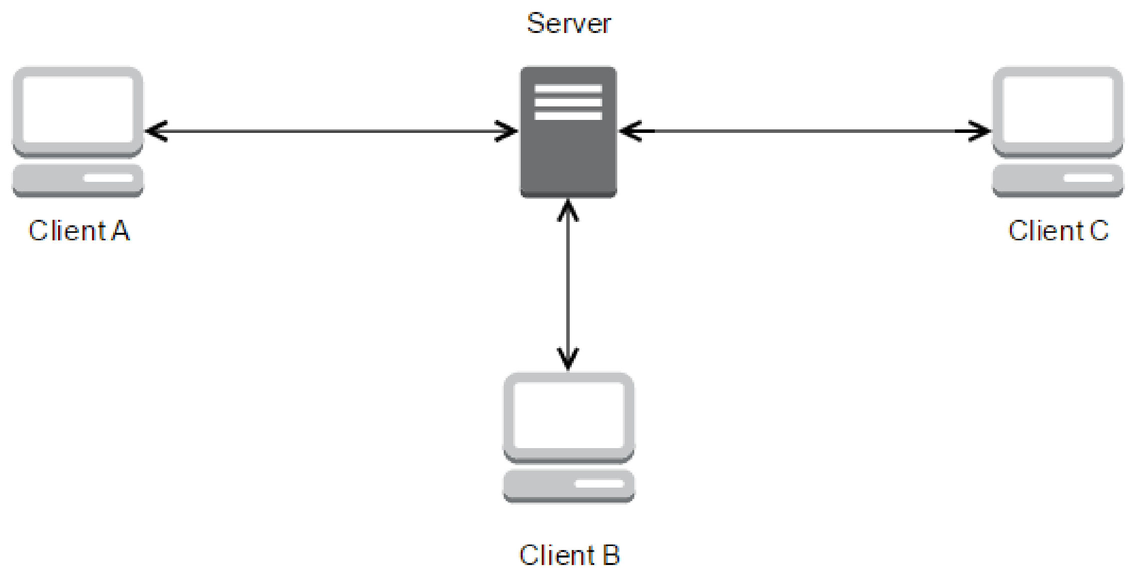

Figure 2. Client-server architecture.

The server usually has a static IP address, because it is more convenient for the clients. If the IP address constantly changed, it would be difficult to connect to the server. The server may be unable to handle all requests from clients with a single-server host. A more powerful virtual server consisting of multiple hosts can be created to alleviate this problem. (Kurose & Ross 2012, 86.) A virtual server refers in this context to a server that consists of multiple hosts, but appears as if it were just one.

2.3.2 Peer-to-peer

A peer-to-peer architecture has minimal or no reliance on dedicated servers. The hosts communicate directly. A host that is part of a peer-to-peer architecture is called a peer. (Kurose & Ross 2012, 86.) Figure 3 shows an example of a peer-to-peer architecture that consists of four connected peers (A – D).

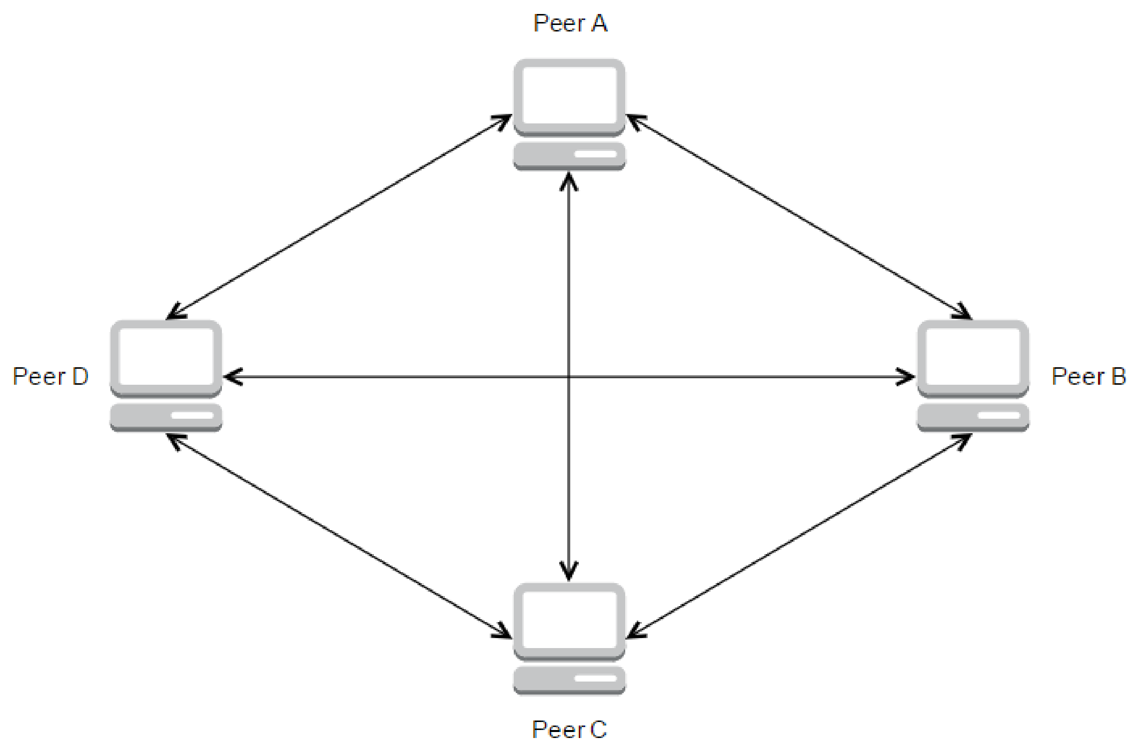

Figure 3. Peer-to-peer architecture.

A peer-to-peer architecture is self-scalable and cost effective. Self-scalability means that each peer adds capacity to the system. It is cost effective, because no significant server infrastructure is typically needed. Challenges that come with a peer-to-peer architecture include security, incentives, and ISP friendliness. Peer-to-peer applications are highly distributed and open in nature, which makes them hard to secure. They depend on the bandwidth, storage, and computing resources of the users, which makes their design problematic from an incentive standpoint. Most residential ISPs are asymmetrical in bandwidth usage, which means that they can handle a lot more downstream than upstream traffic. Peer-to-peer applications create lots of upstream traffic, putting considerable stress on the ISPs. (Kurose & Ross 2012, 87 - 88.)

2.4 Sockets

Sockets are software interfaces through which a process sends messages into, and receives messages from a network. A process is a program that is running within a host. A networked application consists of pairs of processes that communicate by exchanging messages across a

network. Messages sent by one process must go through the network to get to another process. The processes can be thought of as houses, and the sockets as their doors. (Kurose & Ross 2012, 89.)

A socket functions as an interface between the application layer and the transport layer. It is an API between the application and the network. Figure 4 illustrates this. The application developer controls everything on the application-layer side of the socket. The transport-layer side is more immutable. Choices on the transport-layer side are mostly limited to the transport protocol and some parameters such as the maximum segment size. (Kurose & Ross 2012, 89 - 90.)

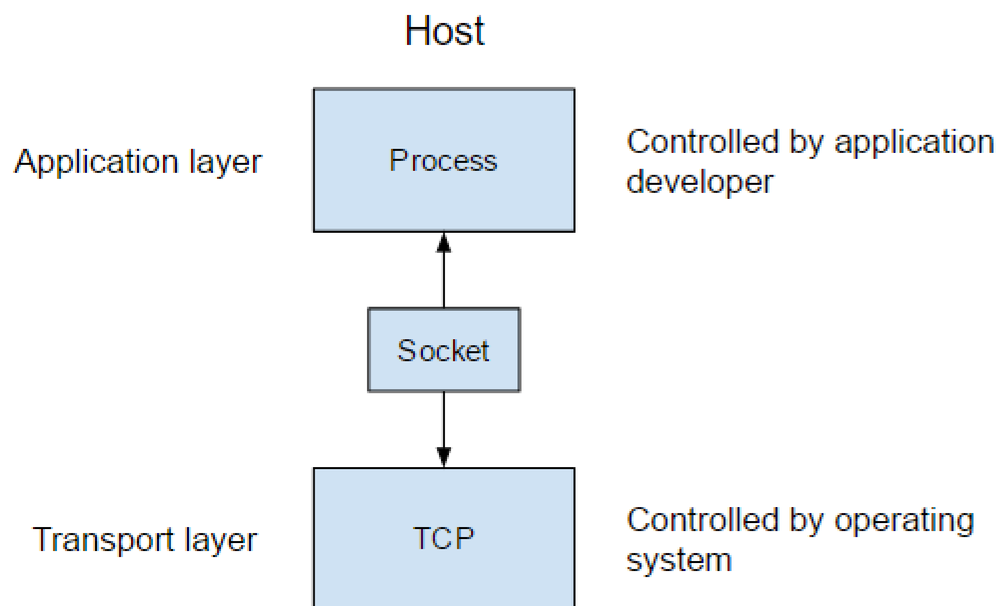

Figure 4. Location of sockets in relation to the Internet protocol stack.

The application developer selects the transport-layer protocol. The application is built using the transport-layer services provided by the protocol. Two transport-layer protocols are available when creating a networked application for the Internet: UDP and TCP. The protocols have different sets of services. (Kurose & Ross 2012, 89 – 90, 93.)

2.4.1 TCP

TCP provides reliable data transfer. Data is delivered in the correct order and with no duplicates. TCP includes a congestion-control mechanism, which throttles a sending process when the network between the sender and receiver is congested. The mechanism also attempts to distribute available network bandwidth evenly between each TCP connection. (Kurose & Ross 2012, 94 - 95.)

TCP requires that the communicating hosts exchange transport-layer control information before the flow of application-level messages can begin. This is called the handshaking procedure. It alerts the hosts and allows them to prepare for incoming packets. A TCP connection exists between the hosts' sockets once the handshaking procedure has been completed. The connection is full-duplex, which means that the processes can communicate simultaneously over the same connection. (Kurose & Ross 2012, 94.)

TCP requires nontrivial connection state tracking at both ends of the connection. The recipient must acknowledge received data. The sender must resend any unacknowledged data. TCP requires a larger header than UDP because of the additional services it provides. (Glazer & Madhav 2015, 42.)

2.4.2 UDP

UDP is a lightweight transport protocol that provides minimal services. It is connectionless. UDP does not perform a handshaking procedure before the two processes begin to communicate. UDP provides unreliable data transfer. Data sent into a UDP socket is not guaranteed to reach its destination in the correct order or at all. UDP has no congestion-control mechanism. (Kurose & Ross 2012, 95.) UDP is used when the loss of packets does not matter, like when streaming media (Microsoft Official Academic Course 2011, 39).

2.5 NAT

Network address translation or NAT allows an entire subnet of hosts to be connected to the Internet through a single shared public IP address. Every host communicating through the Internet must have a uniquely assigned and publicly routable IP address. The number of public IP addresses is limited. Certain IP address blocks have been reserved for use in private networks. Private IP addresses are usable by anyone, but not unique or publicly routable. NAT translates private IP addresses into public IP addresses, and vice versa. It allows hosts in private networks to communicate through the Internet. (Glazer & Madhav 2015, 53 - 56.)

A router forwards packets towards their destination. A router has a private IP address in the local network, and a public IP address in the Internet. Packets sent through the Internet by a private network host go through a router. The router is located between the host's private network and the Internet. The router uses NAT to replace the private source IP address of the packets with its own public IP address. The destination host that receives the packets takes the source IP address of the received packets and writes it as the destination IP address of the packets it sends back in response. The reply packets are destined to the public IP address of the original host's router. The packets are routable through the Internet, because their destination IP address is public. If NAT had not been used, the reply packets would not be routable. Their destination would be the private IP address of the original host. The router's NAT functionality maps the packets' private source IP address, port number, and destination IP address with a unique generated port number. The mapping is stored in a NAT table. The router checks the NAT table when it receives a packet. It modifies the packet based on the NAT table so that the packet is delivered to the correct host. (Glazer & Madhav 2015, 53 - 56.)

Figure 5 illustrates the NAT procedure. Host A is replying to host B. Router B's NAT table has an entry for the destination port used by host A, 3000. Both hosts are in private networks and have private IP addresses. Both hosts have a router between their private network and the Internet. Router A has the private IP address 192.168.0.1 and the public IP address 85.29.121.43. Router B has the private IP address 172.16.0.1 and the public IP address 89.31.130.24. The communicating processes within both hosts use the port number 100. The port number is suffixed to the IP address with a colon, as in 192.168.0.2:100.

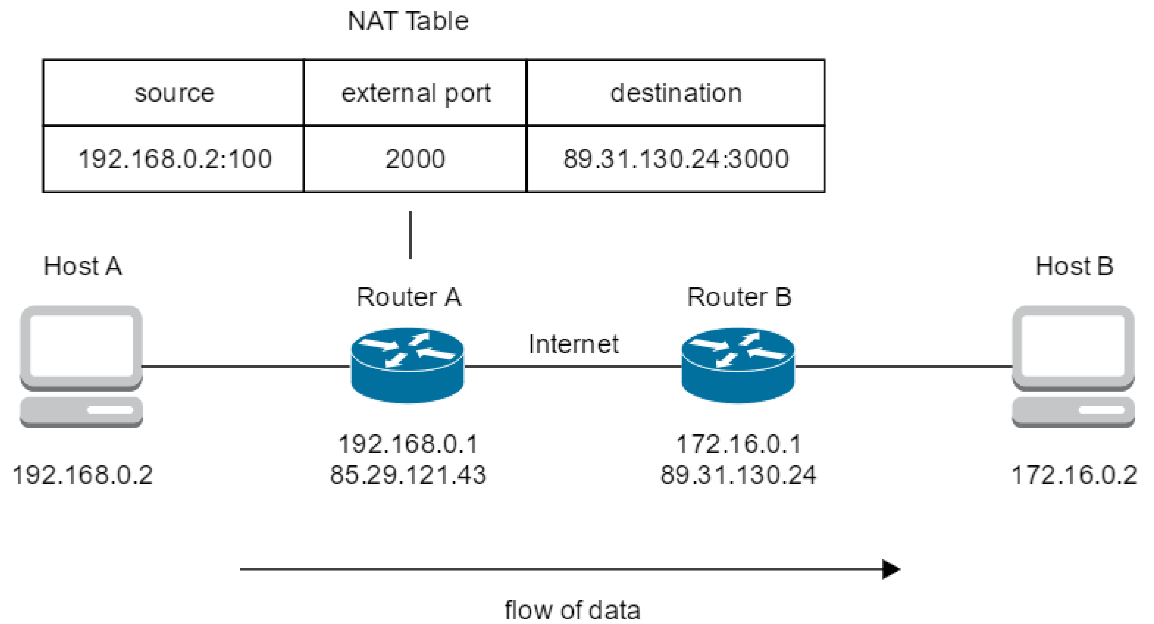

Figure 5. Router A adds an entry to its NAT table after receiving a packet from host A.

Host A sends a packet to Host B. The packet's source is 192.168.0.2:100. The packet's destination is 89.31.130.24:3000. Router A receives the packet, and adds an entry to its NAT table. It generates the unique external port number 2000. This number is mapped to the packet's source and destination information. Router A replaces the packet' source information with its own public IP address and the generated external port number. The packet's new source is 85.29.121.43:2000. The packet is forwarded to the Internet. Figure 6 shows the state of the packet before and after going through router A.

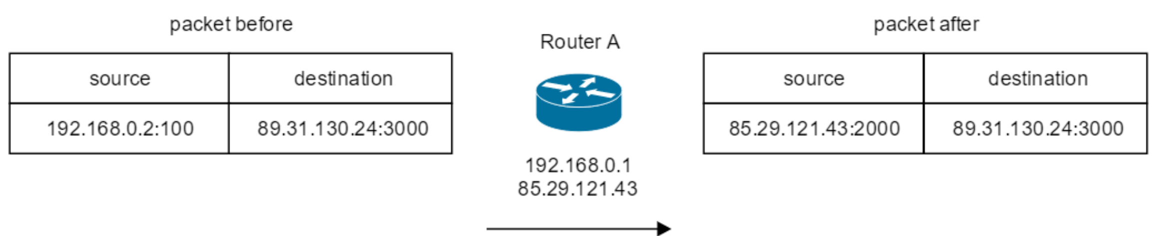

Figure 6. State of the packet before and after going through router A.

Router B receives the packet. It searches its NAT table for the packet's destination port number, 3000. It finds a match. Router B's NAT table is visible in figure 7.

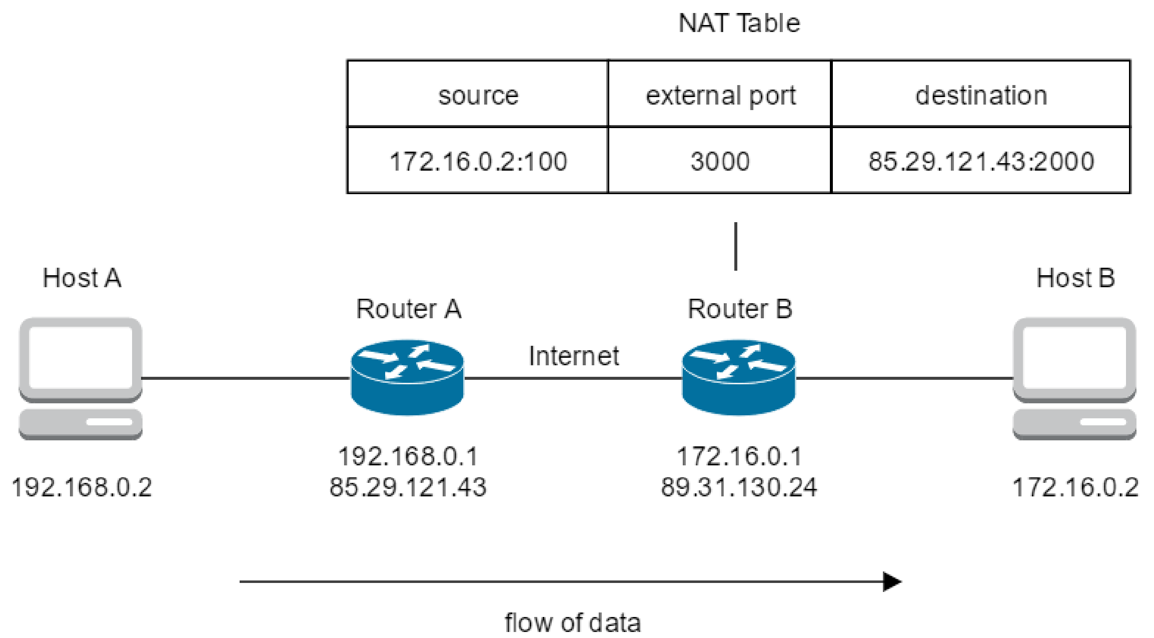

Figure 7. Router B searches its NAT table after receiving the packet sent by host A.

Router B changes the packet's destination to the information corresponding the port number in the NAT table. The packet's new destination is 172.16.0.2:100. The packet is successfully routed to host B. Figure 8 shows the state of the packet before and after going through router B.

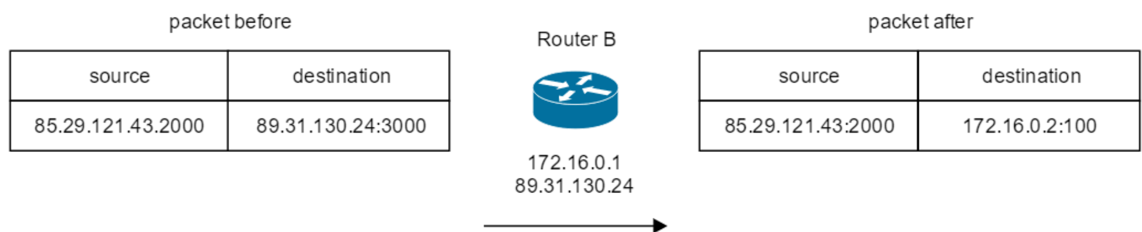

Figure 8. State of the packet before and after going through router B.

If router B had not have had an entry for the packet's destination port number, the packet would have been discarded. Two hosts that are both behind NAT run into a problem when they try to communicate. A connection cannot be initiated. The receiving host's router does not have an entry in its NAT table for the received packet's destination port number and source address. The packet is dropped. The first way to solve this problem is to have the receiving host manually configure port forwarding on the router. Port forwarding allows a port number to be associated with a host in the private network. Data destined to the port

number is forwarded to the corresponding host. The second way is to utilize STUN or simple traversal of UDP through NAT. This requires the help of a third-party host. The third-party host tells the hosts how to initiate the connection so that the correct entries are added in their routers' NAT tables. The hosts can communicate directly after the process is completed. (Glazer & Madhav 2015, 57.)

3 TOOLS OF THE TRADE

The software development process requires the use of a variety of tools. This chapter introduces the tools that are relevant to the practical part of the thesis. Each subchapter first covers the tool in question from a general standpoint. The actual tool that was used to develop the chat application is then introduced.

3.1 Programming language

A programming language is one of the first tools that must be decided on when starting the development of a new application. A simplistic definition of a programming language is that it is used to give instructions to a computer. This chapter describes general properties of programming languages and the decision-making process that goes into the selection of a programming language. The C++ programming language is then introduced.

3.1.1 Considerations

Programming languages provide widely different feature sets. What is the typing model? What is the programming model? Is the language compiled or interpreted? What decision constructs and core data structures does the language use? What unique features does the language support? (Tate 2010, 18.)

The large number of programming languages can be contributed to a few factors. Computer science is a young discipline, and better ways of doing things are constantly found. Many new programming paradigms have come into existence over the past decades. Different programming languages are good at different things. Many languages were even designed to be used within specific problem domains. (Scott 2009, 7.)

Only a few dozen programming languages are widely used. The reason for this is multi-faceted. Any problem can technically be solved with any programming language, but programming languages vary in expressive power. A programming language's features have a significant impact on the programmer's ability to write clear, concise, and maintainable code. Certain

programming languages are easier. Basic is easier to pick up than C++. A programming language that can be easily implemented on many platforms will be more available and widely adopted. A programming language requires an official international standard or a single canonical implementation to be portable. Languages with open-source compilers or interpreters have fared well. The C programming language is a prime example of this. It was developed together with the original Unix operating system. Today, the leading open-source operating system, Linux, is written in C. Certain programming languages owe their success to excellent tools, or more specifically, compilers. Fortran has been around for ages, and companies have put immense effort into creating efficient compilers for it. Sometimes the forces behind a language's success are not purely technical. C# is a programming language that has arguably benefited considerably from the backing of Microsoft. (Scott 2009, 7 - 9.)

3.1.2 C++

Bjarne Stroustrup is the original developer and designer of the C++ programming language. He defines the language in three ways:

1. "C++ is a general-purpose programming language with a bias toward systems programming."
2. "C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight abstractions."
3. "C++ is a language for developing and using elegant and efficient abstractions."

C++ does not specialize in any one application area. It is designed to support a wide variety of uses. C++ is widely used in areas such as financial systems, game development, and scientific computation. (Stroustrup 2013, 9.)

C++ supports the following programming styles most directly: procedural programming, data abstraction, object-oriented programming, and generic programming. The goal is to support effective use of a combination of the styles. Procedural programming focuses on processing and the design of suitable data structures. Data abstraction focuses on the design of interfaces and the hiding of implementation details. Object-oriented programming focuses on the design, implementation, and use of class hierarchies. Generic programming focuses on the

design, implementation, and use of generic algorithms. C++ can be called class oriented. (Stroustrup 2013, 11.)

C++ originates from the C programming language, and mostly retains C as a subset. The primary difference between C and C++ is that C++ places more emphasis on types and structure. C++ renders many techniques used in C programming unnecessary. (Stroustrup 2013, 14 – 15.)

C++ is a compiled language. A compiler processes a program's source files and produces object files. A linker combines the object files into an executable program. Figure 9 illustrates the compilation process. The compilation process is more complex in practice, but the explanation is sufficient for this section. (Stroustrup 2013, 38.)

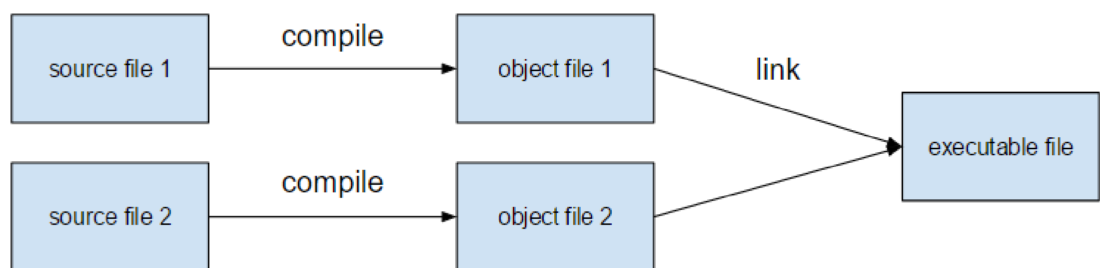

Figure 9. Simplified view of the C++ compilation process.

C++ programs are created for a specific hardware and software combination. C++ programs are not directly portable from one platform to another. The source code can be compiled on another system to achieve portability. (Stroustrup 2013, 38.)

3.2 Language processor

A language processor converts an algorithm in a source language into an equal version in a target language (Demiris 2012, 12). This chapter introduces the compiler, which is a type of language processor. The chapter ends with an overview of the GNU Compiler Collection.

3.2.1 Compiler

A compiler reads a program in one language and translates it into an equivalent program in another language. The original language is known as the source language, and the resulting language as the target language. The compiler reports any errors in the source program that are found during the translation process. If the target program is an executable machine-language program, the user can call it to process inputs and produce outputs. (Aho, Lam, Sethi & Ullman 2013, 1 - 2.)

A compiler is not the only program required to create an executable target program. A preprocessor collects the source program from multiple modules stored in separate files. The preprocessor also expands macros into source language statements. The modified source program is fed to a compiler, which produces an assembly-language program as its output. An assembler processes the assembly language and produces relocatable machine code. A linker links together separate object and library files into the code that ends up being run on the computer. Figure 10 shows the complete compilation process. (Aho et al. 2013, 3.)

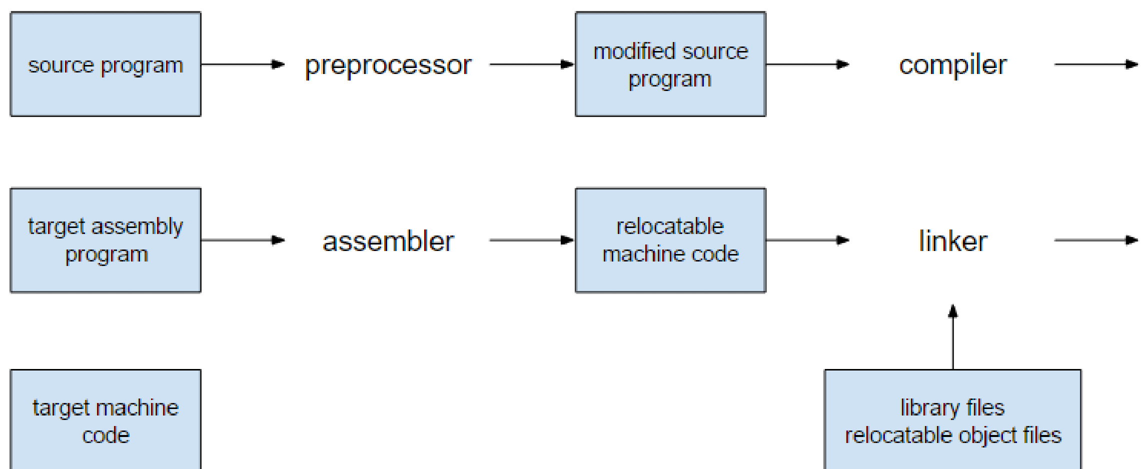

Figure 10. Compilation system.

3.2.2 GCC

GCC or the GNU Compiler Collection is a collection of compilers for several major programming languages. C, C++, and Objective-C are among the supported languages. The

name GCC originally stood for "GNU C Compiler". GCC is also used to refer to the language-independent part of GCC. The official meaning of GCC is "GNU Compiler Collection", which refers to the whole set of tools in general. (Stallman 2016, 3.)

The term "front end" refers to the part of a compiler that is specific to a given language. "Back ends" generate machine code for different processors, and belong to the language-independent component of GCC. The compiler specific to C++ is called G++, though it is also correct to refer to it as GCC. All the compilers in GCC directly generate machine code. GCC supports the original ISO C++ standard, and the 2011 and 2014 revisions. (Stallman 2016, 3, 6.)

GCC supports a wide variety of options that control its behavior. The "-c" option commands GCC to not run the linker. Because the linker is not run, object files are generated as the output. Some options work with all supported languages. Others are specific to a set of languages. A list of all options can be found in the user manual. (Stallman 2016, 9.)

3.3 Debugger

A debugger is a tool used to track down, isolate, and remove defects from software. The defects are called bugs. A debugger helps in understanding how a program works. A programmer uses a debugger to follow the program's flow of execution. The program can be stopped at any point to inspect its state for correctness. (Rosenberg 1996, 1 – 2.) This chapter examines debuggers and the debugging process. The chapter ends with an introduction to the GDB debugger.

3.3.1 Debugging

A debugger is text-based or GUI-based. Many GUI-based debuggers are front-ends, and may use the same underlying debugger. GUI-based debuggers are more convenient for tasks such as setting breakpoints and stepping through code. A GUI-based debugger requires little or no typing to operate. Text-based debuggers work well for debugging through a terminal or when simultaneously debugging multiple programs. (Matloff & Salzman 2008, 5, 10 – 11.)

A debugger allows the programmer to perform four main operations: step through the source code, inspect variables, set watchpoints, and move up and down the call stack. A program's execution can be paused at a specific line by inserting a breakpoint in the code. The program's execution can be continued one line at a time or until the next breakpoint is found. Variables can be inspected for their values when the program is stopped. Watchpoints are a combination of breakpoints and variable inspection. A variable can be marked with a watchpoint. If the variable changes, the debugger will pause the program. Watchpoints can be set based on conditional expressions. A function's runtime information is stored in a stack frame when the function is called. The function's local variables, parameters, and the location from which the function was called, are stored in the frame. A frame is created and pushed onto a stack for every function that is called. The system maintains the stack. A function's frame is popped off the stack when the function exits. The programmer can traverse and inspect the frames in the stack with a debugger. (Matloff & Salzman 2008, 14 – 18.)

3.3.2 GDB

GDB or the GNU Project Debugger is the most common debugging tool among Unix programmers. GDB is a text-based debugger, but many GUI-based front-ends exist for it. (Matloff & Salzman 2008, 3, 5.) GDB supports several programming languages, including C, C++, and Objective-C (Pesch, Shebs & Stallman 2017, 195).

The program to be debugged must request debugging information when it is compiled. The program cannot be debugged with GDB without the debugging information. GCC generates the debugging information with the "-g" option. The debugging information contains the data type of every variable and function. It also contains the correspondence between source line numbers and memory addresses in the executable code. (Pesch et al. 2017, 25.)

Figure 11 shows a simple GDB debugging session. The "gdb" command invokes GDB. The "file" commands sets chat as the program to be debugged. The "break" command inserts a breakpoint at line 23 inside a file called "Scene.cpp". The "run" command executes the program. GDB pauses the program when the breakpoint is hit. Full names of the commands have been used for added clarity. A command such as "break" could be substituted with "b" to reduce typing.

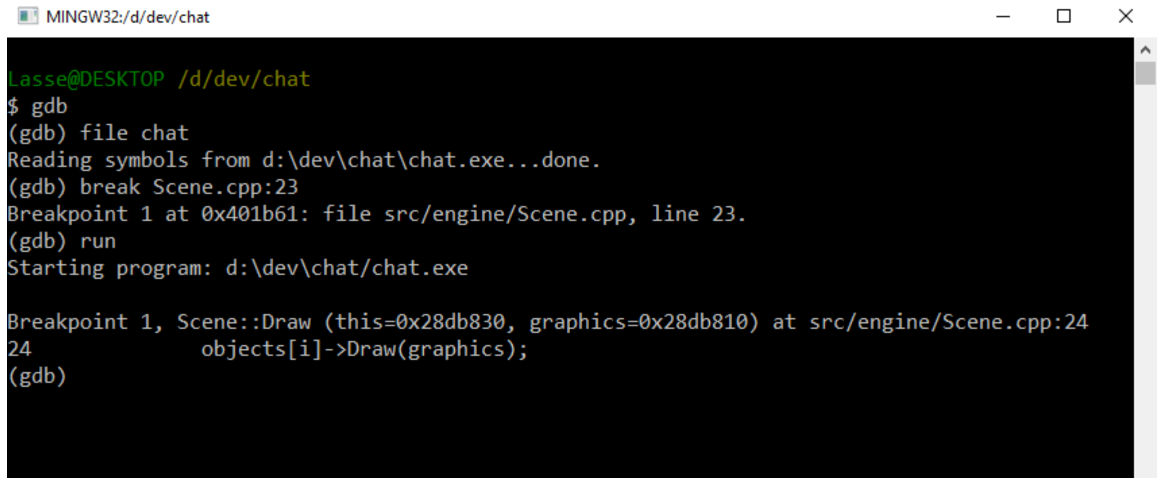

Figure 11. Debugging with GDB.

3.4 Libraries

Libraries allow features and functionality created by other people to be used when developing software. Many languages have their own standard libraries. A library must be linked to a program to be usable. A library is either static or dynamic. The primary difference is that a dynamic library is dynamically linked to the program by the operating system when the program is run. A dynamic library can be shared by multiple programs. (Shaw 2015, 160.) The C++ standard library and the SDL library are the most important libraries utilized in the development of the chat application. Both are addressed in this chapter.

3.4.1 C++ standard library

The ISO C++ standard specifies a set of components that are shipped with every C++ implementation. This collection of functionality is known as the C++ standard library. Bar performance, all implementations of the standard library must provide identical behavior. Use of the standard library over custom implementations is highly recommended. The C++ standard library is more portable and the resulting code more maintainable. The C++ standard

library aims to function as a common basis for other libraries and applications. (Stroustrup 2013, 859 - 861.)

The standard library is massive. Its specification in the ISO C++ standard is 785 pages long. It includes language features such as memory management and run-time type information, concurrent programming facilities, nonprimitive foundational facilities such as I/O streams, standard mathematical functions, random number generators, complex arithmetic, and regular expressions. STL or the standard template library is a major component of the C++ standard library. STL consists of iterators, containers, algorithms, and function objects. (Stroustrup 2013, 860, 885.)

3.4.2 SDL

SDL or the Simple DirectMedia Layer is a library that supports cross-platform development. It provides low-level access to audio, keyboard, mouse, joystick, and graphics hardware. It is used in projects such as emulators, games, and video playback software. SDL officially supports the Windows, Mac OS X, Linux, iOS, and Android operating systems. SDL is written in C and works natively with C++. Bindings are available for other languages such as C# and Python. SDL is distributed under a license that allows it to be used freely in any software. (SDL Wiki 2017.)

The chat application uses the SDL_ttf and SDL_net libraries in addition to the base SDL library. SDL_ttf is a text rendering library. It allows the use of TrueType fonts in SDL applications. SDL_net is a networking library. It makes network programming easier and more portable than what would be possible with plain sockets. The API for both libraries is available in their respective documentation files. (Atkins 2009a; Atkins 2009b.)

3.5 Source code editor

Editing source code is a core part of programming. A source code editor is the tool used to edit source code. Source code editors can be divided in two categories: IDEs or integrated development environments, and text editors. All IDEs include a text editor.

IDEs usually include a text editor, compiler, debugger, and a host of other tools. Visual Studio, Eclipse, and Xcode are commonly used IDEs. Text editors focus on fewer tasks. It is strongly recommended to use a text editor that supports programming related features, such as syntax highlighting and automatic formatting. Notepad++, Emacs, Sublime Text, and Vim are commonly used text editors. Text editors usually support additional features via plugins.

A source code editor is mostly a personal choice. Source code editors vary in platform support, programming language support, general feature support, cost, popularity, activity of the surrounding community, customizability, performance, resource usage, and learning curve. A source code editor should be learned well enough to be able to work efficiently.

The Vim text editor was used to develop the chat application. Vim stands for "vi improved", because it is based on an earlier text editor called vi (Hannah, Lamb & Robbins 2008, 145). Vim is almost universally available and highly customizable (Moolenaar 2017).

3.6 Build automation tool

A build refers to the process that assembles files and other assets into a software product. The process may include compiling source files, packaging compiled files, producing installers, and modifying databases. A build automation tool automates these steps. The build can then be repeated at any time without direct human intervention. Build automation eliminates defects that result from the variation often present in a manual build process. (Agile Alliance 2017.)

The GNU Make build automation tool was used to automate the chat application's build process. Make automatically figures out the parts of a program that need to be recompiled. Invoking the Make utility causes it to run the commands required to recompile the files. The Make utility can be used with any programming language. The only requirement is that the language's compiler is executable with a shell command. Make is not limited to programming related files. It can be used to update any files from other files whenever the others change. (McGrath, Smith & Stallman 2016, 1.)

A makefile must be written to use Make. A makefile describes relationships between files and the commands that update the files. Make updates files based on the makefile database and the time the files were last modified. Make can be invoked with additional command-line

arguments to control which files are updated and how. A makefile's syntax can seem arcane, but the basics are simple. The main component of a makefile is a "rule". A rule consists of a target, a prerequisite, and a recipe. A rule describes how and when its target should be updated. The target is typically the name of the file that is generated, which is often an executable file or an object file. The prerequisite is the file or files that the target depends on. If the prerequisite file has changed, the target must be updated. The recipe consists of the commands that are carried out in the case that the target must be updated. (McGrath et al. 2016, 1, 3.)

3.7 Version control system

A version control system or VCS keeps track of changes to files over time. Various operations can be performed on the files that are tracked. A version control system can revert files or an entire project back to an earlier state. It can compare changes that have been introduced over time. It can also tell who last modified a certain file. (Chacon & Straub 2014, 27.)

A version control system is either local, centralized, or distributed. A local version control system keeps track of changes on the local computer. A centralized version control system stores all the versioned files on a single server. Multiple users can check out files from the server and commit changes to the server. The main issue with a centralized version control system is that the server represents a single point of failure. If the server goes down, nobody can collaborate or save changes to their files during that time. Proper backups must be maintained in case the server's hard disks fail. A distributed version control system is more reliable and flexible, because every client fully mirrors the repository. If a server dies, the repository on any client can be used to restore it. (Chacon & Straub 2014, 27 - 30.)

Git was used as the version control system for the chat application. The finished project is hosted in GitHub. Git is a distributed version control system that was developed for the development of the Linux kernel. It is fast, efficient with large projects, and provides a great branching system for non-linear development. (Chacon & Straub 2014, 31.) GitHub is a popular host for Git repositories. It supports issue tracking, code reviews, and many other features. (Chacon & Straub 2014, 195.)

3.8 Modeling language

A modeling language is used in the design of software. A modeling language depicts the elements that make up a program at a higher-level of abstraction than a programming language. (Fowler 2003, 1.)

UML or the unified modeling language is a collection of graphical notations that help in designing and describing software systems. It is particularly useful for software systems that use the object-oriented programming style. UML has existed since 1997 and its standard is controlled by the Object Management Group. (Fowler 2003, 1.)

UML is used to sketch or to blueprint a software system. Sketching is selective, informal, and dynamic. Sketching is useful when the focus is on communication and not on completeness. Blueprinting is about completeness. Blueprinting shows every detail of a system so that it can be easily implemented or understood. A system is either forward engineered or reverse engineered. Forward engineering draws a UML diagram before any code is written. Reverse engineering draws a UML diagram from existing code. (Fowler 2003, 2.)

UML consists of many types of diagrams: class diagrams, sequence diagrams, object diagrams, package diagrams, deployment diagrams, and many others (Fowler 2003, xxvii). Class diagrams are the type relevant to the thesis. A class diagram describes a system's classes and their relationships. (Fowler 2003, 35).

Figure 12 shows the UML notation used in the practical part of the thesis. A rectangle represents a class. Class A is associated with class B. Association is denoted with a line with no arrows. The asterisk next to class B indicates that class A is associated with any number of class B instances. This is called the association's multiplicity. Multiplicity is marked with a number placed at either or both ends of the line. The default multiplicity is 1. A multiplicity of 1 is explicitly stated when it is of importance. Class D inherits from class C. An empty arrow denotes generalization. Generalization refers to inheritance in a programming context. Class C is an abstract class. An abstract class can't be used to instantiate objects. The name of an abstract class is italicized. (Fowler 2003, 35, 37 – 38, 41, 45, 69).

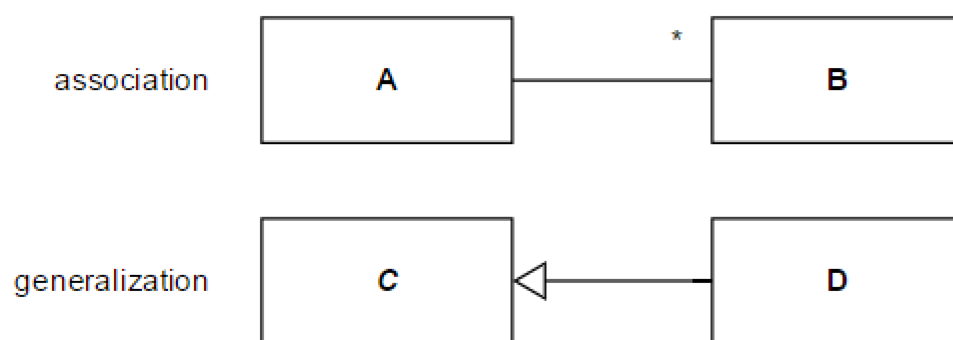

Figure 12. UML notation: association, multiplicity, and generalization.

4 CHAT APPLICATION

This chapter covers the chat application's development. The planning subchapter gives an overview of the application, describes the application's requirements, and shows the application's user interface. The development environment is briefly discussed, followed by a look at the coding conventions used in the source code. The finished program's structure and implementation is thoroughly examined. The program's build system and testing procedures are explored. The finished program's usage is demonstrated. The chapter ends with an analysis of the project.

4.1 Planning

The program was developed in a relatively free-form manner. The program's structure and functionality were planned informally. The plan originally included a more formal approach to scheduling and task management. This proved out to be unnecessary overhead for such a small single-person project.

4.1.1 Overview

The program uses the client-server application architecture. The program can operate as a client or a server. Client and server functionality are included in the same executable file. The program is operated with commands that are issued through the same input box that is used to type messages. The program has a simple single-view user interface, that consists of a chat window and an input box. Additional information is displayed in the application window's title bar. A client can send messages. The server forwards the messages it receives to other connected clients.

4.1.2 Requirements

Requirements are the features that a program should provide. Functional requirements are things the application should do. Nonfunctional requirements describe the application's behavior or constraints, such as performance, reliability, and security characteristics. (Stephens 2015, 54, 63.) An example of a functional requirement is "Allow the user to clear the chat window." An example of a nonfunctional requirement is "The server must handle the load of up to 50 client connections."

The chat application's requirements were selected with the goal of having a reasonable project both in scope and in difficulty. The requirements fall mostly to the functional category. Because the application has no significant nonfunctional requirements, all requirements have been combined into a single category.

The requirements are as follows:

1. Allow the user to control the application by issuing commands. The following commands are supported: connect, disconnect, host, shutdown, clear, quit, and help. Commands are prefixed with a forward slash, as in "/connect".

2. Allow the user to connect to a server and to disconnect from a server. To connect, an IP address, a port number, and a username must be specified. A client can only be connected to a single server at a time.

    - /connect IP PORT USERNAME

    - /disconnect

3. Allow the user to send messages and have the server relay those messages to all other users connected to the same server. Message length is limited to a single line.

4. Display the following information in the title bar of the application window: the application's name, the username selected by the user, the IP address of the server connected to, and whether the program is currently hosting a server.

5. Provide built-in help and instructions. Provide a help message for every individual command.

- /help COMMAND

6. Have the application support at least the Windows 10 operating system.
7. Allow the user to clear the chat window.

    - /clear

8. Provide support for the English alphabet, lower- and uppercase characters, numbers, and the most common special characters, such as period, comma, exclamation mark, and question mark. The forward slash character must also be supported, as it is used to issue commands.
9. Allow the user to quit the application.

    - /quit

10. Have the server broadcast notifications to all users connected to the server whenever another user joins or leaves the server.
11. Allow the user to host a server. Allow the user to stop hosting the server. The program should be able to connect to itself as a client if it is hosting a server.

    - /host PORT
    - /shutdown

4.1.3 User interface

Figure 13 shows the application's user interface. It consists of a title bar, a chat window, and an input box. The title bar displays general information such as the IP address of the server currently joined to. The input box is used to type messages and issue commands. The chat window displays messages. New messages are inserted at the bottom of the chat window. Messages flow towards the top of the chat window. A message is prefixed with the username of the client that sent the message. Automatically broadcasted messages are prefixed with a hash sign instead of a username.

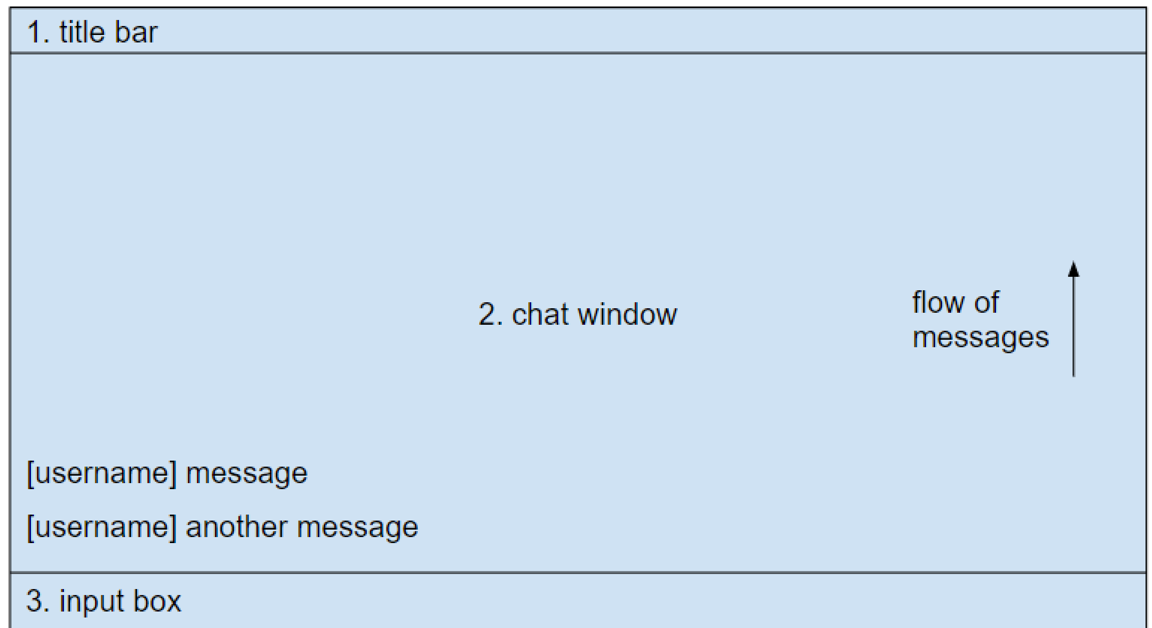

Figure 13. The chat application's user interface.

The user interface has no buttons or other interactive elements. The input box automatically accepts text input when the application window is focused. The input box has a blinking cursor to indicate that it is accepting input. The title bar is part of the standard application window. The user interface uses a monospaced font. Each character in a monospaced font occupies the same amount of horizontal space. This helps in specifying the maximum length of a message, because the program limits the length of a message to a single line. If the characters varied in width, a message consisting of certain letters could cross the edge of the chat window.

4.2 Development environment

The development environment and its full stack of tools consists of C++, SDL, Vim, GCC, GDB, GNU Make, Git, and GitHub. The MinGW or Minimalist GNU for Windows environment was also used. Chapter three described the tools in more detail. Figure 14 shows a screen capture of the development environment. The environment consists of a terminal and a customized Vim text editor placed side-by-side. The rest of the tools are used from the terminal. The tools and technologies were selected due to personal interest.

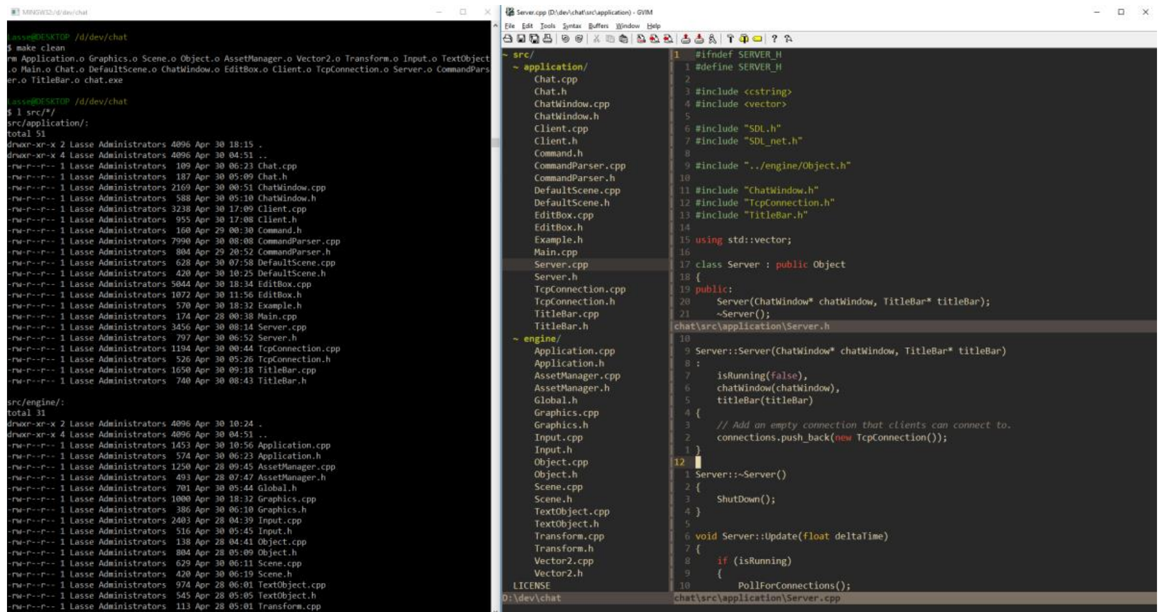

Figure 14. The development environment.

4.3 Coding conventions

The source code follows certain conventions to keep it as readable as possible. The most important conventions concern the naming of variables, functions, and classes. The goal is to format everything uniformly. This includes elements such as names, spaces, indents, comments, and parentheses and brackets.

Figure 15 shows a made-up class header file that illustrates some of the conventions used in the code. The header file begins with include guards. Other header files are included in the following order: C++ standard library headers, SDL headers, engine headers, and application headers. Headers are sorted alphabetically within their individual categories. Namespace directives are placed after the include directives. Functions and classes follow the PascalCase or upper camel case naming practice, where the first letter of each word in a name is capitalized. Variable names follow the lower camel case practice, where the first letter of each word in a name is capitalized, excluding the first word. Member variables and functions are placed in the following order: constructor, destructor, other functions, getters, setters, and variables. The protected and private sections follow the same order. These rules do not cover every situation. Other conventions can be deduced by reading the source code.

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

// Include directives.
// (All alphabetically within their categories.)

// C++ standard library headers.

// SDL headers.

// Engine headers.

// Application headers.

// Namespace directives.

class ClassName
{
public:
    // Constructor.
    // Destructor.

    // Other functions.
    void FunctionName();

    // Getters.
    // Setters.
    int GetVariable();
    void SetVariable();

    // Variables.
    int variableName;

protected:
private:
};

#endif
```

Figure 15. Coding conventions used in the source code.

4.4 Program structure

The program is written "from scratch" with the help of the SDL library. The starting point doesn't even include an empty application window. Everything starting from the main loop to object management and text rendering had to be implemented. The codebase is divided in two parts. The first part is an engine, which contains general functionality not specific to the chat application. The second part contains the application specific code, which implements the features unique to the chat application.

4.4.1 Engine

The application has a custom engine. The engine implements many necessary features such as input handling, text rendering, and scene and object management. Figure 16 shows a UML diagram of the engine's structure. The UML diagrams that depict the source code are reverse-engineered sketches. Their purpose is to communicate the high-level structure of the program. Implementation details are viewable in the source code. The introductory chapter provides a link to the source code.

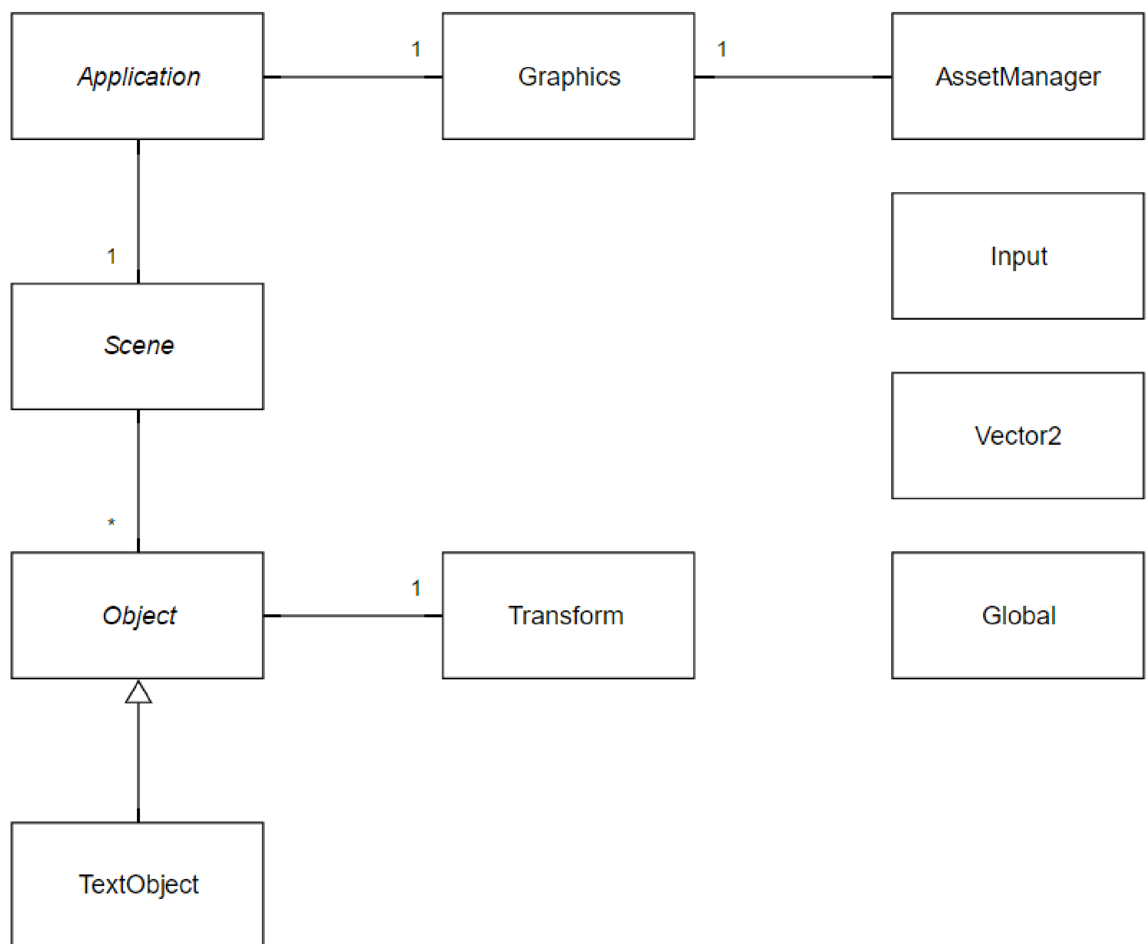

Figure 16. UML diagram of the engine's structure.

Application is an abstract class that serves as a container for the whole application. It initializes SDL. The application class is associated with a single scene and a single graphics object. The program's main loop is implemented in the application class. The main loop keeps the program running. The application class draws and updates the scene, and polls for user input.

The graphics class creates the application window and a renderer. The renderer's main task is to draw objects. The renderer is set to restrict the program's frame rate to the refresh rate of the computer monitor. Every drawable object requires access to an instance of the graphics class. The application, scene, and object classes all have a draw and an update function. The graphics object is passed as an argument to the draw function, which propagates access to the object through the system.

Scene is an abstract class that serves as a container for objects. The scene class draws and updates the objects on every iteration of the main loop. A scene can be thought of as a single level in a game or a view in a more traditional application. The chat application has just a single scene.

Object is an abstract class that functions as the base class for most of the concrete classes in the application code. The object class contains a draw and an update function, and a single transform object. Transform is a simple class with two member variables: position and scale. The position and scale variables' type is vector2. The vector2 class represents a two-dimensional vector.

Text object is a concrete class that inherits from the object class. The text object class holds and draws a piece of text; a string. The string is transformed into a texture, which is drawn on the screen.

Asset manager is a static class that loads font files and transforms strings into textures. The chat program uses a single font, which is loaded when the program is launched. The class combines a string and a font file into an SDL surface. The surface is transformed into an SDL texture, which is drawn on the screen. The texture must be recreated every time the rendered text changes.

Input is a static class that keeps track of which keys are held down. Other classes query the input class for this information. The input class' state is updated every frame. The implementation allows only a single key to be recognized as being held down per frame. The shift key is updated separately to enable upper case and special characters.

Global is a header file that contains global variables, macros, and type definitions. Global variables are limited to the width and height of the application window. The macros

implement functionality such as safe deletion of objects, printing of error messages with line numbers, and checking SDL function calls for errors.

4.4.2 Application

The application code consists of the code specific to the chat's functionality. Figure 17 shows a UML diagram of the application code's structure. The three classes at the top of the diagram are part of the engine. The rest of the classes belong to the application code. The client, server, chat window, edit box, and title bar classes inherit from the object class.

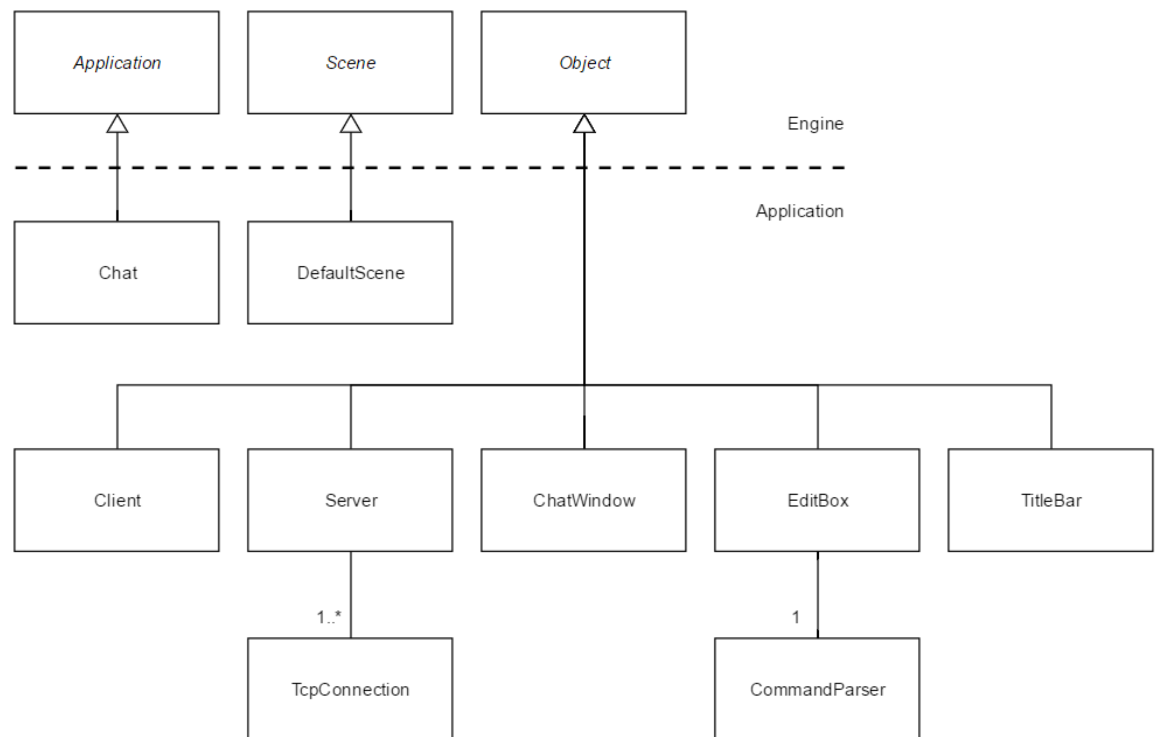

Figure 17. UML diagram of the application code's structure.

The chat class is a formality. It has little added functionality over its base class. The only extra task it takes care of is instantiating the application's single scene. The point is to make a distinction between the engine and the code specific to the application.

The default scene class is a type of scene. It is the only scene in the program. It instantiates all the objects that make up the scene, and adds them to its internal container. One instance of

each of the following classes is added: client, server, chat window, edit box, title bar, and command parser.

The client class allows the program to connect to and disconnect from a server. It polls for incoming data and sends messages to the server. Incoming data is parsed and added to the chat window as messages. The client opens a TCP socket when it connects to a server. The socket is added to a socket set. The socket set is queried for activity when polling for incoming data. The client notifies other clients with an automatic broadcasted message when it connects to or disconnects from the server.

The server class keeps track of client connections. It forwards messages received from one client to all other connected clients. The server polls for connection requests and incoming data from clients. The TCP connection class represents a connection. The server is associated with 1 or more TCP connection instances. The server always maintains one unconnected instance, which is used when the server polls for connection requests.

The chat window class holds and displays messages. The messages are instances of the text object class. The chat window class has functions for clearing the chat window and adding a new message. Clearing the chat window removes all messages. The chat window is simply a rectangle with visible edges and a transparent center.

The edit box class implements a field that accepts text input. It uses the input class to determine which keys are held down, and edits its internal text string accordingly. Pressing the enter key causes the text to be sent to an instance of the command parser class. The command parser class parses the text and evaluates whether it is a message or a command. A message is added to the chat window and sent to the server. A command is further processed by extracting the command and its arguments from the text. The appropriate functions are called to execute the command.

The title bar class' responsibility is to update the application window's title bar. Other classes call the title bar class' functions to update its information. The title bar displays the application's name, connection status, hosting status, and the user's username. If the program is connected to a server, the server's IP address and port number are displayed. If the program is hosting a server, the port number to which the server accepts connections is displayed.

4.5 Build system

The program's build process is automated with GNU Make. The Make utility must still be manually executed, but the detailed instructions do not have to be repeated. A makefile contains the logic required to carry out the build. The makefile is available in the source code repository. The makefile is not optimized or sophisticated. If a new file is added to the project, its name must be manually inserted in the makefile. Object files and the executable file are generated in the project's root directory, instead of separate folders.

The makefile has three rules: all, clean, and run. The "all" rule recompiles the files that need to be regenerated in case changes have been introduced. The "clean" rule removes all object files and the executable file. If the "clean" rule is run first, the "all" rule will recompile the whole project. The "run" rule executes the program. The Make utility can be invoked with any combination of rules. The command "make clean all run" would remove all generated files, recompile the project, and run the program. The makefile specifies the compiler to be used, the compiler's options, the name of the generated program, and the required libraries and their locations.

4.6 Testing

The application's functionality was verified locally on the development computer as features were implemented. The application's network functionality was tested between three separate computers: a desktop computer and a laptop in the same private network, and a virtual machine running in the AWS cloud service. The only scenario where things did not work was when the virtual machine tried to connect to a server running in the private network. A connection could not be established because the computer hosting the server was behind NAT in a private network. Chapter 2.5 describes this problem. A connection could be formed once port forwarding had been configured in the network's router.

The command system validates user input successfully. An error message is added to the chat window indicating the problem in case invalid input is received. The program freezes for around 20 seconds if the connect command receives valid input, but a connection can't be initiated. The function provided by SDL_net that opens a TCP connection does not have a

timeout parameter. A possible workaround would involve probing the existence of the remote server with a UDP packet. The TCP connection would be opened if a response is received within a certain time frame. The port parameter of the host and connect commands is successfully parsed, even if the port number is suffixed with letters. This behavior is accidental, but does not cause problems.

The application window is scalable in size. Scaling the window has the side effect of reducing the font's sharpness. The effect is more pronounced when reducing the window's size. The chat window's and edit box's edges disappear or are not fully rendered with certain window dimensions. The scaling functionality is the default behavior provided by SDL. The topic should be explored more to address the problems.

All testing was performed manually. This is not optimal. Automation would have greatly improved the speed and reliability of the testing. Debugging and troubleshooting a networked program is particularly tedious and error-prone. At least two instances of the program must be launched. Certain problems may only manifest under very specific conditions, such as when connecting or disconnecting multiple instances of the program in a particular way.

The program was not analyzed with a memory profiler. The program's memory usage remained stable after hours of use based on the Windows task manager. The program should not have any substantial memory leaks. Functions have been optimized where possible to not contain blatantly inefficient solutions.

4.7 Demonstration

The final product is a functional client-server chat application that implements the functionality specified in the requirements. This chapter demonstrates the program's functionality in practice.

Figure 18 shows a screen capture of the finished program. The program displays general help information when it is launched. The help information lists the supported commands, the generic format of commands, and how to get help for specific commands. The program is not connected nor hosting a server. Messages can be typed and sent, but they are only displayed locally. Automatic messages sent by the program are prefixed with a hash sign.

Figure 18. Screen capture of the finished client-server chat application.

The program connects to a server with the connect command. Figure 19 shows an example of the command being run. The command is successfully executed and the program connects to a server. Other clients are notified of the new client's connection and username. The server displays the connected client's IP address and port number locally.

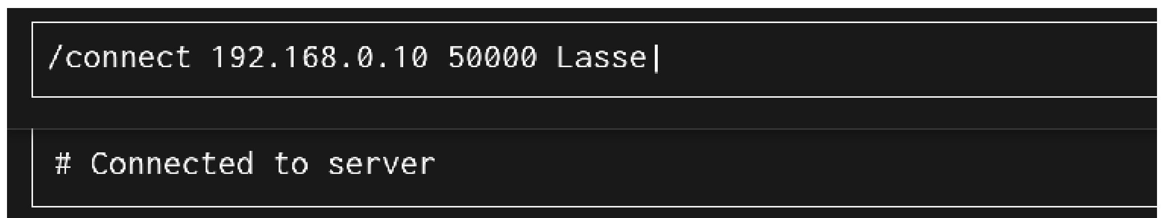

Figure 19. Running the connect command.

The application window's title bar is updated to reflect the change in connection status. Figure 20 shows a close-up of the application window's title bar before and after connecting to the server. The lowest title bar displays the server's status.

Figure 20. State of the client's title bar before and after connecting, and the server's title bar.

The help command can be used to display help information for specific commands. Running the command "/help connect" displays the message in Figure 21. The help command lists the connect command's functionality and format. It also shows an example of running the command and any other relevant information, such as the command's abbreviation.

```
# Command "/connect" connects to a server
# Format: "/connect [ip] [port] [username]"
# Example: "/connect 192.168.0.12 50000 Lasse"
# Abbreviation: "/c". Addresses 127.0.0.1 and localhost are interchangeable
```

Figure 21. Help information for the connect command.

4.8 Analysis

The project is overall successful. The chat application supports the planned functionality. Some key features that are expected of a chat application, such as a scroll bar, were left out. The project should have been planned more carefully.

The source code's quality is acceptable. It adheres to the specified coding conventions with good consistency. The style of C++ used is a little outdated. Many modern features such as smart pointers are not utilized. C++ best practices such as const correctness are not consistently followed. Some sections of the code are not as clear as they could be. Substantial parts of the code would have to be restructured to make improvements. The code remains as it is due to time constraints. The code contains some literal values that should be moved into variables, such as the text and background color values.

Messages, commands, and object references are systems that should be rewritten. Messages sent by the program and not the user are scattered all over the code. Making changes to their format is difficult and time consuming. Commands could be abstracted out as their own class

or classes, which would contain the parsing, messages, and other functionality related to a command. Message formatting should be moved to a single location. References to objects within a scene are passed as arguments through constructors. This is manageable for a program as simple as this, but a total mess for anything larger. Objects in a scene should be searchable.

A myriad of features could be implemented in a chat application. The most important ones that were left out are a scroll bar, channels, and full keyboard and localization support. The program's message history is limited to 17 messages. A scroll bar would allow the user to browse through the full message history. Every message is sent to every user connected to a server. Channels would allow the user to connect to a smaller group within the server. Access could be restricted with passwords. The input box supports a limited set of characters. It is easy to add support for additional special characters. It is harder to add support for characters that are not part of the English alphabet.

The program can be distributed in a folder with the required DLLs and font files, and it will run. Installers and distribution are topics that were not considered. Compilation was not attempted on platforms other than Windows.

5 CONCLUSION

The goal of the thesis was to develop a client-server chat application. The goal was met. The final product is a functional client-server chat application that successfully implements the features laid out in the requirements. The thesis gives a decent example of a small software development project. The application itself has no planned use outside of the thesis.

The underlying theory and tools were introduced successfully. The development process was described in a reasonably detailed manner. The program's high-level structure and functionality were explained and illustrated with diagrams. The program was tested and contains no obvious or substantial flaws. The program's basic functionality was demonstrated.

The development process and the program's structure should have been planned more carefully. Some features were not implemented optimally. Planning was difficult in part because of the concurrent writing of the thesis paper.

The source code is acceptable in quality. The source code's strong points and shortcomings were analyzed. The coding conventions used in the source code were explored. The project's source code repository has been made available as part of the thesis.

The application's future development remains a question. Parts of the code will be used in other projects. The project repository will not be altered, because the thesis paper depends on its current state. Any future development will happen in another repository.

REFERENCES

Agile Alliance. (2017). Automated Build. Retrieved 2017-04-09 from https://www.agilealliance.org/glossary/automated-build/.

Aho, A., Lam, M., Sethi, R. & Ullman, J. (2013). Compilers: Principles, Techniques, and Tools. England: Pearson Education Limited.

Atkins, J. (2009a). SDL_net. Retrieved 2017-04-07 from https://www.libsdl.org/projects/SDL_net/docs/SDL_net.pdf.

Atkins, J. (2009b). SDL_ttf. Retrieved 2017-04-07 from https://www.libsdl.org/projects/SDL_ttf/docs/SDL_ttf.pdf.

Chacon, S. & Straub, B. (2014). Pro Git (2nd Edition). Apress.

Demiris, Y. (2012). Language Processors (E2.15) Lecture 1: Introduction and Overview. (Imperial College London). Retrieved 2017-04-26 from http://www.iis.ee.ic.ac.uk/yiannis/lp/LPLecture1bw.pdf.

Fowler, M. (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition). USA: Addison-Wesley.

Glazer, J. & Madhav, S. (2015). Multiplayer Game Programming: Architecting Networked Games. USA: Addison-Wesley.

Hannah, E., Lamb, L. & Robbins, A. (2008). Learning the vi and Vim Editors (7th Edition). California, USA: O'Reilly Media.

Kurose, J. & Ross, K. (2012). Computer Networking: A Top-Down Approach (6th Edition). USA: Pearson.

Matloff, N. & Salzman, P. (2008). The Art of Debugging with GDB, DDD, and Eclipse. USA: No Starch Press.

McGrath, R., Smith, P. & Stallman, R. (2016). GNU Make: A Program for Directing Recompilation (GNU Make Version 4.2). Boston, USA: Free Software Foundation.

Microsoft Official Academic Course. (2011). Networking Fundamentals, Exam 98 - 366. USA: Wiley.

Moolenaar, B. (2017). Vim - the ubiquitous text editor. Retrieved 2017-04-26 from http://www.vim.org/.

Pesch, R., Shebs, S. & Stallman, R. (2017). Debugging with GDB: The GNU Source-Level Debugger (10th Edition, for GDB version 7.12.50.20170405-git). Boston, USA: Free Software Foundation.

Rosenberg, J. (1996). How Debuggers Work: Algorithms, Data Structures, and Architecture. USA: Wiley.

Scott, M. (2009). Programming Language Pragmatics (3rd Edition). USA: Morgan Kaufmann.

SDL Wiki. (2017) Introduction to SDL 2.0. Retrieved 2017-04-07 from http://wiki.libsdl.org/Introduction.

Shaw, Z. (2015). Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C). USA: Addison-Wesley.

Stallman, R. (2016). Using the GNU Compiler Collection (for GCC Version 6.3.0). Boston, USA: GNU Press.

Stephens, R. (2015). Beginning Software Engineering. Indiana, USA: Wiley.

Stroustrup, B. (2013). The C++ Programming Language (4th Edition). USA: Addison-Wesley.

Tate, B. (2010). Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. USA: Pragmatic Programmers.