BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

AN INTEGRATION OF SIP VOICE CALLS INTO AN IRC CLIENT OR GATEWAY

INTEGRACE SIP HLASOVÉHO VOLÁNÍ DO IRC KLIENTA ČI BRÁNY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DAVID KOCMAN**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

140536

Institut:	Department of Information Systems (UIFS)
Student:	**Kocman David**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**An Integration of SIP Voice Calls into an IRC Client or Gateway**
Category:	Networking
Academic year:	2022/23

Assignment:

1. Familiarize yourself with IRC technology and the SIP protocol and its use for voice calls. Explore the cross-platform solutions available to support IRC and SIP on client devices. Focus especially on WeeChat, Bitlbee, Linphone, and Baresip.
2. Design a way to integrate voice calls using SIP into IRC communication on the gateway or client applications. Choose the appropriate technologies.
3. After consultation with the supervisor, implement the support for SIP calls to the selected IRC clients or gateways according to the design. Also design and implement automated acceptance tests.
4. Thoroughly test the solution, evaluate it against other possible solutions and publish it as open-source.

Literature:

- Flanagan, William A. VoIP and unified communications: internet telephony and the future voice network. Wiley, 2012. ISBN 978-1-118-01921-4
- Belledonne Communications SARL. Linphone [online]. 2021 [cit. 2021-08-18]. Available at https://www.linphone.org/technical-corner/linphone
- Baresip Foundation. Baresip [online]. 2021 [cit. 2021-08-18]. Available at https://github.com/baresip/baresip

Requirements for the semestral defence:
Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	24.10.2022

Abstract

This thesis describes the design, implementation and testing of a Session Initiation Protocol user agent, using an Internet Relay Chat client or gate as its graphical interface. A third-party open-source library, called liblinphone, is used for call related implementation and the application itself is written in the C/C++ language. The program can make calls and other basic SIP-related features, which include proxy registration, ENUM lookup, and instant messaging. An address book is also available for storing contacts and identities, and is implemented with the SQLite3 C/C++ library. The result of this thesis is an ability to make calls with IRC.

Abstrakt

Tato práce popisuje návrh, implementaci a testování Session Initiation Protocol uživatelského agenta, který používá Internet Relay Chat klienta či bránu jako jeho grafické rozhraní. Pro implementaci volání je použita open-source knihovna třetí strany, nazývána liblinphone, a samotný program je napsán v jazyce C/C++. Program je schopen jak volání, tak i základních SIP vlastností, jako je registrace u ústředny, překlad čísel na adresy pomocí ENUM a přímé zprávy. Také je k dispozici adresář pro ukládání kontaktů a identit, napsán pomocí knihovny SQLite3 pro C/C++. Výsledek této práce zavádí možnost volání z IRC.

Keywords

SIP, RTP, IRC, VoIP, C, C++, SIP user agent, calls, SQLite3, database, integration

Klíčová slova

SIP, RTP, IRC, VoIP, C, C++, SIP user agent, volání, SQLite3, databáze, integrace

Reference

KOCMAN, David. *An Integration of SIP Voice Calls into an IRC Client or Gateway.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

An Integration of SIP Voice Calls into an IRC Client or Gateway

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. RNDr. Marek Rychlý Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
David Kocman
April 27, 2023

Acknowledgements

I want to thank my supervisor RNDr. Marek Rychlý Ph.D. for his willingness, feedback, consultations and professional help during the making of this thesis. I also want to thank my family for their support during my studies at BUT FIT.

Contents

List of Figures

Chapter 1

Introduction

People communicated with each other since the beginning, either face to face or via letters. With the invention of the telephone in the 19th century and the internet in the 20th century, communication became much simpler and faster. Nowadays we can call or text another person with just a click of a button because internet connection is almost ubiquitous.

With the Internet also came quite a few call-oriented and message-oriented protocols, such as the Voice over Internet Protocol (VoIP) and Internet Relay Chat (IRC). VoIP enables voice and video calls to use internet infrastructure as their medium with the help of numerous signaling and transport protocols, such as Session Initiation Protocol (SIP). With IRC, users are able to send relayed text messages to each other and talk about various topics together.

This thesis combines these two technologies together and aims to create a program that is able to make calls with the help of an IRC client or gate interface. The result behaves as a full-fledged SIP user agent and is available as open-source software on GitHub[1].

Chapter 2 contains a basic theory of used technologies, mainly Session Initiation Protocol and Internet Relay Chat. It also takes a look at some of the applications that were researched for this thesis.

The summary of the program's requirements is described in Chapter 3 and the design of the architecture can be found in Chapter 4. The main concept, used libraries, and program specifications are described here.

Implementation of the architecture is described in Chapter 5, mainly the usage of said third-party libraries and different original algorithms.

Chapter 6 talks about the testing of the system, the design of the test script, and a comparison with different contemporary SIP user agents.

Chapter 7 contains the conclusion to this thesis, a description of achieved goals, and some ideas for future extensions of this system.

[1] https://github.com/DavidKocman36/IRCPhone

Chapter 2

Analysis of technologies

This chapter takes a look at the technologies and protocols used in this thesis. The first section (2.1) talks about the Session Initiation Protocol, its architecture, the format of the messages, and other SIP-related technologies. In Section 2.2 the Real-Time Transport Protocol for transmitting media is described. Section 2.3 talks about Internet Relay Chat. The last section (2.4) takes a look at some of the SIP and IRC applications, that were researched for this thesis.

2.1 Session Initiation Protocol

When two parties want to exchange data between each other, a session has to be created. Though sometimes it is hard to locate the second participant, some protocols were made for this specific reason.

The Session Initiation Protocol, SIP for short, is an application-layer signaling protocol for IP telephony. It can establish, modify and terminate multimedia sessions and helps these two parties to agree on session characterization with Session Description Protocol, talked about in Section 2.1.3. It is based on a request-response model, which means each transaction or method shall have at least one response [16, p. 8–11].

It was created in the early 2000s by Internet Engineering Task Force (IETF). SIP messages are text-based [16, p. 26] and consist of human-readable text tags, called headers, and are similar to markup languages such as HTML [8, p. 80]. They were designed to minimize the number of messages exchanged during the initiation phase as opposed to H.323[1] for example. Media parameters are sent in the text fields of INVITE and OK methods instead of separate packets [8, p. 86].

SIP is completely independent as an application but it is always used with other protocols to make the architecture work. At the transport layer, it is transported either by Transmission Control Protocol or User Datagram Protocol, depending on the circumstances. For delivering media and QoS monitoring, the Real-Time Transport Protocol is used, more on this in Section 2.2 [16, p. 9] [8, p. 81].

Each user is identified with a unique address, called “SIP URI”, or Uniform Resource Identifier. It is an address that contains enough information for establishing a session with the other end. It is very similar to an E-Mail address and its general format looks like this:

```
[sip or sips]:user:password@[host name, FQDN or IP address]:port
;uri-parameters?headers
```

[1] https://www.techopedia.com/definition/4478/h323

The `user:password@` part might be omitted if no service is associated with a user. URIs also support escaping special characters, using % sign and hexadecimal digits of its ASCII designation. As a separator, & is used [8, p. 80–81].

2.1.1 Architecture and signaling

SIP architecture consists of either hardware or software elements that communicate with each other. These include [8, p. 81–82]:

- **User Agent (UA)**: an end point, able to send requests (also known as User Agent Client – UAC) and receive responses (User Agent Server – UAS). User Agent usually acts as both client and server and some examples are – IP phone, softphone, and camera.
- **UA proxy**: a server that acts as an intermediary for receiving and sending messages to other UAs or proxies. They provide location services when initiating a call and their domains are stored in the DNS SRV resource record for callers to know where to send call requests.
- **Location Service**: a database in which a specific called UA can be found in that domain. The database gets its information from the registrar server to which different UAs are registering.
- **Registrar**: a server or process to which a UA sends its location or address and lets the server know that it is ready to receive requests.
- **Redirect server**: a proxy that tells UA or proxy server where to send messages with the help of location service.

As stated before, SIP follows a request-response model, where one UA takes the role of client (UAC) and sends a request to another UA in the role of server (UAS). The role of one UA fluctuates from client to server depending on circumstances.

The requests and responses are called "SIP methods" [8, p. 82] or "SIP messages" [16, p. 26]. They are used for initiating, managing, and terminating sessions (media connection). Procedures for sending messages differ from when dialog (communication between two UAs) does not exist from when one does, mainly in the fact that both UAs know each other's Tags. Each dialog is identified with this Tag and a Call-ID value [8, p. 82].

Call initiation

Figure 2.1 shows an example of session initiation and key components. A session is initiated with an INVITE method which is the request from a UA client (caller) [8, p. 82]. INVITE has attributes that mainly contain source address, destination address, and information about the session from the caller's perspective [16, p. 11].

If the OK message takes over 200ms to deliver, the progress and status (Trying) messages are sent to the caller. The OK message confirms the connection to the caller and the ACK message confirms to the callee that the connection exists, completing the three-way handshake. The session is now established and media transport may commence, which is logically separated from the session initiation. The dialog and session terminate when one of the parties sends a BYE method which ends the call [8, p. 82–85].

The most common SIP messages, described closely in 2.1.2 [8, p. 83]:

- INVITE: initiation of a dialog and a session.
- OK: confirmation of a request. Content depends on the type of request [16, p. 183].
- ACK: confirming the connection from caller to callee.
- BYE: closes a connection, session, and dialog. Always sent from within the dialog, else CANCEL is sent.
- REGISTER: transmission of a location to the desired proxy.
- CANCEL: cancel an outgoing INVITE. After a connection is established, BYE is used.
- MESSAGE: used for an exchange of content in real time, especially text messages [18, p. 2].

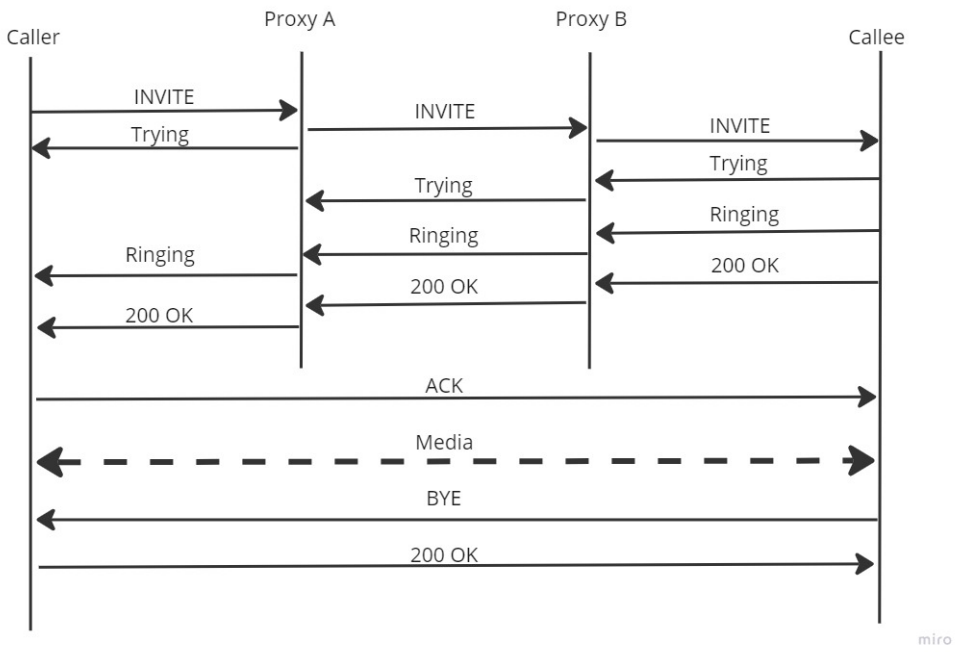

Figure 2.1: A diagram showing basic SIP signalization and key components. Adapted from [8, p. 84].

Locating the callee

SIP also uses additional network services, such as DHCP and DNS servers. DNS servers play a huge role in SIP signaling as they are able to help locate proxies and users. For a user to locate other, an infrastructure has to be set up, such as [8, p. 87]:

- Each user has a unique SIP URI.

- The callee must be registered to a reachable proxy server with a findable IP address. This registration might be permanent or temporary.
- The called proxy must be able to find the current callee's IP address.

Figure 2.2 illustrates the SIP registration process and subsequent localization of a callee. One UA with the URI `sip:adam@example.com` registers (1) to `example` registrar which stores this entry to proxy's location service (2). Our second participant with the URI `sip:david@something.com` sends the INVITE message (3) to his local proxy with `sip: adam@example.com` as the destination address. David's proxy with the help of DNS resolution finds the address of the authoritative proxy for Adam's Address Of Record (adam@example.com). The query asks specifically for the SIP server for the domain `sip.example.com`. The proxy then sends the invite to Adam's proxy (4). When Adam's proxy receives the INVITE method it has to query (5) its location service and receive (6) the registered address or IP address associated with `sip:adam@example.com`. Adam's proxy then forwards the INVITE method to the registered phone (7) [8, p. 87–88].

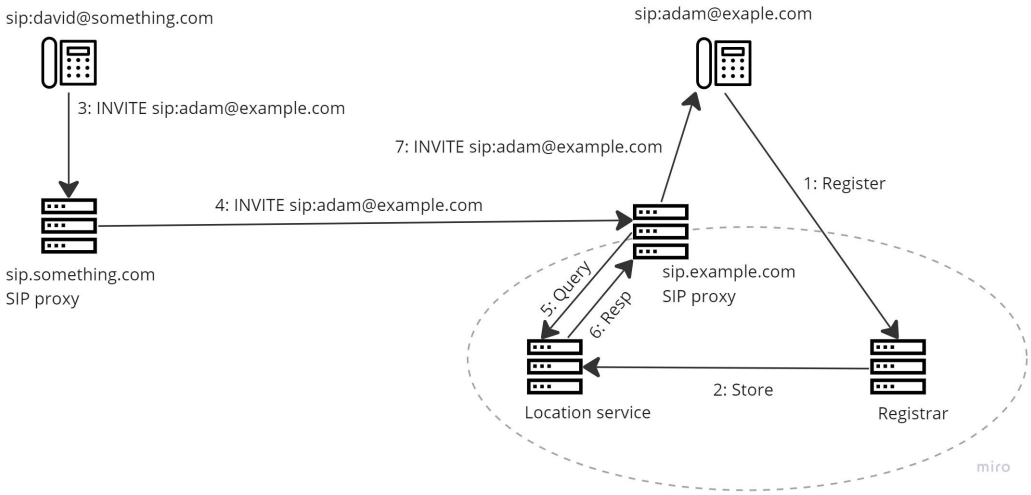

Figure 2.2: SIP registration and second user localization. Adapted from [6, p. 87]

2.1.2 Structure of the messages

SIP messages are human-readable and use the UTF-8 charset [16, p. 26]. They follow the standard format for Internet Messages[2], which is [8, p. 89] [16, p. 27]:

- Start line – method, protocol.
- Header fields in the form of `Type: value;parameters`.
- Blank line.
- Optional message body.

[2] https://www.rfc-editor.org/rfc/rfc5322

UA clients match the requests with received responses by the Tag and Call-ID values. The initial INVITE contains Tag in the From: header and Call-ID. The subsequent OK response adds Tag to the To: header. These two tags and Call-ID define a dialog [8, p. 88].

Request messages start with the method (INVITE, REGISTER) followed by the URI and protocol version, currently "SIP/2.0". Mandatory request header fields are To:, From:, Cseq:, Call-ID:, Max-Forwards:, and Via:. Message body is optional, but most of the time consists of SDP parameters.

Response methods contain protocol version (SIP/2.0) followed by a 3-digit code number and the text explanation. Some response messages also contain headers, such as OK response. Table 2.1 contains types of response codes [8, p. 89–90].

3-digit code	Class	Description
1xx	Provisional	Request received, continuing to process the request
2xx	Success	Action was successfully received, understood, and accepted
3xx	Redirection	Further action needed to complete the request
4xx	Client error	Request contains bad syntax or can not be fulfilled at this server
5xx	Server error	Server failed to fulfill a valid request
6xx	Global failure	request can not be fulfilled anywhere

Table 2.1: SIP status responses. Adapted from [8, p. 91].

Table 2.2 describes the structure of the INVITE method. The structure of other methods is very similar, especially the request ones.

Start Line	`INVITE sip:bob@biloxi.com SIP/2.0` *method called address SIP version*
Header fields	`Via: SIP/2.0/UDP pc33.atlanta.com;` `branch=z9hG4bKnashds` `Max-Forwards: 70` `To: Bob <sip:bob@biloxi.com>` `From: Alice <sip:alice@atlanta.com>;` `tag=1928301774` `Call-ID: a84b4c76e66710` `CSeq: 314159 INVITE` `Content-Type: applcation/sdp` `Content-Length: 142` *header type: value;parameters*
Required Empty Line	CRLF
Message Body	`Alice's SDP goes here`

Table 2.2: INVITE method format. Adapted from [8, p. 89].

The meaning of the headers is drawn from [8, p. 91–92] and [16, p. 39, p. 40, p. 169]. The `Via:` header identifies the location where the response is supposed to be sent to and its transport (such as UDP for example). Branch number is unique for all requests sent by UA. `Max-Forwards:` acts as a hop count before the packet is discarded. The `To:` header

contains the display name and the SIP URI of the user being called. Opposed to this, the `From:` header contains the display name of the caller. `Call-ID:` is a unique identifier for this call based on the caller's host name and a cryptographically generated random number. A sequence number incremented with each message that sets their order is stored in `CSeq:` header. `Contact` contains a SIP URI that is used to contact this exact UA for other requests. `Content-Type:` header describes the message body if present and `Content-Length` states its length.

The message body usually contains SDP (application/spd type) and the MESSAGE method carries text/plain or message/cpim content types in message bodies with the text message itself [18, p. 7].

Table 2.3 takes a look at typical REGISTER method structure which is used for locating the UA by other UAs.

Start Line	`REGISTER sip:biloxi.com SIP/2.0` *method register domain SIP version*
Header fields	`Via: SIP/2.0/TCP 192.0.2.2;` `branch=z9hG4bK-bad0ce-11-1036` `Max-Forwards: 70` `From: Bob <sip:bob@biloxi.com>;tag=d879h76` `To: Bob <sip:bob@biloxi.com>` `Call-ID: 8921348ju72je840.204` `CSeq: 1 REGISTER` `Supported: path, outbound` `Contact: <sip:line1@192.0.2.2;transport=tcp>;` `reg-id=1;+sip.instance="urn:uuid:00000000-` `0000-1000-8000-00A95A0E128>"` `Expires: 3600` `Content-Length: 0` *header type: value;parameters*
Required Empty Line	CRLF
Message Body	`none in REGISTER`

Table 2.3: REGISTER method format. Adapted from [8, p. 89].

The REGISTER method contains an additional mandatory `Expires:` header that contains the desired registration duration in seconds. The `Contact:` header contains an address to bind to AOR (Address Of Record). When a user wants to unregister, it sends a message with "Expires: 0". One user may register one AOR for multiple devices (UAs). `Supported:` header contains a list of option tags for SIP extensions [8, p. 94, p. 100] [16, p. 57].

During the processing of a request, each proxy adds a new `Via:` header with its IP address and data. These headers then allow each proxy to forward the method without the need of DNS lookup, for example. And when handling a response, each `Via:` header, belonging to a specific proxy, is removed so that only `Via:` of the calling phone is present when it reaches it [8, p. 92].

2.1.3 Session Description Protocol

When a session is being initiated, there is a need for media details, transport addresses, and other session metadata to be conveyed to the other participant [15, p. 3]. Session

Description Protocol (SDP) provides these exact services. With SDP, the user is informed about the session, so that he may decide whether to participate or where and how to join a session. The SDP is most of the time carried in the initial INVITE and subsequent OK method. The general structure is [8, p. 101–102]:

```
Session name and purpose
Time the session is active
The media comprising the session
Information needed to receive media (ports, addresses, etc.)
```

More than one media option may be listed and the UAS then can see whether it supports these. UAS may also send a list of its own media options and the overlap of these options is used in the subsequent session [8, p. 102].

The final choice of the media type is based on the "q=" header which contains the quality of each option in the interval between 0 and 1. Types are ranked and the one with the highest rating is used [8, p. 103].

Though UAC is not obliged to send SDP in the initial INVITE. When this happens, the UAS sends its SDP information in a subsequent response, that is "*delivered with a reliable method*" (Trying or OK). UAC then sends its SDP in the ACK method [8, p. 103].

SDP message format

The SDP, just like SIP, is an entirely textual protocol using ISO 10646 character set and UTF-8 encoding, though field names and attribute names use only US-ASCII subset. Format of the headers is as follows [15, p. 7–8]:

```
<type>=<value> with no white spaces
```

Where `<type>` is exactly one case-insensitive letter and `<value>` is structured text with its format dependent on the `<type>` [8, p. 103].

SDP has also its own MIME type, occurring in `Content-Type:` SIP header:

```
application/sdp
```

All messages have a defined order which they must follow [8, p. 105]:

- Session.
- Time.
- Media.

Each message begins with the "v=" header which starts the session-level part and continues to the media-level part, which starts with "m=" [15, p. 8]. Media parts contain specific attributes in "a=" and other lines. The session-level part is as follows (attributes with "*" are optional) [8, p. 104–106]:

```
v=(protocol version)
o=(oroginator and session ientifier)
s=(session name)
i=*(session information)
```

```
u=*(URI of description)
e=*(email address)
p=*(phone number)
c=*(connection information)
b=*(bandwidth information lines)
```

These attributes apply to all media sections unless overridden there. After the session level, a time description follows in the format of Network Time Protocol [8, p. 106].

```
t=(time the session is active) starttime stoptime
r=*(zero or more repeat times)
z=*(time zone adjustments)
k=*(encryption key)
```

After time description, zero or more media descriptions follow. These may override parameters in the session level.

```
m=(media name and transport address)
i=*(media title)
c=*(connection information)
b=*(bandwidth information lines)
k=*(encryption key)
a=*(media attribute lines)
```

The format of `<type>=<value>` is formally defined in Augmented Baus-Naur Form (ABNF)[3].

2.1.4 SIP vs Network Address Translation

Network Address Translation, or NAT for short, allows many hosts on a private network to communicate over the public network via one shared public IP address. NAT maps private addresses, usually beginning with 10.x.x.x or 192.168.x.x, and a port number into another port on the public static IP. When initiating a session from within NAT, a "pinhole" is created in a firewall. The firewall then takes a look at the destination port number of an incoming packet and maps it to the port number of the UA that created the "pinhole" [8, p. 196].

NAT was created as a security measure that hides the topology of the private network so that it is shielded from an attack, but outside connections from the internet are difficult because [8, p. 197]:

- Mapped port numbers do not correspond to well-known services.
- NAT requires an outbound packet with a specific IP and port before passing an inbound packet to the socket.

When traversing through NAT the IP addresses are changed, so signaling protocols and media packets, which contain the IP addresses of both end points, usually fail. The IP addresses used on the network between the NAT devices are different from the addresses of the UAs. Thus UAs can no longer give their native addresses [8, p. 197]. Figure 2.3 shows the topology.

[3] https://www.rfc-editor.org/rfc/rfc4234

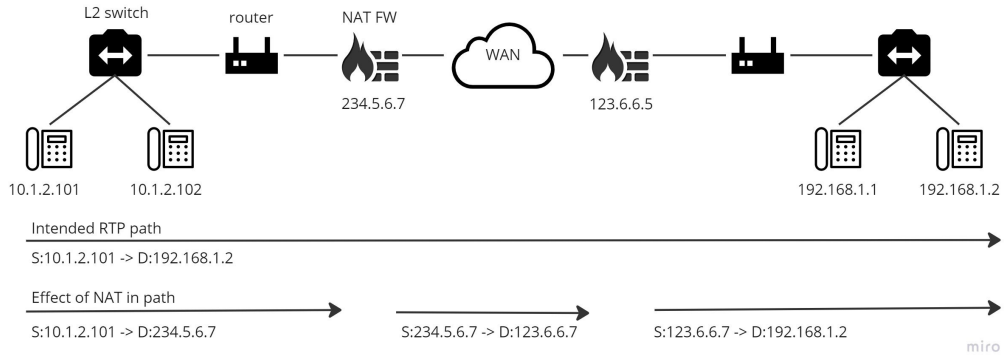

Figure 2.3: Address changes with NAT traversing. Adapted from [8, p. 198].

Solutions to this problem are using Session Traversal Utilities for NAT (STUN), Traversal Using Relays around NAT (TURN), and Interactive Connectivity Establishment (ICE), which all of these might have problems with symmetric NAT or NAT on both ends. The next subsections describe these solutions in detail. The most reliable solution is to register to a SIP proxy server. When a UA registers from within NAT, a pinhole is created, and by keeping the connection open, the proxy server is able to forward the messages to the registered UA [8, p. 197–198].

Session Traversal Utilities for NAT (STUN)

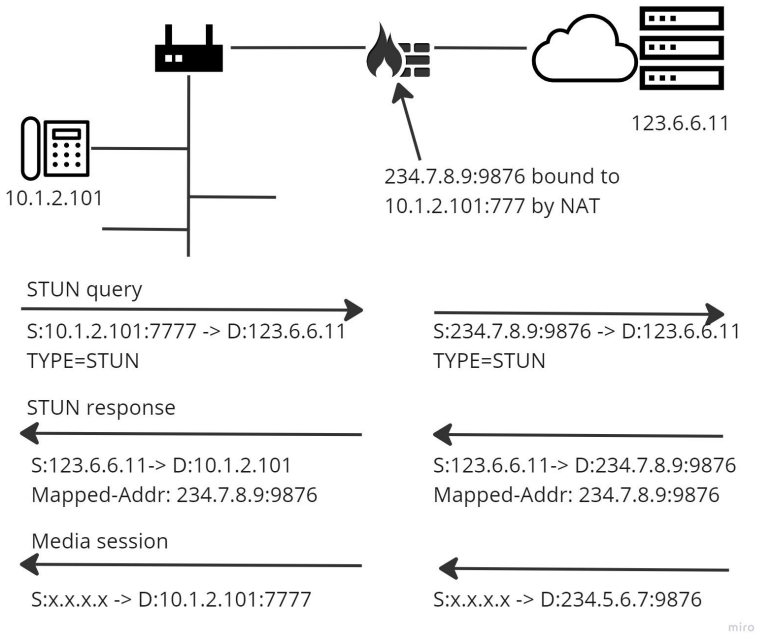

Figure 2.4: STUN topology and message exchange. Adapted from [8, p. 203].

STUN works on a basis of client-server. The server resides on a public part of the internet or in a network where it is reachable for all interested parties. The client then

queries this server from behind the NAT to learn what public IP and port the device was assigned to by the NAT firewall. The mapping of the private socket to the public socket is called "binding" [8, p. 201].

When a query arrives at the server, the source address is the translated or public IP address and port inserted by the NAT. The STUN server then returns this public IP and port in the payload of the response. When initiating a session with a device outside of the NAT, the caller gives its public IP address to the callee. Callee then sends its packets to the public IP and port of the caller [8, p. 203] as shown in Figure 2.4.

STUN has difficulties with symmetrical NATs and when both parties are behind different NATs. It is just a part of the process than a standalone feature – it provides only an address discovery and is not in the path of either media or signaling packets [8, p. 204].

Traversal Using Relays around NAT (TURN)

The TURN server is – again – located outside of the NAT network. It accepts packets from one device and forwards them to another, during which the server changes addresses, recalculates CRCs[4], and resets the Time To Live field. It is a relay extension of STUN designed to work also with ICE (see 2.1.4) or alone [8, p. 204].

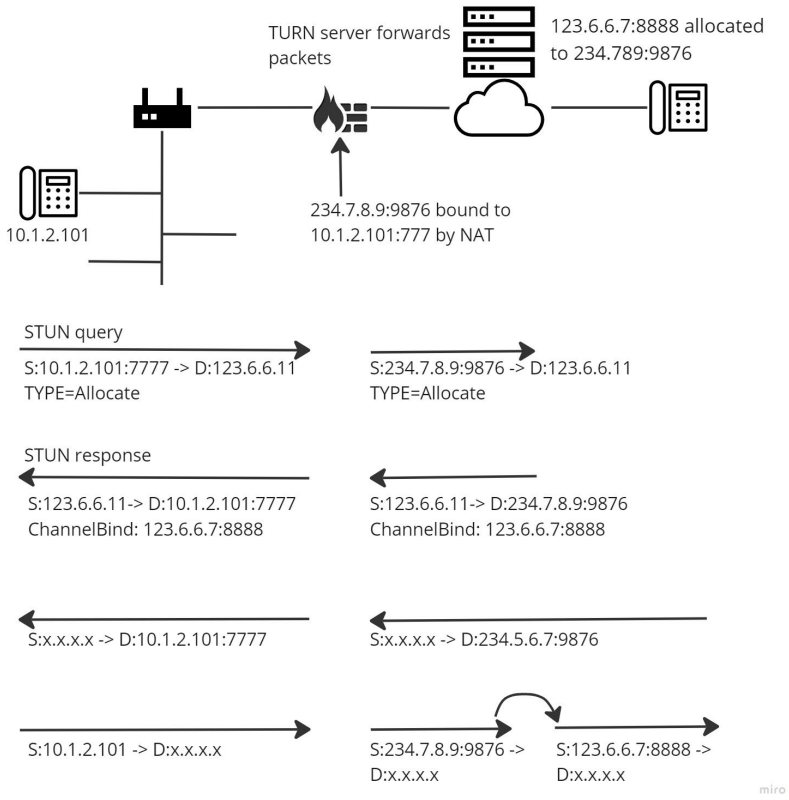

Figure 2.5: TURN topology and message exchange. Adapted from [8, p. 205].

To set up a connection, both end devices shall request an address allocation from the same server, or servers that can find each other. Then, with the ALLOCATE message sent

[4]Cyclic Redundancy Check – used for detection of accidental changes to digital data

by a client, the server allocates a unique public socket on itself and sends the information to the client in an `AllocationConfirm` message. The response sends information similar to STUN but the difference is, that STUN returns a socket on the client's NAT while TURN returns a socket allocated on itself [8, p. 204]. The caller sends packets to the callee by addressing its address, which is on the TURN server. The server translates the addresses and forwards them to the NAT function at the second site. Callee uses the same method of sending packets. Figure 2.5 illustrates the process.

Interactive Connectivity Establishment (ICE)

ICE is designed to work with STUN and TURN and is used for two UAs to learn each other's public addresses and ports for media exchange by the shortest or lowest cost path.

ICE offers each UA a list of CANDIDATE addresses, drawn from options such as:

- Network interface address.
- Translated address (public NAT address).
- TURN server allocated address.

Every address in this linked list is ranked with a preference attribute (q=, as talked about in 2.1.3). The UAs then try to connect directly to each of these addresses in order, until one works or the list is exhausted. The packets open a "pinhole" in the NAT firewall, which is kept open by a STUN keep-alive packets. If no more direct paths are possible, the packets are sent to a TURN server. If this connection test is successful, the route is established without the need of communicating anything special to NAT devices and without having a SIP-aware firewall [8, p. 206–207].

2.1.5 Electronic NUmber Mapping

This subsection draws from [8, p. 136]. Electronic NUmber Mapping, or ENUM for short, is a way of mapping a public telephone number to a unique SIP URI. It uses DNS and its NAPTR (Naming Authority Pointer) records to store these mapped URIs. DNS query may return any number of these records not only for SIP but also for fax, telephone, instant messaging, e-mail, and so on.

ENUM takes an E.164 telephone number, which is globally unique and reachable, makes it DNS-searchable, and assigns the requested SIP URI for it. The ITU recommendation for E.164 number format is as follows:

```
+ Country code.Area code.Exchange.Line
+ 1.202.555.1234 or + 1-202-555-1234
```

In the decimal system, the most significant digit (MSD) in a phone number is at the left, but DNS places the most significant element (TLD) to the right and subdomains are assigned to the left. The algorithm for making these numbers searchable is as follows:

- All non-numeric characters are removed: 12025551234,
- The order is reversed: 43215552021,
- Dots between digits are added. The subdomains and zones are created: 4.3.2.1.5.5. 5.2.0.2.1,

- A domain and TLD are added, making it an FQDN: 4.3.2.1.5.5.5.2.0.2.1.e164.arpa.

Typical NAPTR response also includes an e-mail server, a web server, a phone number (not necessarily the same one), and a SIP proxy server.

2.2 Real-time Transport Protocol

Real-time Transport Protocol, or RTP for short, is an application layer protocol that provides end-to-end delivery of voice and video packets. SIP itself does not transmit any media, so it uses this protocol for it. The RTP is usually run on top of UDP and does not require a particular port number when sending and receiving. However, when communicating through not RTP-aware NAT firewall, it is required to receive and send RTP on the same port [17, p. 4] [8, p. 55].

The RTP itself does not ensure reliable delivery of the packets but relies on the lower-layer protocols to do so. The sequence numbers in the RTP header allow the receiver to reconstruct the packet sequence. For the quality of service monitoring, the Real-time Transport Control Protocol (RTCP) is used (2.2.1) [17, p. 4]. Figure 2.6 describes the RTP packet.

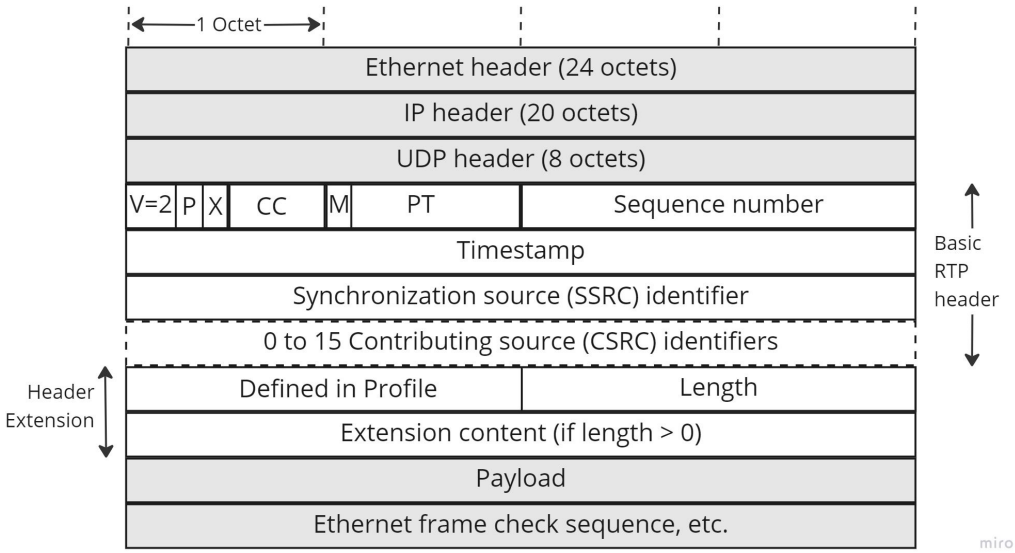

Figure 2.6: Real-time Transport Protocol for voice information. Adapted from [8, p. 55].

The meaning of the most important headers is as follows:

- **PT**: payload type, indicates the type of payload for the receiver to interpret.
- **SN**: the Sequence number.
- **Timestamp**: clock, that allows the receiver to compute jitter and synchronize the playback of different streams.
- **SSRC**: the Synchronization Source Identifier, a random number that should be unique within the whole RTP sequence.

- **CSRC**: the Contributing Source, represents sources of content that have been combined in some way.

Header extensions are optional and were inserted to allow vendors to experiment with proprietary functions. The payload containing the encoded media then follows [8, p. 56].

2.2.1 Real-time Transport Control Protocol

The Real-time Transport Control Protocol, or RTCP for short, is a protocol used for monitoring the RTP streams. The format of the packet is similar to RTP but contains only reports [8, p. 57]. Figure 2.7 describes the format of the RTCP packet.

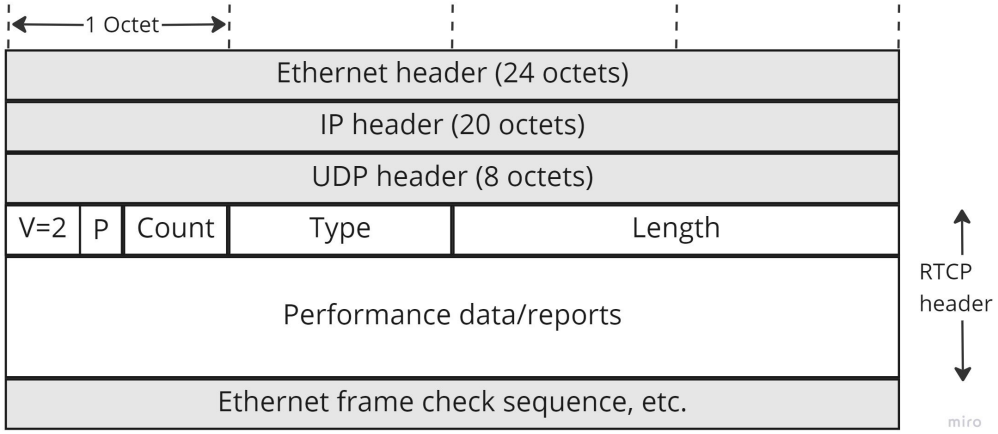

Figure 2.7: Real-time Transport Control Protocol used for RTP sessions monitoring. Adapted from [8, p. 57].

There are five types of RTCP packets [17, p. 21]:

- **SR Sender Report**: transmission and reception statistics from active senders.
- **RR Receiver Report**: for reception statistics from participants that are not active users.
- **SDES**: Source DEScription including the canonical name (CNAME) for the source.
- **BYE**: end of participation.
- **APP**: application-specific function.

Often at least two RTCP packets are bundled into one UDP packet, the first one being either the SR or RR type, and the second must be the CNAME for the source. The reason is that every time a new participant appears, each source identifies its CNAME [8, p. 58].

Key functions of RTCP are to synchronize the various clock and time stamps and keep track of counts of lost packets to calculate the quality of distribution networks. The frequency of the RTCP packets sent depends on some factors – available bandwidth and the number of end points. The report interval shall increase for more participants or lower link speeds but never exceed more than 5% of utilization [8, p. 58].

2.3 Internet Relay Chat

Internet Relay Chat, or IRC for short, is a worldwide system of real-time, synchronous, text-based conferencing on the Internet. Created in 1988 by Jarkko Oikarinen, the protocol is based on the client-server architecture and allows multiple users from around the world to engage in conversation with each other. The messages are not sent directly to the users, but they are relayed through an IRC server [6, p. 311] [14, p. 4].

Users of the IRC use an IRC client to connect to multiple servers on which they are able to join any server they want under an alias, called "nick" or "nickname". With users connected, they are then able to send and receive messages with the use of a graphical interface or text-based IRC clients [6, p. 312–313].

2.3.1 Structure and architecture

The basic components of the IRC structure and architecture are as follows [12, p. 2–3]:

- **Server**: a backbone of the IRC network. The server is the component users connect to with the intention of talking with each other and is intertwined with other IRC servers forming the IRC network.

  The application (on the server) that allows users to connect to the server and relay messages is called an IRC daemon [6, p. 312].

- **Client**: a client is anything connecting to a server that is not another server. There are two types of clients:

  - **User Clients**: a program providing an interface for IRC chatting, also referred to as "users".
  - **Service Clients**: a client that provides some sort of service to other users and is not used for chatting.

  There are also classes of clients: normal, operator, and channel operator. The main difference between them is the types of privileges they have.

- **Channels**: a channel is a named group of users that all receive messages addressed to that channel. The names of servers that start with "#" are global channels, accessible from any server. The ones starting with "&" are accessible only on that server itself [14, p. 5] [6, p. 312].

The web of servers creates the IRC network, where each server acts as a central node for the rest of the network it can see. Servers may communicate with other servers and users. No message is sent between two users directly. All messages travel through IRC servers to the designated user as shown in Figure 2.8. This method is called "relaying" [12, p. 3].

The IRC protocol provides these services that allow real-time chatting and conferencing [12, p. 4]:

- **Client Locator**: localization of two users for message exchange.
- **Message Relaying**: messages travel through a server before being sent to a specified user.
- **Channel Hosting and Management**: servers host and manage channels.

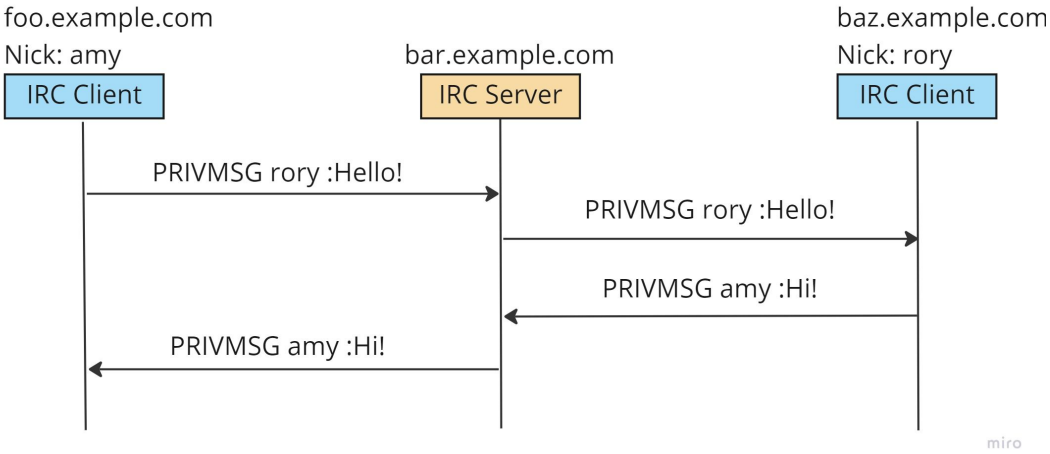

Figure 2.8: The IRC message relaying between two users. Adapted from [19].

The IRC protocol defines a set of concepts that helps with delivering messages of different classes. Next segment draws from [12, p. 5–7]

The `One-to-one` communication occurs mainly between users. With the messages being sent by servers in one direction through the spanning network tree, the resulting path is the shortest between any two end points on the spanning tree.

`One-to-many` style of communication refers to sending a message to multiple chosen targets. A user might send a message to a channel, which acts as a multicast group in IRC so only users in said channel receive the message. A message also might be sent to users whose host or server information matches a specific mask or the least efficient way is to define a list of targets to which to send the message.

`One-to-all` communication is, in fact, a broadcast message sent to all servers, users, or both. These classes of messages are mainly used for updating users' status on multiple servers for example. The message might be sent `Client-to-client`, which means it will be sent to all clients. `Client-to-server` sends the message to all servers. `Server-to-server` sends a message to all connected servers from a server.

To prevent clients to "flood" servers by sending a continuous stream of messages, flood protection is implemented to every server and applies to all clients except services. All clients are able to send one message every two seconds. The algorithm is as follows [14, p. 60]:

- Check to see if client's "message timer" is less than the current time (set to be equal if it is).
- Read any data present from the client.
- While the timer is less than ten seconds ahead of the current time, parse any present messages and penalize the client by 2 seconds for each message.

2.3.2 Messages

Clients and servers of IRC communicate through messages, using TCP as its transport protocol. Some messages may or may not generate a reply when a valid command is provided.

The message is composed of up to three parts: the prefix (optional), the command, and the command parameters (up to 15). All parts are separated by at least one (or more) ASCII space characters. The presence of a prefix is indicated by a single leading ASCII colon character (":") with no spaces around and it is used for determining the origin of the message. The command parameter is either a valid IRC command or a three-digit numeric code. All messages are terminated with CR-LF (Carriage Return - Line Feed) and do not exceed 512 characters, including the CR-LF pair [14, p. 7–8]. The Backus-Naur Form representation of a message is as follows [14, p. 8]:

```
<message>  ::= [':' <prefix> <SPACE> ] <command> <params> <crlf>
<prefix>   ::= <servername> | <nick> [ '!' <user> ] [ '@' <host> ]
<command>  ::= <letter> { <letter> } | <number> <number> <number>
<SPACE>    ::= ' ' { ' ' }
<params>   ::= <SPACE> [ ':' <trailing> | <middle> <params> ]

<middle>   ::= <Any *non-empty* sequence of octets not including
                SPACE or NUL or CR or LF, the first of which may
                not be ':'>
<trailing> ::= <Any, possibly *empty*, sequence of octets not
                including NUL or CR or LF>

<crlf>     ::= CR LF
```

Server registration

The server registration is conducted by sending three commands to the desired server: Pass command (although not required sometimes), Nick command, and User command, in this order. See Figure 2.9 for illustration.

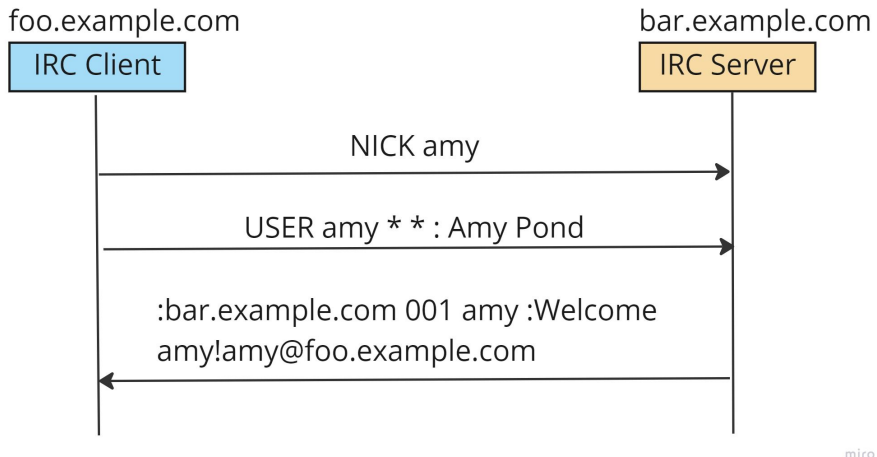

Figure 2.9: The client registration process without the PASS command. Adapted from [19].

The Pass message is used to set a "connection password" and is not required, though it adds some sort of level of security to the connection. This command must precede the Nick and User commands [14, p. 14].

```
PASS <password>
```

The Nick command is used for setting or changing users' nickname which is used for addressing the user. Hop count is used by servers to identify how far the user is from its home server [14, p. 14].

```
NICK <nickname> [ <hopcount> ]
```

The User message is used for setting the username, server name, hostname, and real name. Also used between servers to indicate a new user. The real name should be at the end and prefixed by ":" because the real name might contain spaces [14, p. 15].

```
USER <username> <hostname> <servername> <realname>
```

For terminating the session, command `QUIT [<message>]` is used. The `OPER <user> <password>` command is used to obtain operator privileges [14, p. 17].

Text messages and channel commands

To join a specific channel or channels, the command Join is used. To successfully join a channel the user has to be invited if it is invite-only, the user does not have an active ban, and the correct password has to be provided if set [14, p. 19–20].

The Join message is also relayed to all participants in the channel as shown in Figure 2.10. Note that the Join message is also relayed back to the original user as a confirmation for successfully joining the channel [19].

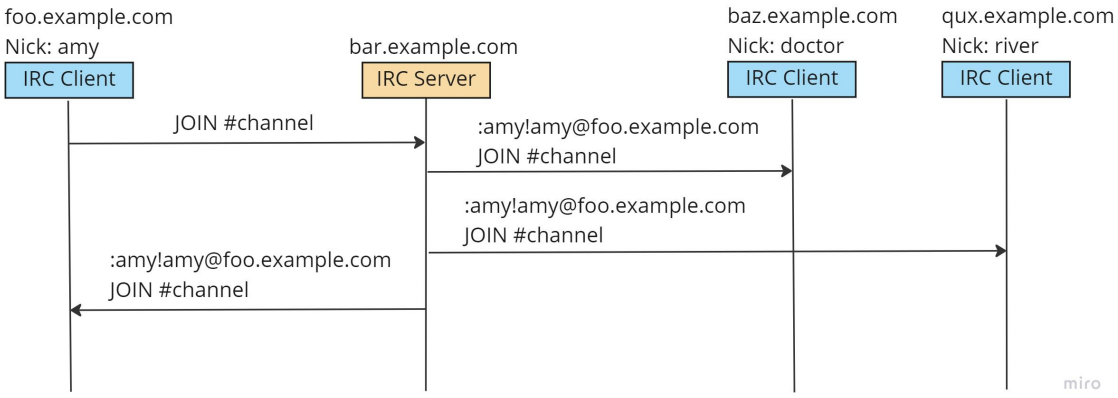

Figure 2.10: Joining the channel and relaying the Join message. Adapted from [19].

Just before joining, the user is able to see a list of public channels on the specific server with the List command [6, p. 312].

```
JOIN <channel>{,<channel>} [<key>{,<key>}]
LIST [<channel>{,<channel>} [<server>]]
```

Names message lists all nicknames visible to a user in a channel [14, p. 24] and Part message removes the user from the active users list [14, p. 20].

```
NAMES [<channel>{,<channel>}]
PART <channel>{,<channel>}
```

Some commands are available only to the Channel Operators. These are `TOPIC` that changes or sets the channel's topic, `KICK` which kicks a user, `MODE` which is used to change the channel's mode, and `INVITE` which invites a user to an invite-only channel [14, p. 6].

For sending text messages, the Privmsg command is used. `<receiver>` contains the nickname of the recipient or a channel or a server. It also might be a list of recipients [14, p. 32]. When a Privmsg is sent to a channel, the message is relayed to all users in the channel as shown in Figure 2.11. When a message is sent to only one user, it is relayed through a server to the specific user as shown in Figure 2.8 before.

```
PRIVMSG <receiver>{,<receiver>} :<text to be sent>
```

The `NOTICE <nickname> <text>` message is similar to Privmsg but automatic replies are never sent in response to a `NOTICE` message [14, p. 33].

These messages also support wildcards for addressing the receiver. The wildcards are "*" and "?". For example, `PRIVMSG #*.edu` sends a message to all channels ending with `.edu` [6, p. 32].

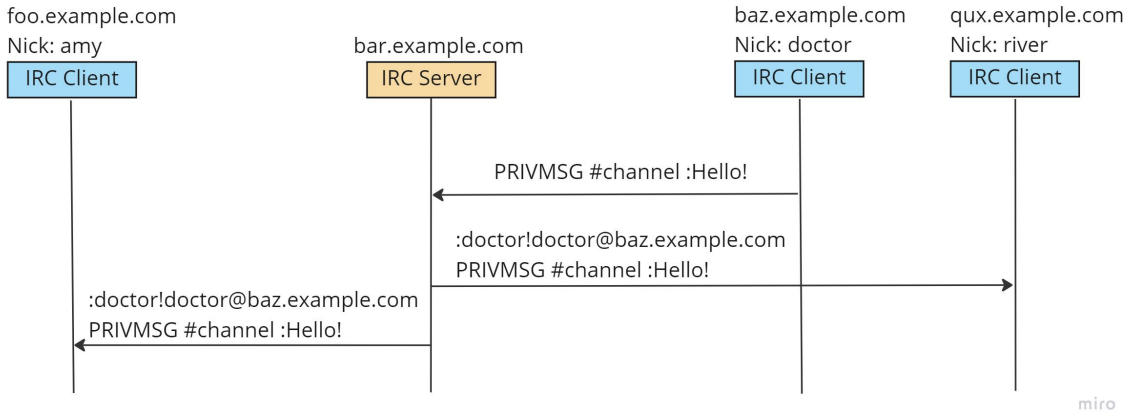

Figure 2.11: Message relaying to channel members. Adapted from [19].

Other commands

Some other commands may be used for the user to query various information about the server. All servers must respond to these queries.

The Version message returns the version of the queried server. The Stats message returns the statistics of a particular server. For the name of the admin of a specific server, the Admin command is used and for the server information, the Info command is used [14, p. 26–31].

```
VERSION [<server>]
STATUS [<query> [<server>]]
ADMIN [<server>]
INFO [<server>]
```

The User-based queries help with finding information about other users or a group of users. The Who command is used for returning a list of information about a certain user. A similar command called Whois returns the same information but the user is able to address a specific server [14, p. 33–34].

```
WHO [<name> [<o>]]
WHOIS [<server>] <nickmask>[,<nickmask>[,...]]
```

The last important group of messages is the miscellaneous messages, used for connection and error-oriented services. The Kill command is used for terminating the connection between a user and server when a duplicate entry in the list of valid nicknames is found. Used by servers and operators [14, p. 36].

```
KILL <nickname> <comment>
```

One of the most important commands is the PING-PONG duo. These are used for testing whether the endpoint is still active or not. The Ping message is sent at regular intervals and if the endpoint fails to send a Pong response in a certain amount of time, the connection is terminated. The Pong response contains the name of the same daemon that arrived in the Ping message [14, p. 37].

```
PING <server1> [<server2>]
PONG <daemon> [<daemon2>]
```

The Error command is used by servers to report fatal errors to its operators and it is server-to-server only. Operators receive this message encapsulated in a Notice message [14, p. 38].

```
ERROR <error message>
```

2.4 Applications

Here are some of the applications that were taken a look at during the research. Two of them are SIP User Agents with their own open-source APIs (Linphone, Baresip), one is an IRC client and the fourth application is an IRC gateway.

2.4.1 Linphone

Linphone is an open-source SIP-based user agent, available for mobile and desktop. It features all basic SIP-related services, such as audio and video calls, call management, call transfer, audio conferencing, and instant messages. It also provides user-experience features such as file sharing, contact lists, and chat during calls. Linphone is available with a graphical interface or as a console application.

Linphone uses its open-source library as its core, called liblinphone. The library is a SIP-based SDK[5] for video and audio over IP and is written in C/C++.

The application is available on Linux, Windows, MacOS, iOS, and Android [5].

[5]Software Development Kit

2.4.2 Baresip

Baresip is also a SIP user agent but uses the command line as a graphical interface. The application uses librem and libre libraries for audio and video calls and real-time communication, respectively. Both libraries are written in C/C++.

Baresip offers similar services as Linphone (SIP features and user experience features), but it omits the comfortability of graphical user interfaces.

Baresip is also available on the same platforms as Linphone, that is Linux, Windows, MacOS, iOS, and Android [3].

2.4.3 Weechat

Weechat is a lightweight IRC client with a text-based user interface. It was designed to be heavily extensible with plugins. It is a full-fledged text client with user-experience features, such as incremental text search, spell checker, scripts manager, and customizable interface.

The user is able to connect to numerous IRC servers, join channels and chat with other users, using Internet Relay Chat protocol.

Weechat is available on multiple platforms: Linux, UNIX, BSD, GNU Hurd, Haiku, MacOS and Windows [11].

2.4.4 BitlBee

As opposed to Weechat, BitlBee is not an IRC client but an IRC gateway. It provides access to other popular messaging applications, such as ICQ, Facebook, Skype, and Twitter for example. BitlBee communicates with the users via IRC protocol.

The user installs BitlBee and connects to the BitlBee server (mostly local) with its favorite IRC client. After registering, user is able to work just like he normally does on IRC or is able to use any other supported instant messaging application.

BitlBee is available on Linux, UNIX, BSD, Windows, AmigaOS, and MacOS [7].

2.4.5 Summary

For SIP and call implementation, the liblinphone SDK was chosen for this thesis. Liblinphone is overall more documented and recent than the libraries offered by Baresip. Furthermore, Baresip offers two libraries, as opposed to liblinphone which bundles all dependencies together.

For the program's "graphical interface", the IRC client Weechat was chosen. The reasoning behind this decision is that Weechat is very lightweight and the process of connecting to an IRC server is much easier as opposed to BitlBee.

Chapter 3

Specifications

This chapter takes a look at the program's formal requirements – what the final program should be capable of – which are described in Section 3.1. For a detailed explanation of the architecture and concepts, refer to Section 4.2.

3.1 Requirements

This thesis aims to create a SIP user agent with an IRC client as its graphical interface. The formal requirements come from the assignment of the thesis, from consultations with the supervisor and some were inspired by researching other applications. Program is required to perform basic SIP services and to maintain a good user experience. These requirements are:

- A console application in the form of an IRC bot.
  - The bot acts as a user agent.
- An IRC client or gateway as the program's graphical interface.
  - Use the IRC protocol for the user agent's control – in the form of commands addressed to the bot.
  - Provide feedback after each command.
- Be able to make voice calls with the use of Session Initiation Protocol.
  - Basic call, either through proxy or peer-to-peer.
  - Be able to process the situation where UA receives numerous incoming calls simultaneously.
  - Be able to host multiple outgoing or incoming calls.
- Standard NAT traversal utilities.
  - STUN and TURN support, inspired by SIP's problem with NAT in Subsection 2.1.4.
- Standard call control.
  - Hold and resume a call.

  - Hangup current call.
  - Accept or decline an incoming call.

- Instant messaging.
  - Send instant text messages not only through IRC but also to other SIP user agents.
  - Send messages also during a call directly to the other participant. Inspired by Linphone.

- Standard proxy registration.

- Be able to call with the help of ENUM (2.1.5).

- A local database containing contacts of the user – address book. The database shall also hold proxy identities, just like other user agents.

The program shall be written in C/C++ with the help of a third-party library for implementing calls. The supported platform is Linux.

Automated acceptance tests are also required. Tests may be in a form of a script, using a SIP traffic generator to test calls. Architecture and implementation of the automated tests are described in Chapter 6.

Chapter 4

Design

This chapter describes the choice of technologies used for this thesis in Section 4.1. The architecture and intended use of technologies are described in Section 4.2, based on the formal requirements from Chapter 3.

4.1 Technologies

The choice of technologies for this project, or any other project, is essential. Section 2.4 introduced four applications researched for this thesis – two SIP user agents, one IRC client, and one IRC gateway. Both user agents are developed with open-source SIP libraries, which might be used for this thesis.

For implementing calls and SIP services, the liblinphone was chosen as the main library. This project uses linphone-SDK[1], which bundles liblinphone and its dependencies together. As the program's graphical interface, Weechat was chosen. For the reasons why, see 2.4.5.

The programming language for this project is C/C++. Although the classes are present, no OOP[2] pattern is implemented. The reason is, the classes are only used for encapsulating data and making the code cleaner with class methods. Standard C++ and C libraries are used, such as `string, vector, sys/socket` and `iostream` for example.

The operating system this application aims at is Linux, mainly because of the Linux sockets used for communication with IRC.

For the address book, which is conceptualized as a local database, a database language is needed. Thus, SQLite3 with its C/C++ interface API was chosen. SQLite3 provides easy integration and database manipulation within a C/C++ program.

4.2 Architecture

The user agent is conceptualized as an Internet Relay Chat bot that joins a channel on an IRC server and awaits for commands, sent via the IRC protocol. The bot is listening to commands sent only from a certain user, specified in the command line argument.

[1] https://github.com/BelledonneCommunications/linphone-sdk
[2] Object-oriented programming

4.2.1 Connecting

Program is launched from the command line, where `server, channel, user,` and `password` are specified. The `server` and `channel` are locations the bot connects to and waits for commands. `User` is a nickname of the user the bot listens to and `password` is the bot's password. Bot's nickname, which is used for its addressing, is created from the provided user's nickname and a suffix "_b", for example, `Joe_b` if the user's nickname is `Joe`.

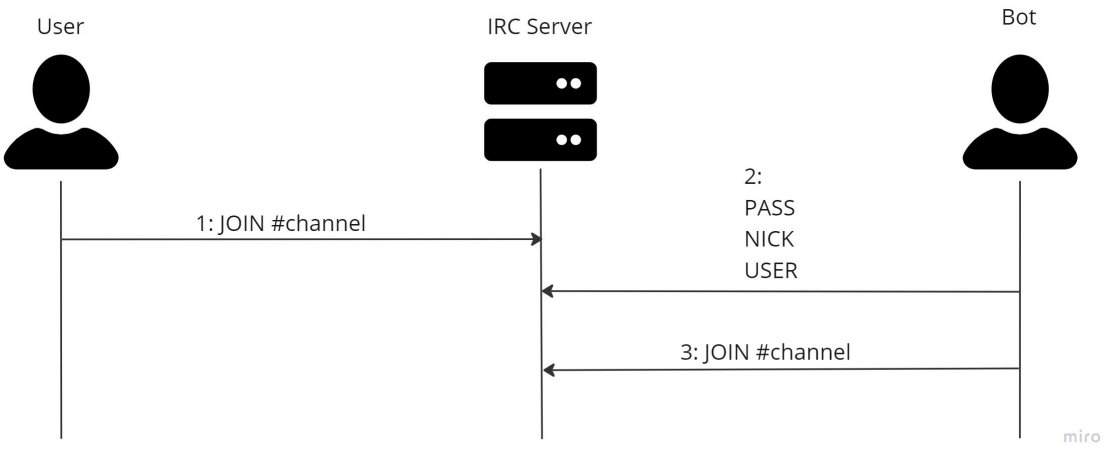

Figure 4.1: Bot and user connecting to the server and channel.

It is advised for the bot and the user to connect to the same server and channel and that the user is connected first. After successfully connecting, the bot sends a welcome message and is ready to accept commands.

4.2.2 Commands

Commands sent from the user are in the form of private messages relayed through an IRC server to the bot, as shown in Figure 2.8. The format is a standard `PRIVMSG` containing the command and its parameters.

```
PRIVMSG nickname_b : <command> <command_parameters>
```

The command is a human-readable text that corresponds to the action, for example:

- register: register to a proxy,
- call: initiate an outgoing call,
- hangup: terminate the current call.

Parameters contain additional important information for the command to be successful, for example, `call` must be followed by a SIP URI that the user wishes to call.

Some commands, especially for working with the address book, are only composed of the initial letters of the actions, preceded with a dash. For example, a command for inserting a contact would be `-ic` (**i**nsert **c**ontact). The full list of commands is in the program manual, Appendix B.

4.2.3 Registration and NAT traversing

When the bot successfully connects it is not registered to any proxy nor connected to any STUN or TURN server. It is possible to initiate outgoing calls, but only in Local Area Network. When a user wishes to communicate with users outside of his LAN, it is crucial to register to a SIP proxy or use utilities for NAT traversal.

For the user to specify STUN or TURN server, the `-s` command is used. This command requires a hostname or an IP address of the server. For passing the TURN credentials, an optional argument `-t` is used.

```
-s <server> [-t <turn_username> <turn_password>]
```

For the bot to be able to register to a proxy, the user must provide its SIP URI and password. The `register` command is used for this. The command is relayed to the bot, then SIP method `REGISTER` is sent to the desired proxy as shown in Figure 4.2.

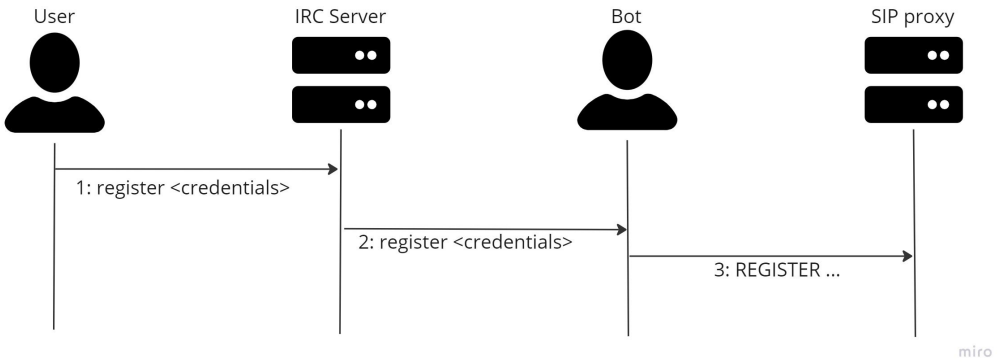

Figure 4.2: Registration of the user agent to a SIP proxy.

Register command also supports the address book lookup with the `-a` argument and provided `<name>`. A closer description of the address book logic is described in 4.2.6. The format of the command is as follows:

```
register {<sip_uri> <password>} | {-a <name>}
```

After a successful registration, a confirmation message with the user's current URI is sent to the user.

4.2.4 Calls

According to formal requests, the program must be able to initiate and receive calls from other user agents. Calls may be either peer-to-peer or with the use of a proxy. How the bot registers or is able to use utilities for NAT traversal is described above in 4.2.3.

The user is able to initiate a call with the `call` command. A valid SIP URI must be provided for the call to be successfully initiated. After the command is relayed, the bot sends INVITE method to the other party, as illustrated in Figure 4.3.

`Call` command also supports dialing a user from the address book with the `-a` argument and `<name>`, see 4.2.6 for a detailed explanation.

With the `-e` argument, the user is able to call with the help of ENUM lookup. The `<number>` provided must be in the E.164 format, described in 2.1.5. The format of the command is as follows:

```
call {<sip_uri>} | {-a <name>} | {-e <number>}
```

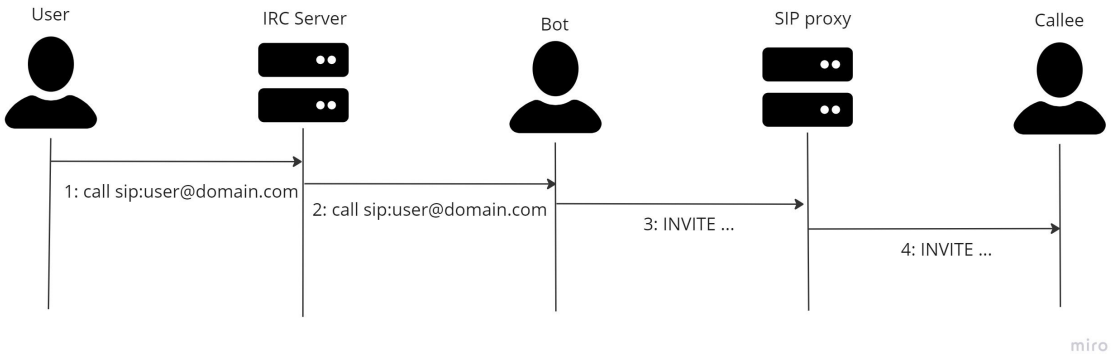

Figure 4.3: Initiation of an outgoing call through proxy.

The bot acts as the applications' "backend". It receives and processes SIP messages, and subsequent RTP streams and handles media. With the call initiated, RTP streams are opened directly between the participants, as the standard states. For voice capture and audio playback, the mediastreamer2[3] library for media processing is used by the bot, which is part of linphone-SDK. With the bot being interpreted as the "backend", the interface of the IRC client is interpreted as the "frontend", for this project.

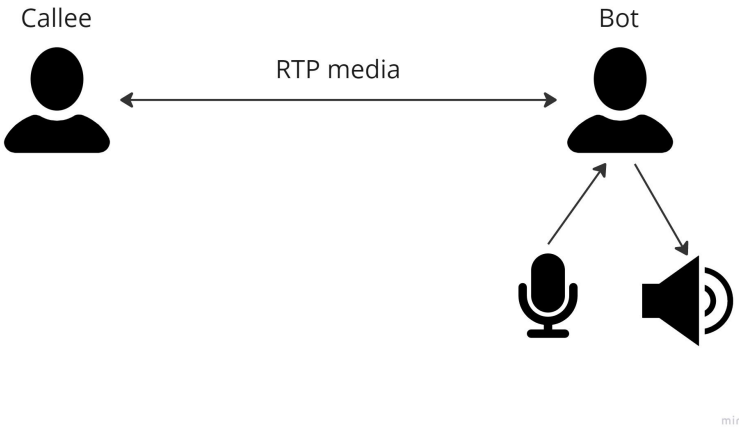

Figure 4.4: Media exchange with voice capture and audio playback.

When another user agent initiates a call, the bot sends a message after receiving the INVITE method, informing the user that someone is calling. The message contains a remote URI and an order. With each new incoming call, the order increments. This order allows

[3] https://github.com/BelledonneCommunications/mediastreamer2

the user to choose a specific call to accept when more than one is ringing. The order of the call must be specified in the `accept` command.

The user has a choice to `accept` or `decline` a call. When a specific call is accepted, all other incoming calls are declined. Figure 4.5 illustrates accepting an incoming call.

```
accept <order>
decline
```

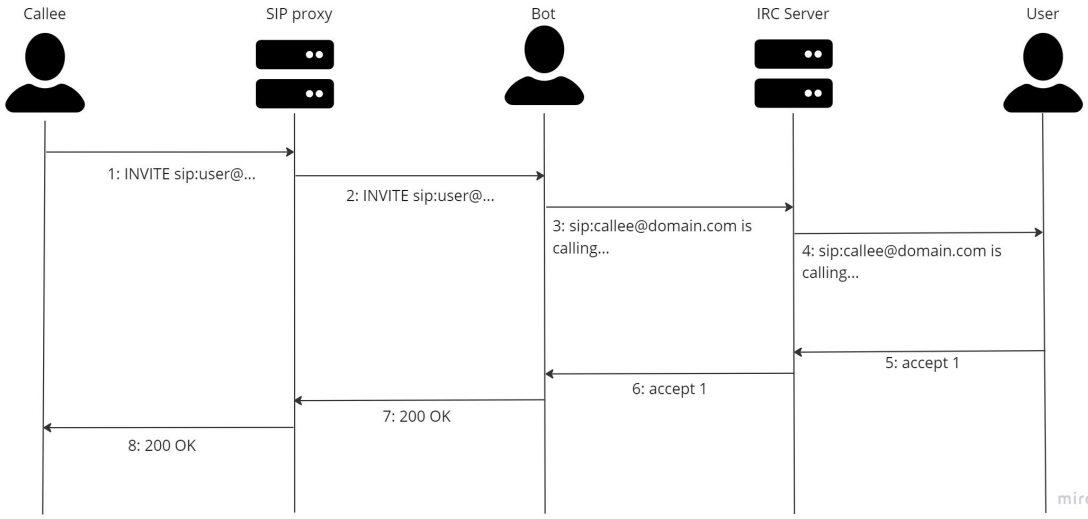

Figure 4.5: Receiving an incoming call and accepting it.

When a call is successfully established, either outgoing or incoming, there is a possibility to control the flow of the call. It is possible to hold and resume a call with the use of commands of the same name. With the `hangup` command, it is possible to terminate the current call.

```
hold
resume
hangup
```

4.2.5 Instant messages

The bot is able to receive and send instant messages with the use of the `MESSAGE` SIP method. When one is received, the remote URI of the sender and the text are displayed to the user. Example of a received message:

```
joe@sip.example.com: Hello!
```

For the user to send a message, the command `mess` is used. The user must provide a SIP URI of the receiver and a text message. It is also possible to send a message to someone from the user's contacts with arguments `-a` and `<name>`.

```
mess {<URI> <text_message>} | {-a <name> <text_message>}
```

The liblinphone library also supports a feature, where after a call is established, the user may send a message directly to the other party. For this feature, the command `-m` is used. Note that it may be used only during a call and the message is sent only to the second participant, that is why no URI may be specified.

```
-m <text_message>
```

4.2.6 Address book

An address book allows the user to save SIP URIs to contacts and also its SIP proxy credentials for faster registrations and dialing. The address book is conceptualized as a local database that communicates with the bot. The SQLite3 C/C++ API allows these functionalities.

The structure of the database consists of two tables, `contacts` and `registrar`. Each entry in the database consists of `ID`, `name`, `sip_uri`, and the `registrar` table also includes `sip_password`. The `name` attribute shall be unique within the table and is used for querying the database for information.

The `-a` argument, supported by `call`, `register` and `mess` methods, is the sign for the bot to retrieve needed information from the database, corresponding to the given `<name>` argument. After the information is retrieved, the command delivered with `-a` argument is performed. The process of calling a user with the database lookup is illustrated in Figure 4.6.

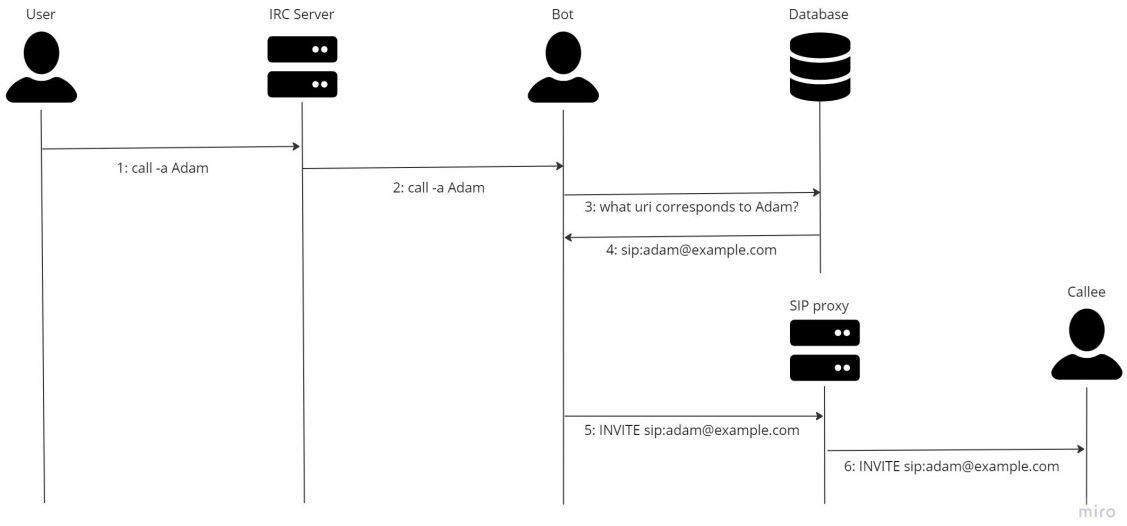

Figure 4.6: Initiation of an outgoing call with database lookup.

The user is able to perform basic CRUD[4] operations. Commands for these operations start with a dash "-", followed by the first letters of the desired actions. For example, the format for **i**nsert **c**ontact and **r**emove from **r**egistrar commands are as follows:

```
-ic <name> <sip_uri>
-rr <name>
```

[4]Create Read Update Delete

To browse the user's contacts or SIP identities, a database browsing mode is available. The mode may be accessed by `-con` for contacts browsing or `-reg` for identity browsing. The data entries are projected to the user, five at a time. If the user has more entries, they are divided into pages. User may also specify a pattern, which acts as a regular expression that filters the `name` attribute. For traversing pages while in the database browsing mode, commands `next` and `prev` are used, and `exit` for leaving the mode. For the full list of commands, see Appendix B.

Chapter 5

Implementation

This chapter closely describes the program's implementation and its details with code samples. Also, work with the third-party library, liblinphone, is explained. Code for calls implementation was drawn from linphone's official documentation and tutorials [4]. Sections have a similar structure as the architecture described in Section 4.2, that is: Connecting (5.1), Commands (5.2), Registration and NAT (5.3), Calls (5.4), Messages (5.5) and the address book (5.6).

The program was developed in Visual Studio Code on Ubuntu 20.04, and the source code is distributed between 15 files, including the header files. Note that the names of files and classes are the same.

Source code is compiled with standard `g++` compiler, with the use of `Makefile`, generated by the GNU Autotools[1]. After the compilation is successful, binary, named `irc_bot`, appears in the root directory.

5.1 Launching and connecting

The program is a console application launched from the command prompt and it does not need any special root privileges. It is launched as any other console application with four required arguments. The main function of the program is implemented in the `irc_bot_main.cpp` file.

```
./irc_bot {server} {channel} {user} {password}

server - IRC server the bot connects to
channel - channel the bot joins
user - a nickname of the user the bot shall listen to
password - the bot's password
```

After launching the program, a linphone core is created. The process of the creation is closely described in Section 5.3. All of the objects used in this program are global singletons and are defined and declared just before the main function.

With the core successfully created, it is necessary to obtain a database handle for the address book. The class method for opening the database handle is `addrBook::addr_book_open()`. The `addrBook` methods are closely described in Subsection 5.6.

[1] https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html

After successfully creating and opening global dependencies, the process of connecting the bot begins. The bot itself is designed as a TCP client, which connects to a TCP server. The logic of how the bot works is described in [1]. Although the tutorial is written in Python, the logic is the same in C/C++. Class called `irc_bot` represents the bot. At the start, the command line arguments are parsed to bot's attributes.

The server's hostname is resolved with the `gethostbyname()` method to get the appropriate IP address. With the hostname successfully resolved, a TCP socket descriptor is created. It is crucial to set a socket timeout with `setsockopt` for the socket to not wait for a packet to continue. The reason for this option to be crucial is that the application has to run in a constant loop for correct functionality. This also ensures that there is no need for implementing multiple threads. The program always awaits a specific command or message for an action to be performed but still continues to perform other tasks.

After the socket is prepared, the `connect` method connects to the server. Bot sends the `PASS, NICK`, and `USER` commands to authenticate and after the authentication is successful, bot sends a "welcome message" to the user with its current SIP URI. Bot joins the channel after the `MODE` command was sent by the server. The program then enters the main application loop and is now ready for interaction.

To terminate the bot, `end` command or `SIGINT` CTRL+C interruption are available. The `SIGINT` signal callback function is defined at the start of the `irc_bot_main.cpp` file. The process of the termination (either for the signal and for the `end` command) is as follows:

1. Close the database handle.
2. Unregister the user from a SIP proxy.
3. Destroy the linphone core.
4. Send a bye message and QUIT the IRC server.
5. Exit the program.

Steps 2 – 4 are implemented in the `irc_bot::bot_terminate()` method and destruction of the core and unregistration of the user are closely described in Subsection 5.3.

5.2 Command handling

Commands are transmitted via TCP packets. These packets are received in the main loop with the `recv` method. The variable `bytes_received` is greater than zero when the bot receives a packet. In that case, the message is processed or else the loop continues.

The CL-RF duo is cut off and the whole command is split with spaces (" ") as a delimiter into the `messages` vector. To split the message, a method with the same name is used. The method is not a standard C++ method and the code is adapted from [9].

```
ircMsg = string(buffer, 0, bytes_recieved);
...
ircMsg.resize(ircMsg.length() - 2);
split(ircMsg, " ", messages);
```

Just before the bot handles a specific command, other miscellaneous matters must be resolved. Whether the `messages` vector contains the `MODE` command (then the bot joins

a channel) or the `ERROR` command for an unsuccessful connection. If the vector contains `PING` command, an adequate `PONG` response is sent.

If the message is a private message, the corresponding nickname has to be checked. The first index of the vector has to be split with the use of "!" as a delimiter. It contains a nickname of the sender and it has to match the `user` argument. The third index shall contain the bot's nickname. Indexing and format of this message are shown in Figure 5.1. If one of these checks fails then the loop continues.

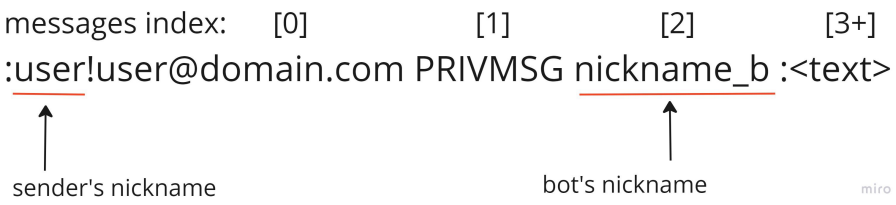

Figure 5.1: Format and indexing of an IRC PRIVMSG message.

The code snippet below shows the process of checking the first index of the `messages` vector for the correct nickname. It is also needed to also check, whether the message was addressed to the bot, i.e. the third index.

```
split(messages[0], "!", aux); // 1st index shall contain the nickname
...
if((aux[0] != correct_nick) || (messages[2] != bot.nick))
    continue;

string command = messages[3];
```

The `command` variable then contains the fourth index which should contain a command. This variable's content is then compared with string literals, containing supported commands. If they match, appropriate action is performed or else an unknown command message is sent.

The command parameters are processed the same way. The appropriate `messages` index is again compared with a string literal. Correct usage of the commands is checked through the vector's size. When the user sends an invalid format of a command, a message containing the correct usage is sent.

After a successful action is performed bot shall always send feedback, confirmation, or a fail message. The `send` socket method is used for communication with the user and the server.

Two `irc_bot` class methods are available for communication. The `send_com()` method sends the desired string and an empty line afterward for better visual separation of responses. The `send_ init_com()` method sends a string literal without the subsequent new line. A typical example of sending a response:

```
string msg = "PRIVMSG " + user_nick + " :text\r\n";
bot.send_com(msg);
```

A complete list of commands, their format, and usage can be found in Appendix B. Also, a structured `help` command can be used for help while using the bot.

5.3 Registration and NAT traversing

When the bot joins and sends the welcome message, it is not registered to any proxy nor uses any NAT traversal utilities. These are needed to be set if the user wishes to call outside of LAN.

The chosen third-party library liblinphone helps to implement these features. The condition for working with the library is to create the linphone core. This core does basic SIP and other functionalities in the background and is crucial.

The `irc_bot_core` class implements all linphone core-related functionalities. The core is created with the `irc_bot_core::core_create()` method. As the documentation states [4, Initializing], the core shall first be created and then started. After both of these actions are successful, the core callbacks are inserted. The code snippet is showcased in Listing 5.1. These callbacks are called after an important action is made by the linphone core and are also implemented in the `irc_bot_core.cpp` file. With the core created at the beginning of the `main` function, the bot may now register to a proxy, set NAT traversal options, initiate or receive calls, and other SIP-related functionalities.

```
this->_core = linphone_factory_create_core_3(...);
...
int err;
if((err = linphone_core_start(this->_core)) != 0){
   ...
}
//create the core callbacks
this->_cbs = linphone_factory_create_core_cbs(this->_factory);
...

/* Add callbacks */
linphone_core_cbs_set_call_state_changed(...);
linphone_core_cbs_set_registration_state_changed(...);
linphone_core_cbs_set_message_received(...);
linphone_core_add_callbacks(...);
...
// set other core options
...
```

Listing 5.1: Creation of the linphone core and callbacks setting.

When the user wishes to terminate the bot, the core is destroyed. First, the core is stopped and then the reference counter to the object is decremented, which shall destroy it. The `irc_bot_core::core_destroy()` method implements this behavior.

Setting the NAT traversal utilities

Enabling the NAT utilities is implemented also in the `irc_bot_core.cpp` file. The corresponding command to enable these functionalities is `-s` with its optional `-t` command to also enable the TURN support.

This command may only be performed if the user is not in a call. The user has to provide a valid STUN or TURN server as its first argument. If the format of this command is valid, STUN and ICE support is enabled with the `irc_bot_core::enable_stun()` method.

The linphone methods recommended for enabling the STUN and ICE support are as follows, according to [4, Network parameters]:

```
linphone_nat_policy_enable_ice(this->_nat, true);
linphone_nat_policy_enable_stun(this->_nat, true);
linphone_nat_policy_set_stun_server(this->_nat, address.c_str());
... // check, whether the stun server is valid or not
linphone_core_set_nat_policy(this->_core, this->_nat);
```

The `LinphoneNatPolicy _nat` attribute first must be initialized with `linphone_core_create_nat_policy()` method. This behavior is implemented in `irc_bot_core::create_nat_policy()` method which is called right after creating the core.

For the optional `-t` part to be successful, the user has to provide a valid TURN username and password. After successfully parsing and checking the arguments, the `irc_bot_core::enable_turn()` method is called.

The method must create a `LinphoneAuthInfo` object that stores provided username and password, then sets the username.

```
linphone_nat_policy_enable_turn(this->_nat, true);
this->_turn_cred=linphone_auth_info_new(...);
linphone_core_add_auth_info(this->_core, this->_turn_cred);
linphone_nat_policy_set_stun_server_username(...);
...
```

To disable STUN and TURN support, the command `-sd` is used which clears the `LinphoneNatPolicy` object.

Registration

The `register` and `unregister` commands are used for registering and unregistering to and from a SIP proxy server. The user has to provide a valid SIP URI and password for the registration to be successful. These commands, as well as the `-s` command, are available only when the user is not in a call.

The `irc_bot_proxy.cpp` file implements the code for registering and unregistering. After checking, whether the user is already registered and whether the `-a` argument is specified, the credentials are stored in the class attributes and `irc_bot_proxy::bot_register()` method is called, as shown in Listing 5.2. The logic of processing `-a` arguments is closely described in Section 5.6.

```
if(proxy.proxy_cfg != nullptr && linphone_proxy_config_get_state
    (proxy.proxy_cfg) == LinphoneRegistrationOk)
{
    //already registered
    ...
}
...
int err = proxy.bot_register(core._core);
...
```

Listing 5.2: Check, whether the user is already registered.

The code for the `irc_bot_proxy::bot_register()` method is adapted from [4, Tutorials – Basic registration]. This tutorial also implements a proxy unregistration.

After the method is successfully performed, it is necessary to call `irc_bot_core::iterate()` method. The iterate function calls the `linphone_core_iterate()` method for a linphone core background SIP work until the registration is either successful or not, as shown in Listing 5.3.

After the user is successfully – proxy config's state is in `LinphoneRegistrationOk` – or unsuccessfully – proxy config's state is in `LinphoneRegistrationFailed` – registered, a feedback message is sent to the user. When the registration is successful, the current SIP URI is sent to the user in the confirmation message.

The unregistration is very similar. First, the program checks whether the user is registered or not, `irc_bot_proxy::bot_unregister()` method is called, and after the unregistration is successful (the proxy config is in the `LinphoneRegistrationCleared` state), a message with the user's current URI is sent.

```
/* Register */
while((linphone_proxy_config_get_state(proxy.proxy_cfg) !=
    LinphoneRegistrationFailed) && (linphone_proxy_config_get_state
    (proxy.proxy_cfg) != LinphoneRegistrationOk))
{
    core.iterate();
    usleep(20000);
}

/* Confirm the registration */
...
```

Listing 5.3: A while loop for performing the registration.

This URI and the "welcome" URI are in the form of peer-to-peer SIP URI, where after the "@" character is the user's IP address.

For the user to tell, whether he is registered or not, a `status` command is available. This command prints crucial information about the user, such as:

- Register state – whether registered or not and current SIP URI.
- NAT utilities – whether the STUN or TURN support is enabled or not.
- Current call and a list of calls.

5.4 Calls

The ability to make and receive calls is a crucial feature of this project. The third-party library, liblinphone, provides all of the functionality for SIP calls. The user is able to initiate peer-to-peer calls, with the help of NAT traversing utilities and also calls initiated through a SIP proxy server.

For the user to initiate an outgoing call, the `call` command is used. This command requires a mandatory argument in the form of a remote SIP URI. It also supports initiating a call with the help of an address book lookup with `-a` argument and an ENUM lookup with `-e` argument. Address book logic is described in Section 5.6 and the ENUM lookup implementation is described below in this section (5.4).

Initiation and termination of the calls are implemented in the `irc_bot_call.cpp` file and the argument checking, processing, and main calls loop are in the `irc_bot.cpp` file.

Initiating a call

After the `call` command is received and processed, the `irc_bot::call()` method is called. This method processes the arguments and initiates an outgoing call by calling the `irc_bot_call::call_invite()` method.

This method uses linphone functions to initiate a call. The functions in Listing 5.4 are used as recommended in the official tutorial [4, Tutorials – Basic calls].

```
//Initiate the call
this->_call = linphone_core_invite(lc, uri.c_str());
if(this->_call == nullptr)
{
    //failed
}
//Get a reference of the call if we want to work with it later
linphone_call_ref(this->_call);
```

Listing 5.4: Call initiation with linphone methods.

If the initial INVITE is successful, the current call is pushed into the `callsVector` vector and the program enters a call loop by calling the `irc_bot::call_loop()` method. The vector is used for keeping track of all outgoing and incoming calls. The main call loop consists of six logical sections:

1. The core iterate method.
2. Call established check.
3. Paused or resumed by the remote check.
4. A wrong URI check.
5. Check for incoming commands.
6. Call is still active check.

Not all commands available in "not-in-a-call" state are available during the call. These include `register, unregister` and `-s` commands. On the other hand, the call control commands, such as `hangup, hold` and `resume` are available. The `irc_bot::check_messages_during_call()` method has the same structure and command processing logic as the main program loop. This method is periodically called during the call to check whether the user sent a command or the server sent a `PING` message.

Liblinphone library implements various call states. These call states are used to control the call flow and when a state changes, the `call_state_changed` callback, implemented in the `irc_bot_core.cpp` file, is invoked and appropriate actions are taken. This program uses the callback for recognition when the bot receives an incoming call or when a call is terminated.

The call states are also used for an on-hold check, call established check, and an error check during the loop itself, without using the callback (steps 2, 3, and 4). The user is always notified by a private message when a major call event happens. Some of these checks (2, 3) also use a boolean flag variable, so that they can be called more easily or called just once.

```
// the auxCall variable holds the current call
if(!callsVector.empty())
{
    bool found = false;
    for(int i = 0; i < callsVector.size(); i++)
    {
        // check whether the current call is still in the vector
        if(auxCall._call == callsVector.at(i)._call)
        {
            // call is still active
        }
    }
    if(!found)
        // exit loop
}
else
    // exit loop
```

Listing 5.5: Active call check at the end of the call loop.

The user is also able to initiate or receive a call during another call. The previous call is always put on hold and the program enters a new call loop. For this purpose, the `callsVector` vector is used. Each new call is pushed into this vector and when either the remote or the user terminates a call, the specific call is removed from the `callsVector` vector in the `call_state_changed` callback. At the end of every loop, a check is performed, whether the call the loop belongs to is still in the vector. When the call is not in the vector anymore, the nested loop is exited and the loop of the previous call is entered. The code is shown in Listing 5.5.

The iterate method is very crucial for all linphone applications. This method does all the background tasks needed for the call to be active, such as receiving SIP messages, recording voice, playing audio, and so on.

Whenever the remote or the user terminates the call and this call is deleted from the `callsVector` vector, a notify message is assembled and assigned to the `remoteHungUp` variable. This callback also simultaneously deals with an incoming call being terminated, this behavior is described closely in the next subsection (5.4).

Contents of the `remoteHungUp` variable are checked at the beginning of each loop, either the main loop or the call loop with a simple `if` statement:

```
if(!remoteHungUp.empty())
{
    // send the notify message that the call is terminated
    string msg = "PRIVMSG " + user_nick + " :" + remoteHungUp + "\r\n";
    send_com(msg);
    remoteHungUp.clear();
}
```

Note that this program allows only one active, unpaused call at a time. The `status` command provides information on which call is current and a list of all calls that are put on hold, in case of multiple calls.

Receiving calls

Whenever a new call comes, the linphone core changes its call state to the `LinphoneCall StateIncomingReceived` state. This state change invokes the callback function in the `irc_bot_core.cpp` file.

This callback is used for assembling the notify message and pushing the incoming call structure into the `incomingCallsVector` vector. This structure is only used for recognizing whether incoming calls are still ringing or not.

```
struct IncCall
{
    LinphoneCall *call; //the incoming call
    int status; // flag, whether the call is still ringing or not
};
```

The contents of the notify message, located in the `incomingCallMessage` variable, are periodically checked at the beginning of each loop (either the main or call loop, just like the `remoteHungUp` message described in the previous subsection). When the variable is not empty, that means there is a new incoming call.

With each incoming call, the `order` attribute is incremented. This order acts as an index of the call in the vector. The user is able to choose which incoming call to accept in case of multiple incoming calls, and the order is always sent to the user in the notification message. The if statement is shown below in Listing 5.6.

```
if(!incomingCallMessage.empty())
{
    order += 1;
    // send the incoming call message to the user
    string msg = ... "Type \"accept " + to_string(order) + "\" to
        accept this call!\r\n";

    send_com(msg);
    incomingCallMessage.clear();
}
```

Listing 5.6: Incoming call notify procedure.

These message variables, as well as the vectors, are conceptualized as global variables. The reason is that the callback function is not a class method, so class attributes cannot be accessed.

The user may accept a call with the `accept` command. This command has a mandatory integer argument – the order of the call user wishes to accept.

Accepting a call is implemented in the `irc_bot::accept()` method. The method first checks the validity of the argument provided and then converts it into an integer, creating the index. The appropriate incoming call struct is obtained from the `incomingCallsVector` vector:

```
int index = stoi(messages[4]) - 1;
IncCall incCall = incomingCallsVector[index];
...
incomingCall = incCall.call;
```

With the help of `linphone_call_accept()` method, the call is accepted and a message of its acceptance is sent to the user. When the user accepts a call from multiple incoming calls, all of the other calls are declined. The decline logic is described below in this subsection. After declining other incoming calls, the `incomingCallsVector` and `order` are reset, the call is pushed into the `callsVector` vector, and the program enters the call loop, as shown below in Listing 5.7.

```
int ret = linphone_call_accept(incomingCall);

// parse the remote address from the call
...
// send a confirmation message
...
call._call = incomingCall;
// decline
...
incomingCallsVector.clear();
order = 0;

linphone_call_ref(call._call);
callsVector.push_back(call);
call_loop(...);
```

Listing 5.7: The process of accepting a call.

The user may also decline an incoming call with the use of `decline` command. This command declines all incoming calls and its behavior is implemented in the `irc_bot::decline _func()` method. Only active calls – with the `status` attribute equal to one – are declined. Liblinphone method `linphone_call_decline()` is used for declining the calls. The vector and `order` are also reset. This logic is also used in declining calls while accepting.

Listing 5.8 describes the logic of declining calls. The `irc_bot::decline _func()` method is called by the `irc_bot::decline()` function, which only contains error checking.

```
for(int i = 0; i < incomingCallsVector.size(); i++)
{
    aux = incomingCallsVector.at(i);
    if(aux.status == 1)
    {
        ret = linphone_call_decline(aux.call, LinphoneReasonDeclined);
        // send a message that the call was terminated
        ...
    }
}
```

Listing 5.8: The process of declining calls. This logic is also used in declining the other not accepted incoming calls.

When the remote terminates an incoming call while still ringing, the `call_state_changed` callback is also invoked. The callback changes the `status` of the call to inactive (value 0) and also assembles the notify message that the remote terminated the call.

Call control

The user is able to control the flow of a call with `hangup, hold` and `resume` commands.

The `hangup` command is used for terminating the current call. The behavior for this action is implemented in the `irc_bot_call::call_terminate()` method. Liblinphone method for terminating a call is `linphone_call_terminate()`. After the user terminates the current call, `remoteHungUp` message is shown to the user. The logic of assembling this message is described above in Subsection 5.4.

```
if(this->_call != nullptr)
{
    linphone_call_terminate(this->_call);
}
```

For putting the call on hold from the user's side, the `hold` command is used. After receiving the command, the program checks, whether the call can be put on hold, then holds it. The adequate liblinphone method for holding calls is the `linphone_call_pause()` method. User is also notified that the call is going to be put on hold.

After successfully putting the call on hold, the user is also able to resume this call with the `resume` command. After receiving this command, the program checks whether the call can be resumed and resumes it. The user is again notified that the call is being resumed. The adequate liblinphone method is called `linphone_call_resume()`.

ENUM lookup

This user agent also supports initiating calls with an E.164 format number. A DNS NAPTR lookup is used for retrieving the mapped SIP URI. This implementation's logic draws from Subsection 2.1.5.

To use this feature, the user has to provide `-e` argument with a valid E.164 number in the `call` command. The functionality is implemented in the `addr_book::get_enum_uri()` method. This method is divided into three logical subsections:

1. Modify the number to be DNS-searchable.
2. Execute the DNS NAPTR lookup.
3. Execute the regex substitution.

The first step for a valid DNS-searchable phone number is to reverse it and put dots (".") in between. For this purpose, a backward `for` loop is implemented. Only numeric values from the original number are used.

```
// number contains the provided number
for(int i = number.length() - 1; i >= 0; i--)
{
    if(!isdigit(number[i]))
        continue;
    // revNum contains the reversed number
    revNum.push_back(number[i]);
    revNum += ".";
}
```

After the number is reversed, an "e164.arpa." suffix must be added:

```
revNum += "e164.arpa.";
```

To retrieve the DNS NAPTR record, the `resolv.h` C/C++ library and its functions are used [2]. The `res_search()` is used for the NAPTR query and the answer is copied to the `response` variable. This answer has to be parsed first. For this, the `ns_initparse()` method fills the according buffer with data and parses them.

```
int responseLen =res_search(qry, ns_c_in, ns_t_naptr,
    (u_char *)&response,sizeof(response));
...
if (ns_initparse(response.buf, responseLen, &handle)<0)
{
    perror("Error: ");
    return "";
}
```

Now the answer has to be looped through to find the appropriate data. When the type of record is NAPTR, the `rdata` section is copied to an auxiliary buffer. An offset of 11 bytes must be used for correct data to be copied. The last step of this resolution is to check whether the auxiliary buffer contains a SIP URI.

```
for (rrnum=0;rrnum<(ns_msg_count(handle,section));rrnum++)
{
    ...
    if (ns_rr_type(rr)==ns_t_naptr)
    {
        // retrieve the rdata
        memcpy(&res, ns_rr_rdata(rr) + 11, sizeof(res));
        aux = string(res);
        if (aux.find("sip") != string::npos)
        {
            /* The SIP NAPTR RR was found */
            found = true;
            break;
        }
    }
}
```

The `aux` variable contains a "regexp" [13] string in POSIX that is used for substitution. Its format is as follows:

```
!<regex>!<substitution>!
```

This string is split into the `regex` and `substitution` parts and is stored in the `regexes` vector. The `number` is then substituted with the help of regular expression substitution in extended POSIX. The C++ regex library is used for this and the corresponding process is shown below:

```
regex re(regexes[0], regex_constants::extended);
uri = regex_replace(number, re, regexes[1], regex_constants::
    match_default | regex_constants::format_sed);
```

The final SIP URI corresponding to the provided phone number is stored in the `uri` variable and is returned by the function. Whenever an error occurs or the number does not map to any SIP URI, the function returns an empty string. Returned URI is checked in the `irc_bot_call::call()` method, whether it is empty or not, and then passed into the `irc_bot_call::call_invite()` method.

5.5 Instant messages

The user may send an instant SIP message with the use of a SIP MESSAGE method. This user agent supports two types of instant messages:

1. Send a message to a user, specified with a URI.
2. Send a message directly to the other participant in the current call.

To send an instant message to any user, specified with a URI, the `mess` command is used. The destination URI must be passed as the second argument of the command. This command also supports address book lookup with the `-a` argument, closely described in Section 5.6.

The instant messaging implementation is in the `irc_bot_message.cpp` file. To be able to send messages, the liblinphone library creates a so-called "chat rooms". The usage of these methods was inspired from [4, Tutorials – Chat room and messaging].

After the command is processed and its arguments checked, the `irc_bot_message::create_chat_room()` method creates a chat room with the specified remote user, as shown below:

```
this->uri = uri.c_str();
LinphoneAddress *addr = linphone_core_create_address(lc, this->uri);
this->chat_room = linphone_core_get_chat_room(lc, addr);
```

Next, the text message is created by concatenating the arguments that have an index greater than five or six (based on whether the `-a` argument is present or not). This complete message is passed to the `irc_bot_message::send_message()`, where a SIP MESSAGE is created and sent to the remote user. The code snippet below illustrates the process.

```
LinphoneChatMessage *chat_message = linphone_chat_room_create_message
    (this->chat_room, msg.c_str());
linphone_chat_message_send(chat_message);
```

The liblinphone supports creating chat rooms associated with a call. The user does not have to specify the remote URI and the message is sent directly to the other participant. The command for sending direct messages during a call is `-m`. This feature was inspired by [4, Tutorials – Real Time Text Receiver].

The corresponding chat room with the other participant is created when a call is established by calling the `irc_bot_message::create_call_chat_room()` method.

After receiving the `-m` command during a call, a text message is created in the same way as in the `mess` command and the `irc_bot_message::send_message()` method is called to create and send the message.

5.6 Address book

The address book is a local database for the user to save contacts and proxy identities, so the user does not have to type long URIs and passwords all the time. The database is implemented in SQLite3 C/C++ library and the corresponding code is located in the `addr_book.cpp` file. The code and routines were drawn from [20].

Initializing

The first step of working with the database is to open a database handle. This action is done right at the start of the program with the help of `addr_book::addr_book_open()` method, which calls the `sqlite3_open()` routine. The database handle is then stored in the `db` attribute.

After the opening is successful, tables `contacts` and `registrar` are created with the `addr_book::addr_book_create()` method, whose code snippet may be found below in Listing 5.9. Tables are created only when they do not exist. A `SELECT` statement is used for this check.

When they do not exist, they are created with the use of `CREATE TABLE` statement. This statement also includes a second layer of security in the form of `IF NOT EXISTS`.

```
sql = "CREATE TABLE IF NOT EXISTS CONTACTS(" \
  "ID INTEGER PRIMARY KEY," \
  "NAME CHAR(150) NOT NULL," \
  "URI CHAR(150) NOT NULL);";
/* Execute SQL statement */
rc = sqlite3_exec(this->db, sql, callback, 0, &zErrMsg);
...
sql = "CREATE TABLE IF NOT EXISTS REGISTRAR(" \
  "ID INTEGER PRIMARY KEY," \
  "NAME CHAR(150) NOT NULL," \
  "URI CHAR(150) NOT NULL," \
  "PASSW CHAR(150) NOT NULL);";

/* Execute SQL statement */
rc = sqlite3_exec(this->db, sql, callback, 0, &zErrMsg);
...
```

Listing 5.9: Creation of the tables.

Both tables contain `ID, NAME` and `URI` attributes and the `registrar` table also contains an additional `PASSW` attribute. The `NAME` attribute is unique within the corresponding table and is used and intended for data retrieval. User may also manually create tables with the `-c` command.

The database may also be dropped with the `-dropdb` command. This command invokes the `addr_book::addr_book_drop()` method. The code snippet below describes the process of creating both `DROP` statements and executing them.

```
sql = "DROP TABLE IF EXISTS CONTACTS; \
       DROP TABLE IF EXISTS REGISTRAR;";
/* Execute SQL statement */
rc = sqlite3_exec(this->db, sql, callback, 0, &zErrMsg);
```

When the program is about to be shut down, it is needed to close the database handle. The `addr_book::addr_book_close()` method is called, containing the `sqlite3_close()` routine.

To process the address book commands, the `addr_book::addr_book_iterate()` method is used. This method is called at the end of either the main or the call loop. It compares the `command` variable with each possible address book command and performs adequate action.

The address book also constructs its own feedback messages. These messages are assigned to the `dbMessage` class attribute and then sent through a `PRIVMSG` to the user as a form of feedback.

Inserting data

To insert some data into the address book, the `-ic` and `-ir` commands for **i**nserting into **c**ontacts or **i**nserting into **r**egistrar are available. Both of these commands invoke the `addr_book::addr_book_insert()` method.

Before the data are inserted, a check whether a record with the same `NAME` already exists is done. This is provided by the `addr_book::addr_book_check()` method. This method runs a `SELECT` statement and tries to retrieve data with the same `NAME`.

When it is ensured that the record does not already exist, the `INSERT` statement is assembled based on the `Option`.

```
enum Option
{
    Contact = 0,
    Registrar = 1
};
```

The `Option` enum determines which insertion type is to be done. This enum also applies to the `addr_book::addr_book_check()` method for the function to determine which table to look in. The enum option is passed to the function via argument. So the function to insert data into contacts is called like this, for example:

```
this->addr_book_insert(messages[4], messages[5], aux1, Contact);
```

The option is then checked with an if-else statement and appropriate action is executed:

```
if(opt == Contact)
{
    if(this->addr_book_check(name, Contact))
    {
        this->dbMessage = "Contact already exists!";
        return 1;
    }
    sql_str = "INSERT INTO CONTACTS (NAME,URI) " \
    "VALUES ('"+ name + "', '" + uri + "');";
}
```

The else branch is very similar. After evaluation, the statement is executed with the `sqlite3_exec()` method and a feedback message is sent to the user.

```
/* Insert */
int rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
```

Updating data

The user may update the contents of tables with the `-uc` and `-ur` commands. These commands are used for **u**pdating **c**ontacts and **u**pdating **r**egistrar, respectively. Both of these commands invoke the `addr_book::addr_book_update()` method. Again, the `Option` enum determines which table is worked with.

Before a contact is updated, it is needed for its existence to be checked. The `addr_book ::addr_book_check()` method is used for this. Afterward, an `UPDATE` statement is assembled and executed.

Updating a registrar identity is a bit different. The user has a choice to update only the `URI` or only the `PASSW` or both. To achieve this, the user types a single dash ("-") instead of the specific argument. The existence of the record is checked and a proper `UPDATE` statement is assembled, based on whether the user wishes to change `URI, PASSW` or both. Example:

```
if(passw == "-" && uri != "-")
{
    sql_str = "UPDATE REGISTRAR set URI='" + uri + "' where
        NAME='" + name + "';";
}
```

Removing data

The user may also delete data from the address book. The logic and implementation are similar to inserting and updating data.

The first step is that the existence of the record is checked, then this record is deleted with a `DELETE` statement. The `Option` enum is again used to determine the table to work with. The commands to **r**emove a record from **c**ontacts or **r**emove from the **r**egistrar are `-rc` and `-rr`.

Retrieving data

The address book data are used for the user to initiate calls, register to a proxy, and send messages faster. Whenever the `-a` attribute is provided within the `call, register` or `mess` commands, it is a sign for the program to take a look in the address book and retrieve some data. The `-a` attribute requires a `NAME` argument which must be unique within the specific table. This `NAME` is then used for data retrieval.

There are two data retrieval methods, one for contacts and the other for registrar. The `call` and `mess` commands invoke the `addr_book::addr_book_get_contact()` method.

The method first checks the existence of the contact and then builds a `SELECT` statement. The statement shall return only one record and the URI is then assigned to `contactUri` class attribute, as shown in Listing 5.10. This class attribute is passed into the `irc_bot_call:: call_invite()` or the `irc_bot_message::create_chat_room()` methods through the `uri` string variable.

The `addr_book::addr_book_get_registrar()` method for retrieving the identity credentials has the same structure as the method above. The only difference is that this method has to also retrieve a password. This difference is shown Listing 5.11. The credentials are stored in the `regUri` and `passw` class attributes, which are then set as the `irc_bot_proxy` credentials.

```
sql_str = "SELECT URI from CONTACTS where NAME='" + name + "';";
....
int result = sqlite3_prepare_v2(db, sql, -1, &selectstmt, NULL);
if(result == SQLITE_OK)
{
    /* The SELECT shall always return only one row because the name
    is always unique. */
    if (sqlite3_step(selectstmt) == SQLITE_ROW)
    {
        this->contactUri = string(reinterpret_cast<const char*>
            (sqlite3_column_text(selectstmt, 0)));
    }
}
```

Listing 5.10: Retrieval of a contact from the database.

```
sql_str = "SELECT URI,PASSW from REGISTRAR where NAME='" + name + "';";
...
int result = sqlite3_prepare_v2(db, sql, -1, &selectstmt, NULL);
if(result == SQLITE_OK)
{
   if (sqlite3_step(selectstmt) == SQLITE_ROW)
   {
      /* Retrieve two sets of data */
      this->regUri = string(reinterpret_cast<const char*>
        (sqlite3_column_text(selectstmt, 0)));
      this->passw = string(reinterpret_cast<const char*>
        (sqlite3_column_text(selectstmt, 1)));
   }
}
```

Listing 5.11: Retrieval of an identity from the database.

Browsing data

The user may browse its contacts and identities with a database browsing mode. This mode shows the user's data divided into pages. The page always has five entries at a time. This mode has its own loop and supports only three commands: `exit`, `next` and `prev`.

The database browsing mode is entered with the use of `-con` or `-reg` commands for browsing **con**tacts or **reg**ister identities, respectfully. These commands are processed at the end of the main and call loop. The user may also specify a pattern, acting as a regular expression. These patterns conform to SQLite standards, with the "%" and "_" characters as wildcards.

First, it is needed to retrieve the corresponding data with the `addr_book::addr_book_get_data()` method. The `Option` enum is used to determine which table is worked with and the corresponding `SELECT` statement is assembled, based on, whether the user specified a pattern. When the user specified a pattern, the `LIKE` statement is added. Note that this pattern corresponds to the `NAME` attribute.

The retrieved data are stored in the `Data` struct, which is then stored in the `dbData` vector, as shown in Listing 5.12. Only `URI` and `NAME` attributes are retrieved either for contacts or registrar.

```
// the struct definition in the header file
struct Data{
    string name;
    string uri;
};
// the data retrieval and storing
while(sqlite3_step(selectstmt) == SQLITE_ROW)
{
    data.name = string(reinterpret_cast<const char*>
        (sqlite3_column_text(selectstmt, 0)));
    data.uri = string(reinterpret_cast<const char*>
        (sqlite3_column_text(selectstmt, 1)));
    this->dbData.push_back(data);
}
```

Listing 5.12: Data retrieval and storing into the Data structure.

After the data are retrieved, the `irc_bot::addr_book_print()` method is called. The program enters a browsing loop. This loop is used for processing certain commands and the `PING` messages.

The `exit` command quits the database browsing mode. To scroll through the pages `next` and `prev` commands are used. These two commands accept an optional argument, which specifies the number of pages to be skipped. If the argument is not specified, the number of pages skipped is implicitly one.

After processing the argument, starting and ending vector indexes of the data entries that are to be printed, and the next page number, are calculated, as shown in Listing 5.13.

```
/* Increase actual page number */
pact += number;
/* If it overflowed then correct it */
if(pact > pmax)
    pact = pmax;
/* Compute the next starting index */
n += number * 5;
/* If it overflowed then correct it */
if(n > size)
    n = lastPageIndex;
/* Compute the last data index */
x += number * 5;
/* If it overflowed then correct it */
if(x > size)
    x = size;

i = print(n, x, i, pact, pmax, addrBook);
```

Listing 5.13: The process of calculating the parameters for the next page.

The `prev` command has a very similar structure. Finally, the `irc_bot::print()` function then prints the appropriate data to the user in the form of private messages. This function also prints the actual page number, and maximum page number and returns an index it ended at. The user is still able to maintain the call during the database browsing but is unable to hang up or control it.

Chapter 6

Application tests

This chapter describes the implementation of the automated application tests. These tests are used for testing the basic features of the program. Other edge cases were tested separately.

Tests are in a form of a Python script. A special tool, called SIPp [10], that generates SIP traffic is used for testing calls.

The first Section 6.1 describes the logic and implementation of the script and test scenarios. Section 6.2 then describes a comparison of this program with other user agents.

6.1 Tests script

The script is conceptualized as another IRC bot that simulates the actions of a human user. It connects to the same IRC server and channel as the user agent and sends direct messages to it.

The script tests only basic user agent scenarios and some of the tests are optional, launched only if an argument is present. The script is implemented in `test.py` file and launched in this format:

```
python3 test.py {server} {channel} [-r URI passw] [-s STUN_server
    TURN_user TURN_passw]
```

The `server` and `channel` are mandatory and run only basic test scenarios – testing the UA's connection, receiving and making calls, and working with the local database.

With the `-r` argument user may also run proxy registration tests. Valid SIP URI and password must be provided.

The last `-s` argument runs also the NAT traversal utilities setting tests. Again, valid server and TURN credentials must be provided.

After launching the script and the arguments are processed, they are assigned into adequate variables. The bot is then connected to the specified `server` and `channel` with Python sockets. Bot's nickname is `SIPTest`. The routine for connecting the bot is similar to connecting the user agent, as inspired by [1]. After being successfully connected, the script runs the desired test scenarios.

Test scenarios

The total number of test scenarios is 14. Each test scenario consists of multiple test cases, totaling 39 test cases. Three test scenarios are optional.

The concept of the test scenarios is to send a command to the user agent and test that it sent a correct response message. After sending a command, a `test_loop()` method is called, which awaits the response. The receiving socket waits fifteen seconds before timing out and flagging the test case as failed. The timeout consists of three, five-second socket timeouts. The general logic of each test case is as follows:

1. Prepare the test case (prepare commands, run external programs, etc.).
2. Send a command.
3. Wait and receive a response.
4. Check the response.
5. Print, whether the test case succeeded or not.

After receiving a response, it is checked, whether it is a `PING` message from the server (by calling the `pong_response()` method) or whether it is the correct user agent response. The contents of the message are checked with a regular expression, whose string is passed through the function argument. The inside of the loop is as follows:

```
# wait for response
try:
    result = recv_timeout(s, 1024)
except TimeoutError:
    aux += 1
    if aux == 3:
        # test failed
        ...
    continue

# print the received result
print(result)
pong_response(s, result)

if re.search(r'(.*)PRIVMSG ' + nick + ' :' + control_mess + '(.*)',
result, re.MULTILINE):
    # test passed
    ...
    break
```

Listing 6.1: The inside of the loop used for receiving and checking the correct response.

Whether the test passed or not is printed to the standard output with the use of colored text. Successful test cases are printed in green and failed in red. Before every test scenario, a header is printed in purple, containing a brief description of the following test. A version of this function that checks for two possible responses is available, called `test_loop_or()`.

There is an exception in the timeout duration at the first test scenario, which will be described in the next subsection.

During the execution of the tests, a CTRL+C interrupt is possible. The interrupt correctly kills all subprocesses created by the script and disconnects the test bot from the server.

At the end of all test scenarios, a final summary is printed to the user. The summary contains a number of passed and failed tests, and a total number of possible test scenarios and test cases.

Launch

This is the first test scenario, that tests the user agent's launch and connection to the server. It is implemented in the `test_bot_connect()` method.

This test scenario first launches the `irc_bot` user agent with the use of Python's subprocess module. The method then waits for one minute maximum to receive a response in the form of a welcome message. Reason for the timeout being this long is that the process of connecting might sometimes take a while. The structure of the loop is similar to `test_loop()` method.

This method is crucial and without the user agent being available no other tests may be conducted. So as opposed to other test cases, this test case exits the script when it fails.

Address book operations

The next two test scenarios are regarding insert, update and delete address book methods. The first scenario tests these methods with the `contacts` table and it is implemented in the `test_crud_contact()` method. The function contains these test cases:

- `-ic` method – inserts a new contact.
- `-uc` method – updates the same contact that was just added.
- `-rc` method – removes the test contact that was just added.

The registrar table is tested with the same logic, implemented by the `test_crud_registrar()` method:

- `-ir` method – insert a new register identity.
- `-ur` method – update just the SIP URI of the same identity that was just added.
- `-rr` method – delete the identity that was just added.

The insert methods are checked with the `test_loop_or()` method. Valid responses are "Record created successfully!" and "Record already exists!". Other methods only accept the "Record **action** successfully!" response. Note that a failed test no longer exits the script.

Calls

Most of the test scenarios revolve around testing the basic feature of this program – calls. In total, seven test scenarios deal with basic call cases this program supports.

To initiate or receive a call, the `SIPp` command line application is used. This tool generates SIP traffic and simulates another user agent.

This tool is always launched with the use of Python's subprocess module. It is needed to create custom SIP scenarios for some test cases. These scenarios are written in XML and can be found in the `test/` folder.

The SIPp tool is launched bound to the loopback IP address, in the background mode and with the scenario set. An example of launching SIPp with the User Agent Server scenario (uas – awaits for SIP methods, `-sn` argument), only one call to be made (`-m`) and in the background mode (`-bg`):

```
subprocess.Popen("cd sipp-3.3 && ./sipp -sn uas -m 1 -bg", ...)
```

The first test scenario tests a basic call ability. The SIPp tool is launched in a style described in a few lines above. This scenario is implemented in the `test_call()` method and the test cases are:

- Initiate a call – send the `call` command with `sip:127.0.0.1:5061` as the remote URI. Wait for the "Calling **URI**!" and "A call with **URI** is established!" responses.
- Terminate the call – send the `hangup` command and await the confirmation response the call was terminated.

The next test scenario, implemented in the `test_call_w_lookup()` method, tests the `call` command ability to dial from the contacts and with the use of a phone number. The SIPp tool is not needed for this scenario. The test cases are as follows:

- Create a record with a non-valid URI and initiate the call with the use of `-a` argument. This case also tests a failure of a call.
- Initiate a call to the given phone number. The number provided is a valid one and shall have a DNS NAPTR record. This case tests the correct lookup, establishing and terminating the call.

The next two test scenarios test accepting and declining an incoming call. The corresponding methods are called `test_incoming_call()` and `test_decline_call()`. The SIPp tool is used with the User Agent Client scenario bound to the loopback IP address.

- Wait for the bot to send an incoming call notification message.
- Accept the call with `accept 1` command and wait for "A call with **URI** is established!" response. The call is then terminated and a termination confirmation message is also awaited.
- Decline an incoming call with the `decline` command and await for the "Declined all calls!" confirmation message.

The fifth test scenario tests the program's ability to control a call flow – hold and resume. This scenario requires a custom SIPp scenario that may be found in the `hold_res.xml` file. The method this scenario is implemented in is called `test_hold_resume()`. The test cases are as follows:

- Initiate a call after two seconds put it on hold. Await the "Holding call **URI**!" response.
- Resume the call and again await the confirmation message.

The sixth test scenario tests the program's ability to initiate more than one call. This scenario initiates a call from an already established call and is implemented in the `test_call_in_a_call()` method. For this, two custom SIPp scenarios are created and can be located in the `call_A.xml` and `call_B.xml` files. The test cases:

- Initiate the first and second calls and await the call established confirmation message.
- Terminate the second call and await the call terminated response.
- The first call shall be put on hold, so this test case tries to resume the call and await the call resumed response.
- Last test case terminates the first call.

The last call-oriented test scenario tests the incoming call stacking ability in the `test_multiple_inc_calls()` method. This scenario receives three incoming calls and accepts the middle one. Three SIPp tools are launched with the User Agent Client scenarios and bound to the loopback address with different ports (5061, 5062, and 5063). The test case is:

- Await for three incoming calls and accept the middle one with the `accept 2` command. Await the "A call with **URI**:5062 is established" response.

At the end of all test scenarios, the SIPp instances are always terminated with the `os.killpg` method. This is just another layer of security for the script to not leave active processes. The processes should terminate themselves.

Messages

The script also tests receiving a SIP MESSAGE method with the use of the SIPp tool. A custom SIPp scenario that creates the SIP MESSAGE method was created and is located in the `mess.xml` file.

The test scenario is implemented in the `test_message()` method. This scenario is made up of three test cases:

- Receive a message – SIPp tool sends a SIP MESSAGE method with a "Hello!" text.
- Send a message – the message is sent to loopback. "Sending a message" response is awaited.
- Send a message with an address book lookup – a new contact containing the loopback IP address is created and a message is sent to this contact. "Sending a message" response is again awaited and the new contact is removed afterward.

Registration

The next two test scenarios are optional and can be triggered by providing the `-r` argument and valid proxy credentials. The SIPp tool is not used.

The first scenario is implemented in the `test_register()` method and tests basic registration with the `register` command. The test cases are:

- Send the `register` command with the provided credentials and await a confirmation response.

- Unregister from the proxy and await a confirmation response.

The second scenario tests the same behavior but with the use of the address book lookup. It is implemented in the `test_register_w_lookup()` method and the test cases are:

- Create a record and register. Await a confirmation response.
- Unregister from the proxy and await a confirmation response. Delete the newly created record.

NAT utilities

The last set of test cases is implemented in the `test_stun_turn()` method. This scenario tests the program's ability to set the NAT traversal utilities.

This scenario is also optional and may be triggered by providing the `-s` argument, valid server, and TURN credentials. The test cases are:

- Enable STUN and ICE. Await the "STUN enabled!" response.
- Enable also TURN. Await the "STUN and TURN enabled!" response.
- Disable the NAT settings. Await "STUN/TURN disabled!" response.

6.2 Comparison

The final program was properly tested for any edge cases and basic functionalities and afterward was compared with two other SIP user agents – Linphone and Jitsi[1]. `IRCPhone` was chosen as the official name of this project.

Linphone and Jitsi have their GUIs and do not use commands for controlling the UA. The IRCPhone, just like the other two UAs, supports basic SIP features – calls, registration, and instant messages. All three UAs have a contacts list (address book) and can save the proxy identities, though the IRCPhone has to first manually unregister to switch accounts.

The biggest setback of the IRCPhone is that it does not support video calls and local conferences. Although, most of the SIP proxies nowadays provide conference rooms.

As opposed to Linphone and Jitsi, IRCPhone supports ENUM lookup itself. Usually, the DNS NAPTR query is performed by a proxy the user is registered to. IRCPhone performs this query itself and initiates the call with the returned SIP URI.

IRCPhone is suitable for users that use IRC frequently and want to also initiate calls with their IRC buddies. The IRCPhone is easily integrated into the user's favorite IRC client and no other user agent has to be downloaded.

[1] https://desktop.jitsi.org/

Chapter 7

Conclusion

This thesis described the creation of a Session Initiation Protocol user agent which uses an Internet Relay Chat client as its graphical interface. The purpose is to connect these two technologies to easily make calls within the IRC network.

The individual technologies used in this thesis were introduced, such as the SIP itself, the IRC technology, and already existing applications, which helped to create the final program.

In the next two chapters, the formal requirements have been summarized, and based on these requirements, the third-party library and IRC client – which acts as the GUI – were chosen and the program's final architecture was created.

The fifth chapter describes how this architecture was implemented into the chosen IRC client with the help of the chosen third-party library. The final program was properly tested and an automated acceptance tests script was created for the users themselves to also test the basic functionality of the final product. The implementation of this script is described in the penultimate sixth chapter.

A final important requirement is to release this program as an open-source project. The name "IRCPhone" was chosen as an official name for this project and it was released on GitHub[1]. The program is distributed with the help of GNU Autotools.

Possible future extensions for this project are video call support implementation, the possibility of being able to create local conferences, and also multi-platform support. For future user experience-oriented extensions, an improvement of the database browsing mode could be implemented as a local web server that projects the database data, for example.

[1] https://github.com/DavidKocman36/IRCPhone

Bibliography

[1] Agarval, M. *Pyhon IRC Bot – A Hands-on Tutorial with Example* [online]. TechBeamers, 2018 [cit. 2023-03-29]. Available at: https://www.techbeamers.com/create-python-irc-bot/.

[2] Albitz, P. and Liu, C. *C Programming with the Resolver Library Routines (DNS and BIND, 4th edition)* [online]. O'Reilly & Associates, april 2001 [cit. 2023-03-28]. Available at: https://docstore.mik.ua/orelly/networking_2ndEd/dns/ch15_02.htm.

[3] Baresip Foundation. *Baresip* [online]. 2010 [cit. 2023-03-21]. Available at: https://github.com/baresip/baresip.

[4] Belledone Communications SARL. *Liblinphone: Modules* [online]. 2020 [cit. 2023-03-24]. Available at: https://download.linphone.org/releases/docs/liblinphone/latest/c/modules.html.

[5] Belledone Communications SARL. *Linphone* [online]. 2020 [cit. 2023-03-21]. Available at: https://www.linphone.org/technical-corner/linphone.

[6] Bidgoli, H. *The Internet Encyclopedia, Volume 2 (G – O)* [online]. 2nd ed. Wiley, 2004 [cit. 2023-03-20]. The Internet Encyclopedia. ISBN 9780471689966. Available at: https://books.google.cz/books?id=gZ9srwU_9xMC.

[7] Bitlbee. *Bitlbee* [online]. 2002 [cit. 2023-03-21]. Available at: https://www.bitlbee.org/main.php/news.r.html.

[8] Flanagan, W. A. *VoIP and Unified Communications: Internet Telephony and the Future Voice Network*. 1st ed. Wiley, 2012. ISBN 978-1-118-01921-4.

[9] Gamblin, T. *C++ – Using strtok with a std::string – Stack Overflow* [online]. 2008 [cit. 2023-03-26]. Available at: https://stackoverflow.com/a/289365.

[10] Gayraud, R. and Jacques, O. *SIPp* [online]. 2004 [cit. 2023-03-30]. Available at: https://sipp.sourceforge.net/doc3.3/reference.html.

[11] Helleu, S. *Weechat* [online]. 2003 [cit. 2023-03-21]. Available at: https://weechat.org/.

[12] Kalt, C. *Internet Relay Chat: Architecture* [RFC 2810]. RFC Editor, 1. april 2000 [cit. 2023-03-20]. DOI: 10.17487/RFC2810. Available at: https://www.rfc-editor.org/rfc/rfc2810.html.

[13] Mealling, M. H. *Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database: NAPTR RR Format* [RFC 3403]. RFC Editor, october 2002 [cit. 2023-03-28]. DOI: 10.17487/RFC3403. Available at: https://www.rfc-editor.org/rfc/rfc3403#page-5.

[14] Oikarinen, J. and Reed, D. *Internet Relay Chat Protocol* [RFC 1459]. RFC Editor, may 1993 [cit. 2023-03-20]. DOI: 10.17487/RFC1459. Available at: https://www.rfc-editor.org/rfc/rfc1459.

[15] Perkins, C., Handley, M. J. and Jacobson, V. *SDP: Session Description Protocol* [RFC 4566]. RFC Editor, july 2006 [cit. 2023-03-16]. DOI: 10.17487/RFC4566. Available at: https://www.rfc-editor.org/rfc/rfc4566.html.

[16] Schooler, E., Rosenberg, J., Schulzrinne, H., Johnston, A., Camarillo, G. et al. *SIP: Session Initiation Protocol* [RFC 3261]. RFC Editor, july 2002 [cit. 2023-03-15]. DOI: 10.17487/RFC3261. Available at: https://www.rfc-editor.org/rfc/rfc3261.

[17] Schulzrinne, H., Casner, S. L., Frederick, R. and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications* [RFC 3550]. RFC Editor, july 2003 [cit. 2023-03-17]. DOI: 10.17487/RFC3550. Available at: https://www.rfc-editor.org/rfc/rfc3550.html.

[18] Schulzrinne, H., Rosenberg, J., Campbell, B., Gurle, D. M. and Huitema, C. *Session Initiation Protocol (SIP) Extension for Instant Messaging* [RFC 3428]. RFC Editor, december 2002 [cit. 2023-03-16]. DOI: 10.17487/RFC3428. Available at: https://www.rfc-editor.org/rfc/rfc3428.

[19] Sotomayor, B. *Example IRC Communications – The UChicago χ-Projects* [online]. The University Of Chicago, 2010 [cit. 2023-03-20]. Available at: http://chi.cs.uchicago.edu/chirc/irc_examples.html.

[20] Tutorialspoint. *SQLite – C/C++* [online]. 2020 [cit. 2023-03-29]. Available at: https://www.tutorialspoint.com/sqlite/sqlite_c_cpp.htm.

Appendix A

Contents of the included storage media

```
root
├── db ..... Folder for storing the local database.
├── doc ..... Folder containing the thesis.
│   ├── src ..... Folder containing the thesis's source codes.
│   └── xkocma08.pdf ..... The thesis.
├── grammars ..... Folder containing important grammar files.
├── linphone-sdk ..... The third-party library.
├── sounds ..... Folder containing the phone's sounds, such as ringtone.
├── src ..... Folder containing the source files.
├── test ..... Folder containing the application tests.
│
├── irc_bot ..... The program's binary
├── LICENCE.txt ..... GNU Affero licence.
├── Makefile.am ..... Makefile template for generating Makefile.
├── manual.txt ..... File containing the commands and their usage
├── packages.sh ..... Bash script for checking the program's dependencies.
├── README.txt ..... The installation and launching manual.
├── configure.ac ..... Template of the configure file
└── build.sh ..... Script used for invoking the Autotools commands.
```

Appendix B

Program manual

This appendix contains all of the possible commands the user agent accepts, also available in the `manual.txt` file.

```
The commands and examples used in IRCPhone.
IMPORTANT: Do not accidentally flood your BOT. It might stop
receiving messages!!
TIP: Use mostly servers where registration is not required.
For example irc.libera.chat

Launching in terminal:
./irc_bot {ip/server} {channel} {user} {password}

- ip/server : ip address or domain of the IRC server.
- channel : channel on the server where the user will be present.
    The format is "#channel" or just channel.
- user : Nick of the user the bot shall listen to.
    Nick of the bot therefore will be "user_b".
- password : Bots password. Sometimes not needed but still mandatory.

After the Hello! message you are able to use these commands. The bot
shall ALWAYS give the user feedback in a form of a short message.


The commands are sent in a form of an IRC private message. After the
bot sends a welcome message, a new chat window shall open.
To communicate, just type

    <command> <arguments>

For communicating outside of the chat window, use

    /msg nick_b <command> <arguments>

Below is the list of all available commands:
```

```
- register {-a <name>} | {<uri> <password>}
    - Registers at a proxy with provided uri and password. You are also
      able to register with uri and password stored in the database
      using -a and unique <name>.

    - example: register sip:username@domain password or register -a joe

- unregister
    - Unregisters at the proxy.

- call {-a <name>} | {-e <number>} | {<uri>}
    - Initiates a call to <uri>. If you have the person stored in your
      contacts you might use -a <name>.
    - Also you are able to make peer-to-peer calls
      with sip:username@ip-address
    - The ENUM lookup is also available with argument -e.
      Just type the phone number in an E.164 format.

    -example: call sip:username@domain
      or call -a john or call -e +431123456789

- accept {<number>}
    - User is informed if an incoming call comes. User then might accept
      this call by typing accept <number> where <number> is the order
      of the call in which the call comes.
      The order is displayed to the user.
      All other active incoming calls are declined.

- decline
    - Decline an incoming call/calls.

- status
    - Displays current status of the bot. If it's registered, if it's in
      an active call, the list of all calls (paused and active) and
      whether STUN or TRUN is enabled.

- help {<option>}
    - Displays list of all usable commands. Use <option> for a command
      category.
    - Options   1. o - options
                2. c - call commands
                3. r - register commands
                4. m - direct messages commands
                5. a - address book commands
                6. dl - database list commands (contacts/proxy
                        ids browsing)

    - example: help o
```

```
- -s {<address>} [-t {<username>} {<passw>}]
    - Enables ICE, STUN and sets the address of the STUN/TURN server (-s)
        and if desired enables TURN and authenticates at the TURN server
        with provided <username> and <password> (-t).
        The -t argument is optional.

    - example: -s stun.domain.com -t username password

- -sd
    - Disables STUN/TURN option.

- end
    - Terminates the bot.

- mess {-a <name>} {<text>} | {<uri>} {<text>}
    - Sends a direct message to the given <uri>.
        If the remote is in contacts the user may use -a <name>.

    -examle: mess -a johnny Hey how are you?
        or mess sip:username@domain Hey how are you?

When a call is initiated, you may also use these additional commands:

- hold
    - Puts the current call on hold.

- resume
    - Resumes the current call.

- hangup
    - Hangs up the current call.

- -m {text}
    - Sends a direct message to the user you are in call with.

On the other hand, these are not available during call:
- register, unregister, -s -t, end

NOTE: Bot can also be terminated by CTRL+C in terminal.

Address book commands are ubiquitous and their role is to manage your
contacts or proxy identities. The database is open since the bot's launch
on server.
NOTE: All <name> attributes ARE unique within each table!

- -dropdb
    - Drops the tables.
```

- -c
    - Creates the tables.

- -ic {<name>} {<uri>}
    - Inserts new contact.

    - example: -ic JohnDoe sip:john@domain

- -ir {<name>} {<uri>} {<password>}
    - Inserts new proxy identity used for registering.

    - example: -ir MyProxy sip:myname@domain mypassword

- -uc {<name>} {<uri>}
    - Updates contact. Must already exist.

    - example: -uc JohnDoe sip:NewUri@domain

- -ur {<name>} {<uri>} {<password>}
    - Updates identity. Must already exist. If you wish to not update
      an attribute, type "-" instead.

    - example: -ur MyNewId sip:NewUri@domain newPassw
    - If you want to update just password: -ur MyNewId - newPassw

- -rc {<name>} or -rr {<name>}
    - Removes a record from contacts (-rc) or from proxy identity (-rr)
      respectively.

Also the user is able to browse its contacts/identities:

- -con [<pattern>]
    - Browse contacts. The pattern is standard sqlite regex with %
      and _ wildcards.

    - example: -con joh% will search for records starting with joh
      and ending with any number of random characters.

- -reg [<pattern>]
    - Browse proxy identities. Same rules for patter as in -con command.

- next [<number>]
    - While in browsing mode you can turn to next page by typing this
      command. You can skip a certain number of pages by providing
      a <number>. By default it is 1.

- prev [<number>]

```
    - Turns to previous page. <number> argument has the same rules
        as in "next" command.

- exit
    - Exits the browsing mode.

NOTE: No other commands than next, prev and exit are available.
You must leave the browsing mode first. If in an active call the user
is still able to keep calling.
```