VOICE-OVER-ALLNET:

A PROJECT REPORT SUBMITTED TO THE
GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

DEC 2014

By

Andreas Brauchli

project report Committee:

Edoardo Biagioni, Chairperson
Copyright 2014 by Andreas Brauchli

You are free to share (copy and redistribute the material in any medium or format), adapt (remix, transform, and build upon the material) for any purpose, even commercially. The licensor cannot revoke these freedoms as long as you follow the license terms:

*Attribution* — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

*ShareAlike* — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

*No additional restrictions* — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

<https://creativecommons.org/licenses/by-sa/4.0/>
To my wife Nicole,

Thank you for your endless love and support

and my unborn son,

May you have a joyful and blessed life full of surprises, may you discover life’s joy in learning and helping others. This irish blessing is for you:

Dearest Father in Heaven,

Bless this child and bless this day
Of new beginnings.

Smile upon this child
And surround this child, Lord,
With the soft mantle of your love.

Teach this child to follow in your footsteps,
And to live life in the ways of
Love, faith, hope and charity.
ACKNOWLEDGMENTS

I would like to express my thanks to Dr. Depeng Li and Dr. Edoardo “Edo” Biagioni for their valuable input and strong support throughout my research.

Additionally, this project would not have been accomplished without the countless hours invested primarily by Dr. Biagioni into the development and stabilization of AllNet, which constitutes the strong foundation this work bases upon.
ABSTRACT

*AllNet* is a decentralized, delay tolerant network that supports custom application extensions. AllNet comes with a text-chat application, but an audio chat feature has, so far, been missing.

This paper discusses the design and implementation of the secure audio-streaming application *Voice over AllNet* created as part of this project. This report details the technologies employed and the design choices leading to the use of the GStreamer framework and the Opus audio codec. We also evaluate the system implementation’s performance trade-offs with respect to network aspects such as packet loss, jitter, and audio quality versus bandwidth, discuss security and privacy aspects of the implementation and, finally, show that the implementation works well in real-world and simulated tests under stress conditions.
# TABLE OF CONTENTS

Acknowledgments ................................................................. iv  
Abstract .............................................................. v  
List of Tables ................................................................. viii  
List of Figures ................................................................. ix  
1 Introduction ........................................................................ 1  
2 Previous Work ................................................................. 3  
   2.1 Ground Work ............................................................. 3  
   2.2 Related Work ............................................................ 4  
3 Voice over Allnet ............................................................... 5  
   3.1 System Requirements .................................................. 5  
      3.1.1 Use cases .......................................................... 5  
      3.1.2 Packet loss, Corruption and Jitter Tolerance .............. 6  
      3.1.3 Codec Choice ..................................................... 7  
      3.1.4 Security and Anonymity ....................................... 10  
   3.2 System Implementation ................................................ 13  
      3.2.1 GStreamer Framework .......................................... 13  
      3.2.2 AllNet Integration .............................................. 14  
      3.2.3 Handshake procedure ......................................... 16  
      3.2.4 GStreamer Pipeline ............................................ 16  
4 Results ................................................................. 18  
   4.1 Testing Method .......................................................... 18
4.2 Empirical Analysis ................................................. 18
  4.2.1 Simulated Packet Drop Results .......................... 19
  4.2.2 Internet Test ................................................. 19

5 Future Work ......................................................... 21
  5.1 Message Caching ............................................. 21
  5.2 Conferencing .................................................. 21
  5.3 Dynamic Rate Adaptation ................................... 22
  5.4 User friendliness and Seamless Two-way Communication 22

6 Conclusion .......................................................... 23

A Test Setup .......................................................... 24
  A.1 Creating a Sample Recording ............................... 24
  A.2 Using VoA to Stream Prerecorded Audio .................. 24
  A.3 Using VoA to Stream Test Tone .............................. 25
  A.4 Setting up a Bidirectional VoA Call ....................... 25
  A.5 Measuring AllNet Latency ................................... 25

Bibliography .......................................................... 26
LIST OF TABLES

3.1  Content of a Stream Encrypted Packet................................. 11
3.2  AllNet Header.................................................................... 15
3.3  AllNet Handshake Payload................................................... 16
LIST OF FIGURES

3.1 Codec Bit-rate and Latency Comparison ............................................. 8
3.2 Codec Quality vs. Bit-rate Comparison .................................................. 9
3.3 AllNet process diagram ........................................................................ 15
3.4 VoA GStreamer Encoder Pipeline ........................................................... 17
3.5 VoA GStreamer Decoder Pipeline ........................................................... 17
I grew up with the understanding that the world I lived in was one where people enjoyed a sort of freedom to communicate with each other in privacy, without it being monitored, without it being measured or analyzed or sort of judged by these shadowy figures or systems, any time they mention anything that travels across public lines.”[1] With this citation and recent news showing that most unencrypted communications are not trustworthy enough any more, the demand for secure communication from all kinds of people has never been higher[2]. Encrypted computer-based communication can take many aspects such as textual, visual, audio or even a combination of them.

Text based communications have been a useful and proven commodity since the dawn of networking. They have since been complemented by other technologies that have gained further attraction as compression and quality of transmission have improved. Audio chats, for instance, provide a more interactive and personal experience than text alone while not requiring nearly as much bandwidth as video chats. All those forms of communication must be transported by a protocol and network. It should not surprise that end users can normally not distinguish between encrypted and unencrypted transmissions. Instead, they mostly put their trust in a program or protocol’s history and reputation.

AllNet[3] is a decentralized network that is intended to provide fully encrypted communication between any participating peers. A message1 may travel through different nodes in the system before reaching its destination but only the intended recipient will be able to decrypt it. Some information required for routing is not encrypted, but is also not strictly necessary to provide2. With its freely available and liberally licensed source code, a key aspect of AllNet is its openness and extensibility. A particular design goal of AllNet is to donate a very small portion (around 1%) of the node’s resources to forward messages

---

1In an abstract sense that includes text as well as other binary data.
2When a message is sent without recipient it is broadcast a number of times before it either reaches its destination or is discarded by the network.
from unknown peers. This ensures that messages can find their way to the destination and promotes sparing use of bandwidth.

AllNet already provides a textual chat application but, so far, no audio streaming. This paper discusses the Voice over AllNet (VoA) implementation with its benefits and challenges for AllNet audio streaming.

The paper is structured as follows: After this introduction we briefly survey previous work in related domains and ground work that is being used as part of this project. We then introduce the requirements for VoA and describe the implementation before discussing the results obtained from simulations and live tests. Lastly, we give some ideas for future research work and implementations, and draw conclusions.
This chapter surveys previous work that we are basing upon. We differentiate between ground work that is being used underneath the VoA implementation and related work in the general domain of security and privacy aware audio streaming solutions.

2.1 Ground Work

This section aims to list previous work that is being used at the base of our VoA implementation.

AllNet is at the very heart of this work: It provides the library functions for interfacing with it and calls for both public key cryptography and symmetric encryption. Public key cryptography is used to initiate the connection by encrypting and signing the handshake messages, and symmetric cryptography is then used to encrypt the actual audio stream since it consumes less processing and network resources than public key cryptography. We extend AllNet by adding the VoA functionality. While VoA does not require changes in the existing protocol, its implementation has indirectly lead to changes beneficial to the streaming use case. One such example is the ability to set a do not cache flag on messages\(^1\).

We also used previous research to guide the decision as to which audio codec to use by default to compress and encode the stream:

In [4], the authors show an experimental evaluation of the Opus codec used in a Voice over IP (VoIP) context. We use their insight in determining Opus as the default codec used in this implementation, along with sensible default settings for the voice encoding in a live streaming environment very close to VoIP.

In two comparative studies at Google[5, 6], Opus was compared to a number of other codecs and was a consistent winner in the categories tested: At 32kbps output the testers were not able to discern the encoded audio from the original, while at 20kbps, Opus by

\(^1\)https://sf.net/p/allnet/code/ci/fcb15b47fc7d8f4f04f74ea634f230206eda34201
far outperformed Speex at 24kbps. These results show that Opus is a qualitatively good encoding codec for speech in both English and Mandarin. Further testing with similar results was performed in [7].

2.2 Related Work

While, to the best of our knowledge, no one else is working on audio streaming over AllNet, there is a multitude of both open and proprietary audio streaming solutions available.

[8] concludes that full-duplex live voice transfer over the low-bandwidth 802.15.4 (ZigBee) network is feasible. The study uses the Speex codec and lists Opus as candidate for a Variable Bit Rate (VBR) codec and improved audio quality in their future work section. While the research is based on ZigBee, we infer that Opus is a likely candidate for general low bandwidth transmission.

Jitsi is an open and encrypted VoIP-like application that also uses, amongst others, the Opus codec for audio data. It supports both video and audio data and other features such as desktop sharing. The encrypted end-to-end stream is set up using one of the supported open and proprietary centralized chat systems. Jitsi is thus meant to protect the content of the communication but can not hide with whom one is communicating in the way that AllNet can with its decentralized architecture and anonymity features.

2\text{https://jitsi.org/}
CHAPTER 3
VOICE OVER ALLNET

This chapter provides the bulk of the paper. It starts by discussing the system requirements and use cases. It continues with the reasoning behind Opus as the chosen codec for the audio stream compression and discusses the impact of relevant network aspects. Finally a security and anonymity analysis is performed to reason that the system implementation is secure.

3.1 System Requirements

The current AllNet streaming implementation is based on the findings listed in this chapter that a low-bandwidth packet loss and jitter tolerant codec is needed to accommodate the typical live streaming use case.

3.1.1 Use cases

The use cases driving the system requirements are described between two users Alice and Bob:

- Alice wishes to speak to Bob using a secure means of communication. It is more important to Alice that the spoken content remain safe than it is to have good audio quality.

- Alice wishes to speak anonymously to Bob whom she does not know, but whose public key she possesses. She already uses AllNet to communicate with him.

- Alice wishes to transmit an audio message to Bob. She does not know whether Bob is currently online.

The currently implementation fully supports the first two use cases. For the first use case, Alice must however first acquire Bob’s public key. This can be done using AllNet’s xchats application and a secure and authentic side channel to share a common secret string.
The third use case only works when Bob is currently reachable through AllNet. If the contact request is not replied to, the audio stream is not established and the message not transmitted. Future work is expected to cover this use case.

### 3.1.2 Packet loss, Corruption and Jitter Tolerance

All internet-based communications are subject to packet loss, corruption and out of order arrival, however, when the Transmission Control Protocol (TCP) protocol is used, these effects are hidden by the protocol’s automatic retransmission. As AllNet, however, implements and uses its own network protocol\(^1\) it can become susceptible to those cases. Additionally, TCP is generally not desired in streaming applications, as a late packet\(^2\) is not relevant anymore and should thus be discarded and not be retransmitted.

Packet jitter, or out-of-order arrival, is especially common when the packets of the stream take different routes to the destination. This is a common case in multi-hop AllNet transmissions, even more so when partial addresses (see 3.1.4) are used. Packet jitter is tolerable when a jitter buffer is used. This jitter buffer queues and, if needed, re-orders incoming packets up to a given size, length or time. Out of order packets that arrive later than the set buffer window are then discarded. The buffer thus increases the latency by at most the time length of the buffer at the benefit of potentially not losing, or losing fewer, packets.

Packet corruption depends mainly on the (weakest) link connection. In the context of audio streaming, it can be dealt with in the following ways:

- The packet can be discarded if it has been identified as corrupt \(e.g.\) by a Hash-based message authentication code (HMAC). It may make sense to discard an encrypted packet entirely if it has been identified as corrupt before decryption\(^3\). A discarded packet will then result in one missing frame, \(i.e.\) 20ms of audio with our default settings.

---

\(^1\)AllNet sometimes uses TCP connections but an AllNet application has no control over that.
\(^2\)Arriving later than the jitter buffer’s window, if used.
\(^3\)A corrupt bit in the cipher is spread across most bits in the plain-text resulting in completely garbled output, not just where the corruption occurred.
• The packet can be processed as-is in the hopes that it is still somewhat playable.

The problem with this solution is that, while it works well in unencrypted scenarios, a slightly corrupted encrypted packet does not decrypt at all and thus must be discarded.

Since we encrypt our audio stream, we are only left with the first option of discarding corrupt packets. The effect, though, may result in less than one lost frame if the codec is able to use Forward Error Correction (FEC) to reconstruct the missed frame. FEC is discussed next in 3.1.3.

3.1.3 Codec Choice

Our choice of the Opus codec depends on a variety of factors: We considered the license, the general availability, configurability of options such as sampling rate, bandwidth, processing power requirements, and ease of integration. While we are choosing only one codec for the reference implementation, we do not exclude others from being used at a later time.

Most important for an open project such as AllNet, we need the codec to be freely available and, if possible, not be patent encumbered in a way that may discourage users from using VoA. A list of candidates\(^4\) that were more closely considered includes Vorbis, Speex, and Opus.

Since our main usage target is real time transmission, we need a codec that provides a low processing latency option. This paragraph focuses on processing latency which only accounts for the latency between when the packet has been received and the audio is played back on that same system. Additional latency is also introduced by transmitting the packet over the network\(^5\). This aspect is however not directly controllable. The processing latency generally stems from using larger buffers, which, in turn, result in lower processing and transmission overhead. Lower processing latency thus does come at a trade off. A comparison of bit-rate and latency is shown in Figure 3.1 and clearly shows Opus as superior contender in that it sports the lowest latency and scales over a wide band of bit-rates. The


\(^5\)[10] provides a very good overview of latency in real-world internet communication. It also states that typically only 5% of the overall latency is caused by packet routing/switching.
next paragraph discusses the processing requirements that must be fulfilled in order for the latency to be met.

Opus is not computationally heavy on the CPU but it is also not trivial. [11] mentions that one should “not expect Opus to run on really slow devices like 8-bit micro-controllers” it does however hint that slower CPUs and Digital Signal Processors (DSPs) might be capable enough, depending on the complexity setting that Opus provides. This codec setting trades complexity for quality.\footnote{While only marginally scientific, this \url{http://beta.slashdot.org/comments.pl?sid=3110273&cid=41306433} comment by a programmer on the announcement of the IETF standardization of Opus reads “Compared to speex, [i]t’s far better coded, uses far fewer CPU cycles, and sounds vastly better”}

Vorbis’ best delay is just shy of 200ms which is rather high for our real time streaming use case. According to [12], the measurable call quality degrades rapidly where the latency exceeds 200ms. VoIP systems generally target 150ms as maximal latency. Speex, like Opus, provides a very good latency of between 20 and 40ms. Opus provides a configurable latency and ranges best at between six to 20ms delay. Opus and Speex were both specifically

Figure 3.1: Codec Bit-rate and Latency Comparison\cite{9}
designed for low latency, while Opus was also designed for high fidelity[13].

The natural choice codec that fits the liberal licensing model, low latency communication and high configurability is Opus. The VoA reference implementation thus uses Opus in its lowest bandwidth (Narrowband)\(^7\) and lowest latency setting. Speex, as liberally licensed alternative candidate, was not chosen due to the lower CPU efficiency, quality and scalability when compared to Opus (See Figure 3.2).

The main aspect in the fidelity of the audio is the sampling rate. Voice is still understandable with a rather low sampling rate of 8-16kHz but this is too low for music to be even remotely enjoyable. Opus provides options to scale the sampling rate from the narrowband 8kHz to the wideband 44.1kHz and even up to the fullband 48kHz. Using narrowband encoding, Opus only requires 6kbps while in fullband, it requires just over 32kbps. Allowable frame durations range from 2.5-20ms\(^8\). As added benefit, Opus supports switch-

\(^7\)Which is also the lowest quality setting, but has been deemed perfectly acceptable in real-world tests.
\(^8\)A frame of 2.5ms at 6kbps weights ca. 2 bytes while a frame of 20ms weights, at the same bit-rate, ca. 15 bytes.
ing modes on the fly, such that the quality could dynamically be adapted to the available bandwidth[13].

Opus supports encoding in both Constant Bit Rate (CBR) and VBR mode. VBR comes at a slightly higher processing requirement but has the benefit of being able to use more bandwidth for more complex parts of the stream and less for less complex parts while keeping the same average bit rate. This results in a higher quality at the same average bit rate when compared to CBR. Since we are already defaulting to the lowest sampling rate, we have thus chosen to take advantage of the quality benefits of VBR encoding.

FEC is another feature that already comes included with Opus. It can be enabled to correct or reconstruct a missing previous packet. The cost is a slight increase in delay of one frame length in the case of a missing packet and the reduction of the available bandwidth for audio since the total frame size does not change when enabled9. Additionally, FEC can only be enabled when the frames are longer than 10ms. Since we are using 20ms frames, we decided to enable FEC.

3.1.4 Security and Anonymity

This section focuses on security and anonymity aspects of audio streaming. Given AllNet’s overall goal of providing secure communication, it would not be appropriate to use unencrypted audio-streaming. AllNet has support for anonymous delivery of messages. We show that we can also use some of those features such as partial addresses for VoA.

Security

There are multiple threats to security amongst which eavesdropping, Denial of Service (DoS) and packet injections are discussed here.

DoS is generally a problem when a communication runs through a hostile network that can simply cap the connection. The only solution to this is, obviously, routing around it. AllNet will always forward packets if there is a way to do so. The other DoS, or

Distributed DoS, case of being flooded with hostile requests can be weakened by minimizing the processing of a packet before it is discarded, \textit{i.e.} until the packet is detected as invalid or malicious. Minimizing processing is, however, not a magical solution. It simply means that the attacker needs to invest more bandwidth to overpower the discarding process.

We note at this point, that AllNet can use its own protocol on top of the IP protocol. Non-TCP protocols are generally less susceptible to DoS attacks because when they don’t implement congestion control, and therefore don’t slow down the sending rate when encountering congestion.

Currently, VoA packets are marked with the \textit{do not cache} flag which slightly elevates the forwarding priority of the packets inside AllNet’s queue-priority algorithm. When the flag is set, and the packet does not arrive at its destination in time, it is discarded instead of being cached for later delivery. If the unreachable destination is a more permanent condition, \textit{e.g.} due to a malicious link, then not setting the \textit{do not cache} flag is a possibility for the packets to reach the destination at a time when the link becomes available. This currently only works if the client keeps its receiver program open until the packets arrive due to the stream cipher state that must be kept for successful decoding. This could be worked around by saving the stream cipher’s state when exiting the program.

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Counter 2 bytes</th>
<th>HMAC 6 bytes. Covers Buffer and counter bytes</th>
</tr>
</thead>
</table>

Table 3.1: Content of a stream encrypted packet, without the preceding unencrypted AllNet header.

For encryption, VoA uses asymmetric cryptography (RSA\cite{14}) for setting up the stream but then switches to symmetric (\textit{Advanced Encryption Standard} (AES)\cite{15}) encryption to encode the actual audio stream. During the handshake, the initiating party sends the key to decrypt the first stream packet and secret to verify the packet’s HMAC. Each audio packet then contains a counter value (See Table 3.1). The counter is used to keep track of lost packets for the decryption process: when a packet does not decrypt, the next counter value can be tried. If this succeeds, one packet was skipped. This works as long as the counter
doesn’t wrap around more than once. Currently a two byte counter is used, allowing up to
\(2^{2^8} = 65536\) skipped packets. This is the equivalent of 21.8 minutes worth of audio with
the used 20ms frames.

**Anonymity**

AllNet supports sending messages without or with only a partial sender and destination
address. A partial address means that only the number of specified bits of the total address
length is considered. When a partial destination is used, a packet should reach all users
that share the common address prefix, effectively creating a broadcasting scenario. Each
recipient then decides if the message is meant for them based on whether they can decrypt
it.

This same principle applies to VoA streams. In addition to the address, however, the
AllNet header of the packet also lists a *stream ID* (unencrypted) that identifies the stream.
Although the stream ID is likely to be unique in the network at any given time, it not
crucial that it be unique nor is it enforced. Rather, it provides an quick way for routing
nodes to decide whether to forward all or no packets of a stream, and for the potential
recipients to quickly identify the relevance of the packet without having to attempt a full
decryption. The stream ID is sent encrypted to the recipient in the initial digitally signed
handshake. Although encrypted with the public key of the sender, it is not a requirement
that the stream ID be sent encrypted as an eavesdropper is not able to tell which out of all
the recipients accepts the message. It is also conceivable – but not currently implemented
– that the stream is delivered without stream ID at all. The downside of having the stream
ID is that multiple packets can easily be put in direct relationship, possibly making traffic
analysis easier. Despite this fact, the advantage of a more reliable routing and thus delivery
generally makes it worthwhile to include the stream ID.
3.2 System Implementation

This section describes the steps performed to implement the current VoA system: First, we looked into recording voice on Linux systems, second, finding an appropriate codec and implement the audio processing pipeline to use it, third sending the data over the network and last, encrypting the data.

3.2.1 GStreamer Framework

We have chosen the GStreamer framework\(^{10}\) to record and encode audio. We wanted a solution that would be as portable as possible focusing on Linux and Unix systems, if possible, a solution that is flexible enough to accommodate different codecs for future expansion. Crucially, it needs to provide access to the system microphone.

Given the multitude of audio servers available on Linux, we have considered ALSA, the low-level access to hardware drivers directly provided by the kernel\(^{11}\) and GStreamer.

Due to its driver-level nature, ALSA is limited in its feature set. It does provide microphone access, but only provides raw audio data. With raw audio, we would then have to encode it ourselves by passing the data to the codec’s library. ALSA is supported on most Unix-type systems, not on OS X or non Unix system however.

GStreamer, in contrast runs on all common Unix-type systems including OS X, and, as bonus, also runs on windows and the common mobile platforms. GStreamer provides a full audio pipeline with a multitude of plugins and codecs. While not as lightweight as ALSA with its dependency on the GObject type system, its feature set and ease of integration outweigh its larger footprint. We thus chose GStreamer to record and convert audio. For future use, GStreamer will also allow easier integration when multiple microphones are available or the audio codec is changed. Already implemented is non-microphone input support from other audio sources such as files and streams supported by GStreamer. This allows recorded messages to be streamed and was used during testing (See chapter 4.)

\(^{10}\)http://gstreamer.freedesktop.org/
\(^{11}\)Some software audio servers such as pulseaudio (GNOME), phonon (KDE), or jack expose a subset of ALSA as interface.
GStreamer comes with the command line tool `gst-launch-1.0` that allows easy setup of a string-defined pipeline. *e.g.* `gst-launch-1.0 audiotestsrc ! audioconvert ! audioreampse ! opusenc ! udpsink host=localhost port=5000` sends a test tone (`audiotestsrc`) to the opus encoder (`opusenc`) which forwards it on the network (`udpsink` to `localhost` on port 5000.) We started the VoA implementation by replicating this pipeline in code. Instead of using `udpsink` we re-transmitted the raw encoded audio buffers over a User Datagram Protocol (UDP) socket. The socket implementation was then tested by receiving the transmitted data with `gst-launch-1.0`. Once this worked, we implemented the reverse solution: Capturing the incoming data on a socket and piping it into the GStreamer pipeline. When this step also worked in conjunction with the `gst-launch-1.0` tool, we then sent data from the socket-based implementation to another instance of itself.

Naturally, some challenges were encountered during the implementation. A short excerpt is presented in this paragraph: Wireshark\textsuperscript{12}, which was used to analyze the transmitted packets, proved to be very helpful in debugging\textsuperscript{13}. Another problem was that we noticed that the `gst-launch-1.0` solution was able to pick up stream at any position and play it back, also after missing packets and a restart of the sender application. Our implementation however would not continue playing after an interruption or a lost packet. It turned out that we needed to disable the pipeline’s `sync` flag, which, when enabled, pauses playback while waiting for the missing packets. Luckily, pure experimentation instead of good documentation of supported GStreamer element flags resolved this issue.

Once the socket based prototype worked, the code was refactored to send the buffers using AllNet instead of sockets.

### 3.2.2 AllNet Integration

This part describes how the functional socket based implementation, described previously, was refactored to use AllNet instead of sockets to transmit the audio stream.

\textsuperscript{12}https://www.wireshark.org/
\textsuperscript{13}e.g. a highly compressed audio stream should look rather random, yet some longer byte-ranges were filled with zeroes. This clue revealed and ultimately lead to a solution.
AllNet clients communicate on local sockets with \texttt{alocal}, the AllNet process which distributes incoming and outgoing messages to each client. The communication, however, is hidden to the developer by convenient library functions provided by AllNet. As all local traffic is also broadcast to all clients, we can locally test the AllNet integration by simultaneously connecting both the VoA server and the VoA client to the same AllNet daemon\textsuperscript{14}.

Table 3.2: AllNet header with following payload.

Packets, consisting of an AllNet header and a payload (See Table 3.2), are encrypted and signed using AllNet’s library function for encryption and signing. The handshake is encrypted and signed using the public key infrastructure and the stream is encrypted and signed using the recently added AllNet stream cipher calls.

\textsuperscript{14}This requires setting up two local contacts which have each others public key.
3.2.3  Handshake procedure

To initiate a stream, a two-sided handshake procedure consisting of the SYN and ACK phases is first performed. In the initial SYN phase, an encrypted request is sent to the recipient. The request contains the stream ID, encryption and verification keys, and a list of possible codecs to choose from (See Table 3.3). When no reply is received within 2 seconds, another request is sent for at most 10 times. When the recipient gets a request he is interested in he replies with an ACK packet, specifying which stream he is accepting and what media type he chooses.

<table>
<thead>
<tr>
<th>AllNet App Media Header</th>
<th>Identifies VoA packet</th>
<th>Used to indicate handshake SYN or ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>app_id (4B)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>media_id (4B)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SYN header</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>stream_id (4B)</strong></td>
<td>Stream ID</td>
<td></td>
</tr>
<tr>
<td><strong>enc_key (16B)</strong></td>
<td>Stream cipher key</td>
<td></td>
</tr>
<tr>
<td><strong>enc_secret (64B)</strong></td>
<td>HMAC secret</td>
<td></td>
</tr>
<tr>
<td><strong>num_media_types (2B)</strong></td>
<td>Number of media types</td>
<td></td>
</tr>
<tr>
<td><strong>media_type[] (4B ea)</strong></td>
<td>In order of preference</td>
<td></td>
</tr>
<tr>
<td><strong>ACK header</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>stream_id (4B)</strong></td>
<td>Stream ID from SYN</td>
<td></td>
</tr>
<tr>
<td><strong>media_type (4B)</strong></td>
<td>Selected media type</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: AllNet handshake payload for both SYN and ACK cases.

3.2.4  GStreamer Pipeline

In subsection 3.2.1, the details of the underlying pipeline are shown. We now explain how that pipeline is modified for sending the encoded audio buffers over AllNet.

On the encoder side, the pipeline’s `udpsink` element is replaced with an `appsink`, an adaptor that allows custom processing of the audio buffer. Our custom processing then encapsulates each audio buffer into an AllNet packet, encrypts it using the stream encryption framework provided by AllNet, and sends it to the given destination. The encoder pipeline

---

15 Only Opus audio is currently supported but for extensibility, we are using a list.
16 VoA provides an option to immediately start streaming after sending out the SYN handshake. This option can be used in non-interactive cases or when broadcasting to multiple recipients. Currently the SYN packet is, however, only sent once. Stream decryption is obviously only possible after having received the keys in this packet.
17 He may choose to only wait for requests from a specified contact, or, if unspecified, from any party in his contact list.
is visualized in Figure 3.4.

![VoA GStreamer encoder pipeline](image)

**Figure 3.4: VoA GStreamer encoder pipeline**

The decoder pipeline, as shown in Figure 3.5, has been modified analogously to the encoder pipeline: The `udpsource` element is replaced by an `appsrc`, which, after making sure that the packet is an audio-packet for the given stream, decrypts it and injects it into the pipeline for further processing.

Both custom encoder (appsink) and decoder (`appsrc`) elements are agnostic to the audio codec used. Changing the codec does thus not affect the AllNet encapsulation.
CHAPTER 4
RESULTS

This section discusses the testing and results that the VoA implementation underwent. We show that while AllNet latencies are generally higher than for direct streaming, the resulting latency is still acceptable.

4.1 Testing Method

For better reproducibility in testing we use a recorded voice sample instead of live audio. This allows us to repeat all tests under the same condition. The spoken voice sample is the 10 seconds long reading of “The quick brown fox jumps over the lazy dog’s back and runs into the forest, never to be seen again.” We tested VoA in the following scenarios:

1. Streaming a 440Hz sine wave test tone in a lossy environment.
2. Streaming the pre-recorded test sample in a lossy environment.
3. Streaming the 440Hz sine wave test tone from one host to the other over two separate but neighboring home 20/5Mbit DSL internet connections\(^1\).
4. Streaming the pre-recorded test sample over the same connection as in 3.
5. A bi-directional\(^2\) live streaming test over the same connection as in 3 and 4.

4.2 Empirical Analysis

We will now discuss the results of the empirical VoA test runs. First in a simulated lossy environment, followed by a real-world test. The results show that even with very high packet

\(^1\)The direct round-trip time between the connections (established with ping) is 12ms. This value is however only marginally relevant since AllNet is free to choose its path to the destination.

\(^2\)VoA currently only indirectly supports this mode by manually opening one connection in each direction.
loss in the simulation, the message is still understandable\textsuperscript{3} while real-world test shows that while the latency due to the many hops is not ideal, it is still manageable.

\subsection{Simulated Packet Drop Results}

The simulated packet drop test qualitatively establishes at what packet loss rate a received audio stream is not understandable anymore. Packet loss simulation is done locally by skipping packets in the sending process when a random number modulo 100 is above the set threshold. Thus the dropped packet rate is only reached approximately but the result is more “natural” than dropping every \( n \)-th packet.

In the first test, the sine wave is choppy but audible until about 60-70\% packet loss. With a higher loss ratio, it is not clear anymore that a sine wave is being played. Although very subtle, the chopping is already audible at 1\% loss.

With the spoken test sample, the content is still understandable up to \textit{ca.} 45\% packet loss. At 50\% loss it is understandable in around half the cases. At 70\% only parts of the sentence can be made out and at 80\% one only hears a chopped voice but no words.

\subsection{Internet Test}

After the simulated test, we repeated the test over the internet without the packet loss simulation. We took the \texttt{trace}\textsuperscript{4} results to count the number of hops and round-trip time for a trace packet to the destination.

In our first test, the indicated round-trip time from the sender to the recipient was between 250 and 370ms, averaging at \textit{ca.} 300ms, over three hops. The voice was clearly understandable with an occasional audible crackling every \textit{ca.} 10 seconds indicating lost or late packets.

The second test, using the sine-wave test tone, shares the results with the voice test: Clear audio with the occasional slight crackling.

\textsuperscript{3}The results may be biased since the test person knew the spoken text before hearing it.

\textsuperscript{4}\texttt{trace} is the AllNet equivalent of \texttt{traceroute}. It shows the number of hops to the destination and the round trip time to each node on the path.
In a third test, we established a simultaneous bidirectional connection between both test nodes. Again, both streams were forwarded by two intermediate nodes with a total delay averaging at \textit{ca.} 300ms. That delay does slightly affect the conversation during a call in that persons will simultaneously start to speak, and when the counterpart is heard, both will simultaneously pause to listen. The quality, as expected, is the same as with a unidirectional stream and the voice is well understandable.
CHAPTER 5
FUTURE WORK

This chapter is dedicated to future research and implementation of VoA. So far, we have created a working proof of concept that streaming live audio over AllNet is indeed feasible. This opens up many new possibilities to explore, some might even be unique to AllNet due to its decentralized nature with limited storage on other nodes.

5.1 Message Caching

A central part of AllNet is the ability to cache messages when a recipient is not currently available or reachable. An interesting research area is to find a way for audio messages to be cached within the network. Simply holding back each individual packet at one node might of course work when they are appropriately marked for caching and retransmitted in the correct order. Better even, if the sender could send the whole message in one (possibly split) package in that case. This would ensure that the message arrives wholly instead of in possibly unplayable fragments. This may also involve dynamically coalescing the encrypted audio packets into larger ones at one of the intermediate hops. Also, when a sender knows that the streaming will not be live, it might also make sense to pack multiple audio frames into a single AllNet packet.

5.2 Conferencing

Given AllNet’s existing support for broadcasting (see 3.1.4), adding an audio broadcast mode comes down to modifying the handshake procedure (See 3.2.3) to send the encryption keys to all participants. For the receiving of audio from multiple sources the recipient should create one audio pipeline for each sender. If this is not done, the GStreamer audio pipeline still accepts all valid packets from the $n$ senders but will introduce a lag since the audio buffer will grow $n - 1$ times faster than the playback at regular speed.
5.3 Dynamic Rate Adaptation

Opus supports dynamically changing modes. This could be used to maximize the audio quality such that a higher fidelity encoding is used when the bandwidth is available and scaled down accordingly during contention.

5.4 User friendliness and Seamless Two-way Communication

While the current implementation can be used to set up a two-way audio stream by both parties, it would be easier to have an option that would do this for the user. A user-friendly implementation would add the functionality to AllNet’s existing graphical xchat client. This way both text and voice chat can be performed simultaneously through the same interface.
CHAPTER 6
CONCLUSION

With the VoA project we have shown that it is feasible to stream live audio over the AllNet network. We have elaborated on the design decisions of using Opus as the audio/voice compression codec and GStreamer as both the encoding and decoding pipeline for both Opus and alternative future codecs. For Opus, we reasoned about sensible encoding parameters given the expected link parameters. The results show that VoA works in real-world scenarios when both the sender and recipient are reachable through an indirect AllNet link passing through multiple forwarding nodes. Further more, we have proposed future additions and research possibilities for VoA.

We hope that this project will promote individual privacy and security and optimistically look forward to new possibilities in that general area.
This appendix explains how to reproduce the test steps including creating a voice record and exchanging AllNet keys.

A.1 Creating a Sample Recording

This section documents how to create a recording that can be played back using VoA. GStreamer’s `gst-launch-1.0` is used to record the sample. Any other recording tool will also work.

The following command records raw PCM audio from the system’s primary input source – usually a microphone. Press `[CTRL]+[C]` to stop the recording.

```
gst-launch-1.0 autoaudiosrc ! wavpackenc ! filesink location=out.wav
```

The file can then be played back with the following command:

```
gst-launch-1.0 filesrc location=out.wav ! wavpackparse ! wavpackdec ! autoaudiosink
```

This previous command only plays back `.wav` files. VoA uses the `uridecodebin` module which accepts any supported audio type. Thus, compressed formats such as *e.g.* `ogg-vorbis` are also accepted.

A.2 Using VoA to Stream Prerecorded Audio

VoA supports playing back prerecorded audio from an accessible URI. Since any supported sources work, the full URI must be used. Use the following command to play back a recorded audio file on the local harddrive on VoA:

```
voa -s -c contact-name -f file:///full/path/to/file.wav
```
A.3 Using VoA to Stream Test Tone

To stream the 440Hz test tone, we have modified the source code to use GStreamer’s `audiotestsrc` instead of `autoaudiosrc`. The change requires a source recompilation and the following line from `voa.c:925`

```c
data.enc.source = gst_element_factory_make("autoaudiosrc", "source");
```

must be replaced with

```c
data.enc.source = gst_element_factory_make("audiotestsrc", "source");
```

Alternatively, the test tone can also be recorded to a file which is then played back. Use this command to record the test tone:

```
```

The full VoA source code is available on GitHub[16].

A.4 Setting up a Bidirectional VoA Call

To test VoA in the interactive call scenario, invoke the following commands concurrently (e.g. using two terminals) on contact one’s machine:

```
voa -c contact-name-two
```

```
voa -s -c contact-name-two
```

and the same commands on contact two’s machine replacing `contact-name-two` with the respective `contact-name-one`. Both machines must already be set up with the respective contact’s public keys.

A.5 Measuring AllNet Latency

In our results, we have measured the AllNet latency using `trace`. When invoked without arguments, a broadcast trace is performed and the latency to all reachable nodes is shown. To only trace a specific address, use the command

```
trace -i 08.18
```

This will trace all addresses staring with the 32 bits `08.18` and report the round trip time.


