Bachelor thesis

Improving Voice over GNUnet

Fakultät IV - Elektrotechnik und Informatik
Internet Network Architectures (INET)
Research Group of Prof. Anja Feldmann, Ph.D.

Christian Ulrich
July 6, 2017

Prüferin: Prof. Anja Feldmann, Ph.D.
Betreuer/innen: Theresa Enghardt, M.Sc. und Christian Grothoff, Ph.D.

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den July 6, 2017 Christian Ulrich

Zusammenfassung

Im Gegensatz zu den allgegenwärtigen Cloud-basierten Lösungen bietet die Telefonie-Applikation *GNUnet conversation* vollständig dezentrale, sichere Sprachkommunikation und erschwert damit Massenüberwachung. Das Ziel dieser Arbeit ist es zu erforschen, warum GNUnet conversation unter typischen Internet-Bedingungen momentan mangelhafte Quality of Experience aufweist.

Nachdem Netzwerk-Shaping und die Initialisierung zweier isolierter GNUnet-Peers automatisiert worden waren, wurden Latenzmessungen durchgeführt. Mit emulierten Netzwerk-Charakteristiken wurden Netzwerk- Kryptographie- und Audio-Codec-Latenzen gemessen und die übertragene Sprache aufgezeichnet.

Die Analyse der Messergebnisse und eine subjektive Beurteilung von Sprachaufnahmen machten deutlich, dass in den meisten Szenarien extreme Ausreißer auftreten und die QoE beeinträchtigen. Außerdem wurde nachgewiesen, dass GNUnet conversation große Latenzen verursacht, die den Rahmen in dem gute QoE möglich ist einschränken. In der Messumgebung traten immer mindestens 23 ms Latenz auf, von der ein großer Teil durch Kryptographie entstand. Es konnte nachgewisen werden, dass eine Optimierung des Verschlüsselungs-Moduls und anderer Komponenten möglich ist. Schließlich wurden die Voraussetzungen, um gegenwärtig gute QoE zu erreichen, bestimmt und Ideen für weitere Nachforschungen präsentiert.

Abstract

In contrast to ubiquitous cloud-based solutions the telephony application *GNUnet conversation* provides fully-decentralized, secure voice communication and thus impedes mass surveillance. The aim of this thesis is to investigate why GNUnet conversation currently provides poor Quality of Experience under typical wide area network conditions and to propose optimization measures.

After network shaping and the initialization of two isolated GNUnet peers had been automated, delay measurements were done. With emulated network characteristics network delay, cryptography delays and audio codec delays were measured and transmitted speech was recorded.

An analysis of the measurement results and a subjective assessment of the speech recordings revealed that extreme outliers occur in most scenarios and impair QoE. Moreover it was shown that GNUnet conversation introduces a large delay that confines the environment in which good QoE is possible. In the measurement environment at least 23 ms always ocurred of which large parts are were caused by cryptography. It was shown that optimization in the cryptography part and other components are possible. Finally the conditions for currently reaching good QoE were determined and ideas for further investigations were presented.

Contents

1 Introduction

In times of global bulk surveillance[1] voice communication is one of many surveillance targets[18]. This endangers the right to privacy[1] of individuals all over the world. Even politicians are not safe from surveillance of their confidential communication[16][21]. Encrypted voice communication can help protecting individual privacy and professional confidentiality. Unfortunately most of the existing encrypted Voice over IP (VoIP) applications employ a centralized infrastructure. This still allows bulk surveillance of metadata which, compared with the communication contents, in many cases seems to be equally or more suitable for gaining personal information[24]. In fully decentralized systems metadata do not accumulate in a central database and are not controlled by organizations. Thus mass surveillance is effectively impeded.

Skype has shown that voice communication over p2p networks is feasible and can provide good **Quality of Experience (QoE)**. It never employed a fully distributed architecture though[2] and recently moved into the cloud[6]. **GNUnet**[10] is a p2p framework which provides a secure internet stack for distributed applications. GNUnet's protocol stack works as an overlay on top of traditional transport protocols. In contrast to ubiquitous cloud-based solutions GNUnet provides fully decentralized services such as a decentralized public-key infrastructure[23], a decentralized routing protocol and end-to-end encryption by default[22].

GNUnet conversation is an application that provides secure voice communication in a fully decentralized way by employing GNUnet for routing and transport, hence it is a *Voice over GNUnet* application. Making calls is already possible but the voice quality currently tends to be poor in wide-area networks. This bachelor thesis identifies the conditions under which bad QoE occurs and examines possible causes. In a measurement environment of two virtual machines calls between two peers were made under different emulated network conditions and the resulting audio recordings were evaluated. Delay measurements were done. This allowed determining what shares different GNUnet components have in the overall delay and pointing out opportunities for optimization. Furthermore a minimum bandwidth of 200 Kbit/s was found to be required for good QoE.

The delay measurements were restricted to ongoing delays, that is delays that occur during a call. Another type of delay, namely connection establishment delays, that is the time difference from the initiation of a call and the beginning of audio transmission, is left for future work.

Existing GNUnet services and command-line tools were modified and capabilities for measuring different delays were implemented. Fully automated measurements were achieved with an architecture comprising a **measurement controller** and multiple **measurement workers**. This includes initializing the GNUnet peers running on the measurement workers and network shaping.

Chapter 2 gives an introduction to QoE-related metrics and presents related work. GNUnet's architecture and basics of different GNUnet components relevant for the measurements are described in Chapter 3. While Chapter 4 explains the measurement environment including the automated network shaping and topology establishment, and the audio setup, Chapter 5 describes the tools used for the measurements. A detailed description of the experiments and an evaluation of the measurement results can be found in Chapter 6. Finally Chapter 7 describes recommendation for optimizing Voice over GNUnet and Chapter 8 summarizes the findings and presents a conclusion.

[1] for an overview of revelations of surveillance programs since 2013 see `http://projects.propublica.org/nsa-grid/`

2 QoE-related metrics

This chapter gives an overview on how QoE can be assessed and associates the methodology of this thesis with the described methods (Sections 2.2 and 2.3). As QoE cannot be measured directly, **Quality of Service (QoS)** metrics have to be used for the assessment. Section 2.1 gives an introduction to those metrics.

2.1 QoS metrics

Examples of QoS metrics are **delay**, **packet delay variation (PDV)**, **bandwidth** and **packet loss**. This thesis focuses on delay.

As explained in the Chapter 1 the delays examined are ongoing delays after the connection is established (i.e. a call is active). These delays can have the components *propagation delay*, *processing delay* and *queueing delay* (see e.g. [7]). Propagation delay is caused by the limited speed a signal can propagate through fibers or wires. Processing delay may be caused by *"actual packetization, compression, and packet switching"*[7]. For GNUnet conversation processing delays are also caused by encryption and decryption as explained in Section 3. Queueing delay is often caused by congestion. Generally all incoming or outgoing queues, which are used for several reasons (e.g. for jitter compensation as explained in [13]) cause queueing delay.

2.2 Objective QoE assessment

QoE in voice applications can be assessed in different ways. As elaborated in [19], *objective quality assessment* is possible by analyzing properties of the audio signal such as the peak-signal-to-noise-ratio or by using standardized assessment models like the E-model [3] which calculates a quality rating based on measured QoS metrics such as packet loss rate, and delay.

Extensive objective quality assessment as defined in [19] was not done in the scope of this thesis. Instead the QoS metric delay was assessed under different emulated network shaping scenarios (described in Chapter 6). One requirement for satisfying QoE is a sufficiently low mouth-to-ear delay, that is the time difference between the recording at the sender's microphone and the playback at the receiver's speaker. ITU-T Recommendation G.114 states that for mouth-to-ear delays of less than 150 ms *"most applications [...] will experience essentially transparent interactivity"*[13].

2.3 Subjective QoE Assessment

As explained in [19] subjective assessment requires user ratings of speech samples, either absolute or in comparison to a reference speech recording. An example for absolute user ratings is the *Mean Opinion Score (MOS)*. Users are asked to rate a stimuli on the MOS scale, that is from 1 (bad quality) to 5 (excellent quality)[14]. For a reliable assessment *"a balanced set of sufficient number of subjects that represent different level of expertise, age groups and gender."*[19] are required.

User studies with multiple subjects were not done in the scope of this thesis. Instead recordings of speech transmitted over GNUnet conversation under different network conditions were examined for specific anomalies such as silence periods or similar artifacts. This can give hints about the conditions under which good QoE can be achieved.

3 GNUnet Basics

This chapter provides an overview on the general modular architecture of GNUnet and gives an introduction about the GNUnet modules relevant for the measurements described in Section 6.

3.1 Peers and identities

GNUnet peers are addressed using *peer IDs* which are EdDSA[4] public keys which are also used for encryption and authentication. They are base32-encoded and 52 Bytes in length. In additions applications can use *identities* which are defined by an ECDSA[15] key pair. These are needed for signing messages or records of the GNU name system (GNS) which is GNUnet's replacement for DNS.

3.2 GNUnet Architecture overview

The GNUnet modules relevant for this thesis are **TRANSPORT**, which makes use of the **Automatic Transport Selection (ATS)** module, **CORE**, **Confidential Ad-Hoc Decentralized End-to-End Transport (CADET)** and **CONVERSATION** which makes use the **GNU Name System (GNS)**. Each module provides a service API[2] to applications or other modules. This allows building a stacked architecture: Modules can use the APIs to *"build higher GNUnet layers on top of lower ones"*[11]. Figure 1 shows the service stack used by GNUnet conversation. The modules TRANSPORT, CORE, CADET and CONVERSATION are described in more detail in the following sections.

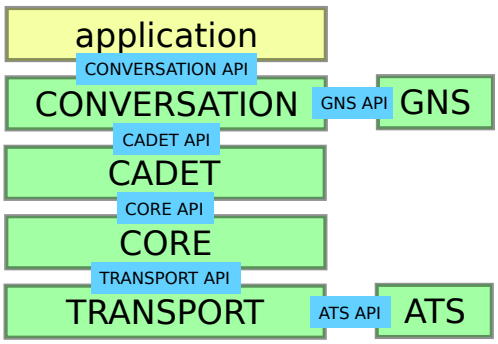


Figure 1: service modules relevant for GNUnet conversation: The application uses only the CONVERSATION API which abstracts from the lower layer modules

GNUnet modules are are composed of a service which runs in a separate executable and a client library implementing the service API. The inter-process communication used between service and client part is expected to cause a small amount of delay.

3.3 The TRANSPORT service module

TRANSPORT is the lowest GNUnet layer and is able to use several protocols as an underlying transport mechanism. While in this bachelor thesis the focus is on TCP and UDP, TRANSPORT may also send packets directly over WLAN or even use HTTP as transport protocol. The TRANSPORT service provides a plugin API to add new underlying transport mechanisms. TRANSPORT's functionality includes choosing the best-suited transport mechanism, receiving and validating information about other peers from the

[2] All GNUnet APIs are documented in `https://gnunet.org/doxygen/modules.html`

network (so-called HELLO messages, which contain the IP addresses of a peer) and establishing connections to known peers. Clients using the TRANSPORT API (the CORE service is usually the only client) are notified about newly established or lost connections.

The TRANSPORT service delegates the decision which transport mechanism to use to a GNUnet module called Automatic Transport Selection (ATS). It informs ATS about available addresses and ATS determines the best-suited transport mechanism by collecting performance characteristics, e.g. by measuring throughput and delay. This means that ATS may decide to switch to another mechanism whenever it detects performance changes. For measurements this is not wanted and thus this behaviour has to be disabled in the TRANSPORT configuration.

3.4 The CORE service module

The CORE service uses the TRANSPORT service trough the TRANSPORT API and adds security properties (e.g. authentication and confidentiality) to the insecure communication provided by TRANSPORT. The cryptographic functions used are based on those used by Off-the-Record messaging (OTR)[3].

3.4.1 OTR cryptography

OTR was designed to be used on top of messaging protocols like XMPP and provides authentication and confidentiality with forward-secrecy. GNUnet uses it already on the CORE layer which is analog to the link layer in the OSI model.

By default the TRANSPORT service establishes communication channels to all peers it knows and provides message queues to the CORE service. The CORE service is then able to performs elliptic curve Diffie-Hellman key exchange (ECDHE) using the Curve25519 curve. Once the session key is exchanged, CORE uses that key to encrypt all outgoing packets with both 256bit AES and TWOFISH.

OTR provides forward-security by periodic re-keyings, that is performing ECDHE after a defined amount of time. GNUnet CORE performs these re-keyings after 12 hours.

3.5 The CADET service module

While the CORE service provides secure link-layer communication between two peers the CADET service provides end-to-end encrypted communication using the *double ratchet algorithm* described in Section 3.5.2. An introduction to CADET and its terminology is given in Section 3.5.1.

3.5.1 CADET: routing, transport, security

CADET finds routes to an other peer using the set of known peers stored in the PEER_STORE and DHT lookups. The DHT used by GNUnet is R^5N[9]. Figure 2 shows a CADET *tunnel* which provides end-to-end encrypted authenticated communication between two peers. A tunnel can be used by different applications, each opening a *channel* to the target application on the peer at the tunnel end-point. Like TCP and UDP flows, channels are identified by a port. If multiple *connections* (which are established by the TRANSPORT layer) exist, data sent through the tunnel is multiplexed over multiple routes so that compromised or malicious peers along a route don't see all packets.

[3] `https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html`

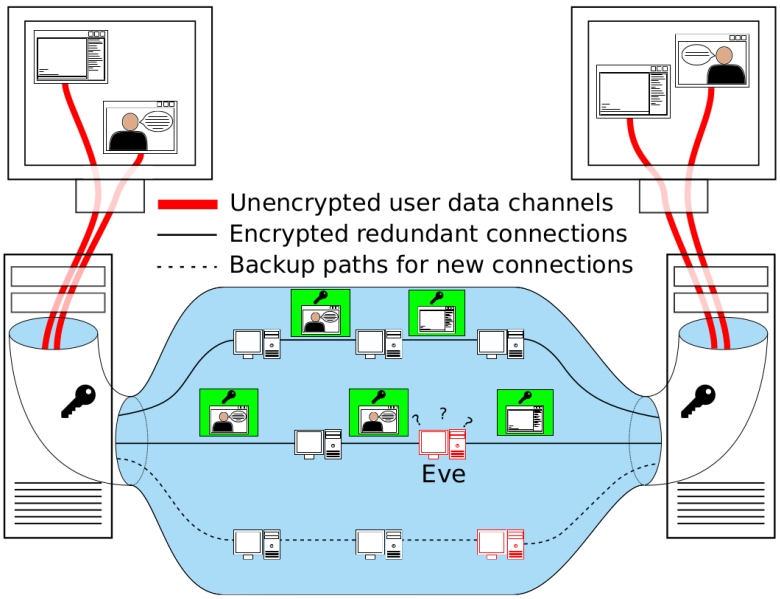


Figure 2: CADET architecture, source: [22]

3.5.2 Double ratchet cryptography

The double ratchet algorithm [17] (previously known as the Axolotl ratchet) is a cryptographic protocol designed to provide end-to-end encryption to asynchronous messaging applications. Among other properties it provides forward security without the need of frequent re-keying (hence it is suitable for asynchronous communication) and break-in recovery.

Forward security is provided by making sure every message is encrypted with a unique message key that can be deleted after encryption or decryption. The message keys are the output of one of two ratchets (the term ratchet describes the process of deriving keys for encrypting and decrypting in parallel using a key derivation function (KDF)). This ratchet is called the symmetric key ratchet. An attacker who learns a secret key used in the KDF at some point cannot decrypt previously intercepted messages, thus the algorithm is forward-secure.

Break-in recovery is provided by combining the symmetric ratchet with a second ratchet, called the Diffie-Hellman ratchet. The key pair used in the symmetric ratchet is replaced regularly with the output of a Diffie-Hellman key exchange. An attacker who learns a key used in the KDF and uses same symmetric ratchet as sender and receiver can only decrypt intercepted messages until the next re-keying (Diffie-Hellman ratchet step).

The symmetric ratchet uses a HMAC-based KDF[4] for key derivation and CADET uses the resulting message keys are used for encrypting each outgoing message with both 256 bit AES and 256 bit TWOFISH, both in the *Cypher feedback (CFB)* mode of operation (defined in [8]). For both AES and TWOFISH the same cryptographic operations are used for both encryption and decrypting and thus in theory both cause the same delay. As the decryption for the CFB mode of operation is parallelizable but encryption is not (see [5]), encryption will probably take longer in practice. The double ratchet algorithm adds another factor that might influence the encryption/decryption time ratio. For an encrypted message multiple decryption attempts and thus increase the decryption time. The reason is that message keys may be skipped so that the receiver has to derive a new

[4] `http://www.rfc-editor.org/rfc/rfc5869.txt`

message key and try decryption again.

Both peers can initiate a re-keying, that is a Diffie-Hellman ratchet step, after the first message has been sent. As soon as a peer learns about the other party's new public key, it also performs the ratchet step. The re-keying period CADET uses is configurable. By default CADET performs a re-keying after 1 hour no matter how many messages were sent. If messages are sent a re-keying is performed after 64 messages have been sent. For key exchange CADET uses ephemeral elliptic curve Diffie-Hellman (ECDHE) with a Curve25519 elliptic curve.

3.6 The CONVERSATION service module

The CONVERSATION service uses the CADET API to create a CADET channel when a call is started. In the original CONVERSATION implementation two CADET channel existed, one for audio data and the other for control packets. In the current implementation only one CADET channel is used for both audio data and control packets. As control traffic has to be sent reliably the channel is created with the GNUNET_CADET_OPTION_RELIABLE option which means that CADET's flow control and congestion control are activated. Reliable transmission of audio data is not needed because due to the low-latency requirement it does not make sense to wait for lost packets to be retransmitted before decoding. In order to eliminate this unnecessary overhead it is planned to allow a per-packet reliability switch in the future. Control packets would still be transmitted reliably while audio packets would not.

3.6.1 Audio encoding/decoding

For audio encoding and decoding in CONVERSATION the libopus implementation of the OPUS codec[26] is used. As it is possible to determine the language spoken in the encrypted conversation when using variable bit rates[28], CONVERSATION uses a fixed bit rate (resulting from fixed a sample rate of 48000 Bit/second and a single channel). Another codec parameter is a maximum size for an encoded payload of 1024 Byte. This does not mean that in calls this packet size is reached. See Section 5.1 for more information about the packet sizes. In order to abstract from the codec implementation, generic libraries exist for audio recording and playback: the MICROPHONE library[5] and the SPEAKER library [6]. When a microphone or a speaker instance is enabled, a helper, that is a separate executable is started. The microphone helper outputs audio obtained from a Pulseaudio source to stdout in the OGG format which is then read and routed to the microphone implementation by the GNUnet HELPER library. The speaker helper reads OGG audio from stdin and sends it to a Pulseaudio sink. This helper architecture is relevant for the measurement method as described in Section 5.1.1.

[5] https://gnunet.org/doxygen/d5/d5c/group__microphone.html
[6] https://gnunet.org/doxygen/d4/d62/group__speaker.html

4 Methodology

This section gives an overview on the measurement methodology. Starting with a description of the virtual machines (Section 4.1) the measurement scripts are then described (Section 4.2. The Sections 4.3 and 4.4 describe how different network properties were emulated and how different network topologies were established. The audio setup is explained in Section 4.5.

4.1 The virtual machines

Overall 20 virtual machines were available. The used virtualization technology was KVM. All machines had a virtual dual-core AMD CPU and 1 GB of RAM. The AES hardware acceleration of the host's CPU was delegated to the guests. The operating system on all guests was Debian 8 (jessie).

4.2 Measurement automation

Below the virtual machines conducting the experiments described in the previous section are referred to as *measurement workers*, the machine initiating the experiments and collecting the measurements is referred to as *measurement controller*. For automating the measurements two Python scripts were written: the measurement worker script (`mw.py`) and the measurement controller script (`mc.py`). Usage information can be found in Appendix B.

On all workers the directory `environment` is present which contains the worker script, GNUnet configuration files (e.g. for using only TCP or only UDP as underlying transport protocols) and other scripts, of which some are called by the worker script. E.g. there are scripts for initializing the virtual machine, reinstalling GNUnet and initializing Pulseaudio.

The experiment specifications used by both controller and workers are written in YAML. The specification files only have to be stored on the controller, the controller script takes care of sending it to the workers. Listing 1 shows an example experiment specification. It contains the topology of the worker peers (in this thesis only directly connected peers have been used), the network shaping parameters and parameters for the measurements, e.g. the count of RTT measurements. *mw4* and *mw5* are *worker IDs* which reference information stored in a separate file (`config.yaml`). It contains information such as a worker's IP addresses and SSH ports.

The procedure of an experiment is as follows:

1. The controller script reads an experiment specification file, extracts information about the workers involved: the worker initiating the measurements and the target worker (e.g. where an echo service for round-trip time measurements is to be run). SSH connections to the obtained workers are established.

2. GNUnet is started on all workers using a remote procedure call to `init-gnunet`. Which configuration file is used depends on which transport protocol was specified when starting the controller script. The call returns information about the started peer: a *GNUnet HELLO message* (see Section 3.3), the *peer ID* and the conversation address (see Section 5.1.1).

3. The obtained information is distributed to all involved peers using a remote procedure call to `set-workers`.

4. The experiment is initialized on all workers (using the `experiment --init` RPC). The worker script does the network shaping (as described in Section 4.3, introduces neighbour peers using HELLO messages (and thus establishes the topology (as described in Section 4.4), initializes the Pulseaudio sources and sinks (see Section 4.5) and starts command-line tools in listening or echo mode.

5. The experiment is conducted (using the `experiment` RPC). On the worker side this includes starting tcpdump, starting audio playback and audio recording (if conversation is involved). Once finished the RPC returns the measurement values and the controller stores them to the results directory.

6. The controller fetches additional files, i.e. tcpdump output and the audio recording.

7. GNUnet is stopped on all workers using the RPC `shutdown-gnunet`.

```
---

worker: mw4
topology: [[mw4, mw5]]
shaping:-

        links: [[mw4, mw5], [mw5, mw4]]
        parameters:
            latency_ms: 80
            bandwidth_kbps: 100000
measurements:-

        type: conversation_delay_ping
        target: mw5
        count: 1000-

        type: conversation_delay_call
        target: mw5
        count: 1000
```

Listing 1: specification of experiment cv_delay_3: **topology:** mw4 (measuring worker), mw5 (target worker), **network shaping:** 80 ms delay, 100 Mbit/s bandwidth, **measurements:** delay measurements using ping method and call method (see Section 5.1.2), 1000 measurement values each

4.3 Network shaping

The effects of multiple network properties which were not inherently present in the environment were to be examined. E.g. the latency between the virtual machines was very low because all VMs were on the same physical host machine. Section 6.1.2 shows that it was below 1 ms. Realistic latencies up to 300ms were to be measured, so they had to be emulated.

For emulating delay, delay variation, bandwidth, and packet loss the network emulator netem[12], which is included in the Linux kernel, was used. It allows to define filter rules for specific hosts. The measurement worker script extracts the network shaping parameters from the specification file and configures netem accordingly. Netem configuration was done using the tc command line tool from the *iproute2* package[7]. Examples of how tc was used can be found in the following Sections 4.3.1 and 4.3.2.

[7] `https://wiki.linuxfoundation.org/networking/iproute2`

4.3.1 Delay and delay variation

Delay and delay variation (often called jitter) can be emulated within the same netem filter rule. Because multiple filter rules were to be applied first a classful *queueing discipline (qdisc)* had to be created. qdisc is the term that describes a scheduler in the Linux traffic control system[8]. qdiscs are responsible for applying traffic shaping rules to packets. The *prio* qdisc was chosen as it allows multiple filters per traffic shaping rule by creating a class (in the example in Listing 2 the class is 1:1) and assigning filter rules to it. The example shows the tc usage for configuring a one-way delay of $60ms \pm 10ms$. tc employs a normal distribution with a mean $\mu = 60ms$ and a standard deviation $\sigma = 10ms$ to vary the delay, so that 68.27% of the emulated delay values are within the standard deviation. Calculating the actual delays for emulation is done by the netem kernel component[9].

```
1 tc qdisc add dev eth0 root handle 1: prio
2 tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 60ms 10ms
3 tc filter add dev eth0 protocol ip parent 1: prio 1 u32 match ip dst\
      130.149.221.140 flowid 1:1
4 tc filter add dev eth0 protocol ipv6 parent 1: prio 2 u32 match ipv6 dst\
      2001:638:809:ff14:130:149:221:140 flowid 1:1
5 tc filter add dev eth0 protocol ipv6 parent 1: prio 2 u32 match ipv6 dst\
      fe80::6011:7aff:fe51:3c9b
```

Listing 2: tc usage for emulating a delay of 60 ms with a delay variation of 10 ms. **Line 1:** add qdisc *1*, **line 2:** add network shaping rule to the queueing discipline, **lines 3-5:** apply network shaping to packets destinated to the target peer's IP addresses.

4.3.2 Packet loss and low bandwidth

Like delay and delay variation also packet loss and low bandwidth can be emulated using netem. All network properties to be emulated have to be added to the same queuing discipline in the same call so that the filters for specifying the target IP addresses have to be added only once. Listing 3 shows a `tc` call for adding delay, delay variation, packet loss and low bandwidth.

```
1 tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 60ms 10ms loss 1% rate
      200kbit
```

Listing 3: tc usage for emulating a delay of 60 ms, a delay variation of 10 ms, a packet loss rate of 1% and a bandwidth of 200 kbps. The network shaping is added to an existing qdisc *1*.

4.4 Establishing the topology

Although the specification format described in Section 4.2 allows specifying arbitrary topologies, the only topology the Python scripts are known to work with and that was used for measurements in this thesis is a direct connection between two GNUnet peers. In order to isolate the peers it was necessary to disable GNUnet's automatic learning of other peers. By default a GNUnet peer learns about new peers in four different ways[27]:

- peer information bundled with the software package
- UDP neighbour discovery in a LAN (IPv4 broadcast, IPv6 multicast)

[8] `http://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html`

[9] tc passes a CDF (cumulative distribution function) table to netem which is *"generated as part of the iproute2 compilation and placed in /usr/lib/tc"*[20], netem then uses it to calculate the pseudo-randomly distributed delays for emulation (see function `tabledist` in `net/sched/sch_netem.c`, linux kernel 3.16[25])

- topology gossiping (already connected peers send information about other peers)
- the HOSTLIST service (getting peer information from bootstrap servers)

Using bundled peer information was disabled using the `USE_INCLUDED_HELLOS = NO` configuration option in the `peerinfo` section. By default peers introduce themselves to their network by sending UDP packets to the IPv4 broadcast or IPv6 multicast addresses. This UDP neighbour discovery can be disabled using the `BROADCAST = NO` option in the `transport-udp` section. Topology gossiping is sending peer information to already connected peers (`TOPOLOGY` service). There are also bootstrap servers which provide lists of peers as HTTP download (`HOSTLIST` service). Both mechanisms can be disabled by not starting the respective service. The complete GNUnet configuration file used for all experiments can be found in Appendix A.

After these mechanisms were disabled the link between the two peers was established by exchanging peer information out-of-band, that is passing GNUnet HELLO URLs (see Section 3.3) to the `gnunet-peerinfo` command-line tool.

4.5 Audio setup

For providing virtual recording and playback devices on the virtual machines the Pulseaudio commandline tools were used. Listing 4 shows the initialization procedure for both a virtual microphone and a virtual speaker. After (re)starting the Pulseaudio daemon two *null sinks* are created. Pulseaudio sinks have monitors which can be used as sources. Thus the output sink is configured as the default sink for applications playing audio and the monitor of the input sink is configured as the default source for applications recording audio.

```
1 pulseaudio --kill
2 pulseaudio --start
3 pactl load-module module-null-sink sink_name=input
4 pactl load-module module-null-sink sink_name=output
5 pactl set-default-sink output
6 pactl set-default-source input.monitor
```

Listing 4: Audio initialization on the virtual machines

For audio playback and recording the commandline tools `paplay` and `parecord` which are installed with pulseadio were used. Listing 5 shows the commands used.

```
1 paplay --device=input $AUDIO_FILE
2 parecord --device=output.monitor --file-format=wav > $RECORD_FILE
```

Listing 5: Playback with paplay and recording with parecord

4.6 Verifying the setup

tcpdump records were stored for all experiments as mentioned in Section 4.2. Randomly chosen records were analyzed in Wireshark in order to verify that the automatic GNUnet configuration for exclusively using UDP or TCP worked: By checking in Wireshark that either only UDP or only TCP packets were sent between the involved machines, it was confirmed that the configuration scripts were working correctly.

Ping[10] and iperf[11] were used to check that the network shaping was in effect for randomly chosen experiments. Ping provides statistics about RTT, the RTT's standard deviation and packet loss while iperv allows bandwidth measurements.

[10] `https://wiki.linuxfoundation.org/networking/iputils`
[11] `https://iperf.fr`

After the experiment was initialized (that is network shaping was done) a ping and ping6 (for ICMPv6) measurement with default parameters was performed between the involved worker machines. The measured RTT and packet loss were compared to the emulated delay (which was expected to be half the measured RTT) and packet loss. In order to verify the emulated delay variance, that is the emulated delay's standard deviation (see Section 4.3.1), it has to be considered that the RTT's standard deviation measured with ping contains the standard deviations of two one-way delays (referred to as σ_1 and σ_2) because delay was emulated in both directions. As the emulation is symmetrical, we assume $\sigma_1 = \sigma_2$ and thus the measured RTT's standard deviation and the emulated delay variation are expected to differ by a factor of $\sqrt{2}$:

$$\sigma_{RTT} = \sqrt{\sigma_1^2 + \sigma_2^2} \tag{1a}$$

$$= \sqrt{2 \cdot \sigma_1^2} \tag{1b}$$

$$\Leftrightarrow \sigma_1 = \frac{1}{\sqrt{2}} \sigma_{RTT} \tag{1c}$$

The RTTs, RTT standard deviations and the packet loss rates measured with ping complied with the emulated delays, delay variations and packet loss rates. Thus a functional shaping of these network properties can be assumed.

For verifying that the bandwidth emulations were in effect, the bandwidth was measured with iperf for the network shapings of randomly chosen experiments. iperf was started in server mode on one peer and in client mode on the other peer. It then transfers data for a specified amount of time and the server reports the measured bandwidth in specified intervals. Measurements were done for the IPv4 and IPv6 addresses (for IPv6 the `-V` option has to be specified) and using both TCP and UDP in separated measurements. Listing 6 shows an example for measuring bandwidth over TCP using IPv4.

```
1 iperf -s -i 10 -p 9999
2 iperf -c 172.29.0.5 -p 9999 -t 60
```

Listing 6: measuring bandwidth with iperf using TCP. **Line 1:** start iperf in server mode (measured bandwidths are reported in intervals of 10 s), **line 2:** start iperf in client mode (stop sending traffic after 60 s)

For measuring bandwidths over UDP a bit rate for sending traffic has to be specified. As UDP does not provide congestion control, packets will be dropped if this rate is higher than the available bandwidth. Thus if the client sends with a bit rates higher than the expected bandwidth, the server is expected to measure the maximum available bandwidth. For all iperf measurements over UDP 110% of the expected bandwidth was specified at the client. Listing 7 shows an example of such a measurement for an expected bandwidth of 200 kbps. The bandwidth measured at the server was around 194 kbps which is approximately the emulated value.

```
1 iperf -s -i 10 -u -p 9999
2 iperf -c 172.29.0.5 -p 9999 -u -t 60 -b 220K
```

Listing 7: measuring bandwidth with iperf using UDP. **Line 1:** start iperf in server mode (measured bandwidths are reported in intervals of 10 s), **line 2:** start iperf in client mode (for expected bandwidth + 10%; stop sending traffic after 60 s)

The bandwidths measurements using iperf showed the expected results which indicates that bandwidths were shaped correctly.

5 Measurement tools

For measuring several GNUnet command line tools and GNUnet services had to be modified. Section 5.1 describes these changes. Section 5.3 gives an overview on the sizes used for the ping packets.

5.1 GNUnet command line tools

For most GNUnet modules command-line tools exist for testing, scripting and also measurement purposes. The naming convention for these tools is `gnunet-<module name>`. For instance the command-line tool `gnunet-cadet` provides information about CADET tunnels and can also be used to establish an end-to-end encrypted connection (by running `gnunet-cadet <peer id> <port>`) to a listening GNUnet peer (that is `gnunet-cadet -o <port>` was run). Data to transmit is read from stdin and output to stdout at the receiver, very similar to a telnet client.

All relevant command-line tools were extended to support RTT measurements. Also some of the services had to be modified to allow service-internal delay measurements such as encryption delay. All modifications made to the GNUnet source code can be found in the `improving-conversation` branch in the official GNUnet git repository[12]. The modifications to the GNUnet command-line tools and services are described in the sections 5.1.1 through 5.1.5. The measured values are printed to stdout or written to CSV files. See Appendix C for details.

5.1.1 gnunet-conversation

`gnunet-conversation` is the only one of the discussed command-line tools that is interactive. Once started it can be controlled with these commands:

Command	Description
/help	lists available commands or prints help about a specific command
/address	prints own conversation address
/call	initiate call using GNS address of a peer
/accept	accept incoming call on specific line
/suspend	suspend active call
/resume	resume incoming call on specific line or resume outgoing call
/cancel	reject incoming call or terminate active call
/status	print information about available lines, current calls etc
/quit	quit the application

The `/address` command prints an address that is used internally to initiate a call. It consists of two base32-encoded strings: the peer ID which is 52 bytes in length and a 102 characters long string containing the phone line (port equivalent) where gnunet-conversation is listening, and the CADET port. In mainline GNUnet this address can be used for debugging but not as an argument to the `/call` command. Instead it is expected to make calls using a GNS[13] address (which has the form `username.gnu`). Users are expected to store a PHONE record in their GNS zone using the `gnunet-namestore`

[12] `https://gnunet.org/git/gnunet.git/`

[13] GNU name system, see https://gnunet.org/gns-implementation

command-line tool or the GUI application `gnunet-namestore-gtk`[14]. gnunet-conversation then passes the given GNS address to the GNS service which resolves it to a conversation address. In order to avoid this procedure each time a CONVERSATION delay measurement is initialized the implementation of the `/call` command was extended. Now it accepts conversation addresses directly besides GNS addresses (the distinction is done by checking for a GNS ending, that is .gns or .zkey).

An echo service and RTT measurements were added to gnunet-conversation. The packets used for RTT measurements are referred to as *GNUnet echo requests* as they have the same role as *ICMP Echo Requests*. The echo requests contains an identifier and dummy data, so that the average size of CONVERSATION audio packets is met (see Section 5.3). The echo service sends copies of these packets back and thus these are referred to as *GNUnet echo responses*. The echo service can be activated with the `-e` command line option and the RTT measurement functionality comprise three command line options: `-r` for activating it (it will begin when a call starts), `-n` for specifying the number of RTT measurements to be done and `-w` for specifying a timeout for each RTT measurement. See Appendix D for documentation about the existing and newly implemented command line options. Measured RTT values are logged to a file in the CSV format (see Appendix C for details about output files).

If RTT measurements were activated when gnunet-conversation was started and a call starts (that is it has been accepted by the receiver; it does not matter which side has initiated the call) the current time is stored and the first GNUnet echo request is sent. GNUnet echo requests include the timestamp, too. It is used as an identifier for sorting out delayed GNUnet echo responses. If the identifier of an incoming GNUnet echo response matches the last GNUnet echo request, the time since sending it is logged and the measurement counter is increased. If a timeout was specified and the timeout value is reached, an invalid value (-1) is logged. Lost packets are not considered for the measurement counter, so that after an experiment has finished successfully, the number of *valid* values in the output file equals the specified count (`-n` option).

GNUnet echo requests are sent in fixed time intervals in order to minimize effects of different bandwidth usage for different emulated delays (currently the interval is hard-coded as 300 ms, which is at least 100 ms greater than the RTT caused by delay emulation). This means, when an echo response is received, the task for sending the next GNUnet echo request is scheduled to be executed after $300ms - RTT$.

Due to the codec abstraction used in the conversation API the audio data is not processed in the source code of gnunet-conversation, but inside the MICROPHONE library[15] and the SPEAKER library[16]. This made it necessary for both the echo service and the RTT measurement implementations to create dummy microphone and dummy speaker types and functions. For the echo service the dummy speaker copies all incoming payload data into the dummy microphone by calling the handler for recorded audio data. For sending GNUnet echo requests the handler for enabling the dummy microphone does not enable the hardware microphone but instead schedules a task for sending a GNUnet echo request.

5.1.2 CONVERSATION service and audio helpers

Delays were measured not only by sending GNUnet echo requests containing dummy data (referred to as *ping method*) but also by sending GNUnet echo requests containing actual

[14] for a detailed tutorial on how to use gnunet-conversation see https://gnunet.org/first-steps-using-gnunet-conversation

[15] see `https://gnunet.org/doxygen/d5/d5c/group__microphone.html`

[16] see`https://gnunet.org/doxygen/d4/d62/group__speaker.html`

audio data during a call (referred to as *call method*). The call method differs from the ping method in that audio packets are sent as they are generated by the microphone whereas the ping method sends GNUnet echo requests in a fixed interval. This results in a higher packet rate for the call method.

For the RTT measurements the called gnunet-conversation instance has to be operated with an enabled echo service (the same one used for the ping method). As the echo service was implemented using a dummy microphone and dummy speaker, no encoding and decoding happens on the target peer. In order to be able to nevertheless measure RTT, encoding delay and decoding delay in the same run, both the encoding delay and the decoding delay were measured on the peer that initiated the call. Figure 3 illustrates this setup.

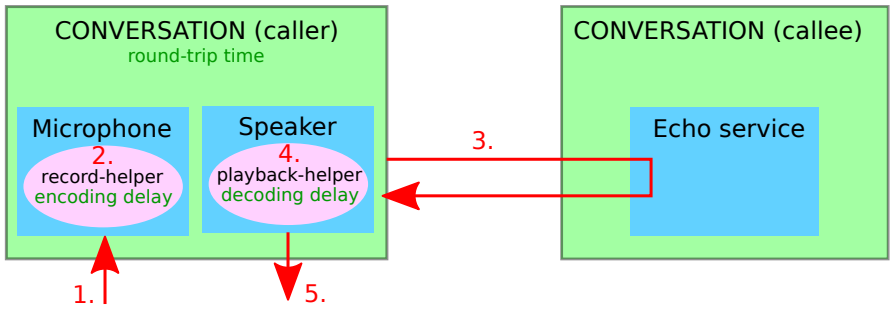


Figure 3: Setup of of the *call method*: **1.** recording playback from *input sink* monitor, **2.** encoding audio data; measuring encoding delay, **3.** sending encoded data and receiving reply from echo service; measuring RTT time, **4.** decoding audio data; measuring decoding delay, **5.** playback to *output sink*

For implementing the call method the CONVERSATION service had to be modified. Extending the CONVERSATION API was avoided and thus measuring delay using the call method and measuring encoding/decoding delay happens during every call and cannot be switched on or off from the command line. Instead both measurements can be disabled at compile time if `#define MEASURE_DELAY 1` is removed from `conversation.h`. In function `transmit_call_audio()` (`conversation_api_call.c`) before passing the audio data to the cadet service using a GNUnet message a timestamp is included into that message. When a packet that was sent back by the other peer's echo service is handled by function `handle_call_audio()`, the included timestamp is unpacked and the time difference is logged to a CSV file.

As described in Section 3.6.1 audio encoding and decoding, that is calling the libopus API, happens in extra helper binaries. For measuring the encode delay the execution time of the `opus_encode_float()` function was measured in `gnunet-helper-audio-record.c`, for measuring decode delay it is the function `ogg_demux_and_decode()` function in `gnunet-helper-audio-playback.c`. The audio codec delay measurements, like the cryptography delay measurements described in Section 5.1.4, were done using the `GNUNET_TIME_absolute_get_duration()` function from GNUnet's TIME library. The measured values of both the encoding and decoding measurements are written to separate CSV files.

5.1.3 gnunet-cadet, gnunet-core and gnunet-transport

Like for gnunet-conversation RTT measurements and an echo service were implemented in gnunet-cadet, gnunet-core and gnunet-transport. The RTT measurement behaviour is the same as for the *ping method* in gnunet-conversation (see Section 5.1.1), the only difference is that the measurement starts soon after the tool was started because unlike

in gnunet-conversation there is no user interaction required to establish a connection to the other peer. Without user interaction there was no need to write measurement values to output files. Instead they are simply printed to stdout. Further information about the GNUnet echo request packets on the different layers can be found in Section 5.3, Appendix D provides documentation about the command-line options, both the existing and the newly implemented ones.

5.1.4 CADET service

While for the RTT measurements on the CONVERSATION layer only the GNUnet command-line tools had to be modified, CADET's cryptography delay required modifications of the CADET service. The encryption and decryption delays arise and were measured in CADET's service component and the overall RTT delays was measured in the application (that is gnunet-cadet). This means that encryption and decryption delays are contained in the overall RTT delay which is also valid for the echo service side. The procedure (which is illustrated in Figure 4) is as follows:

1. At the measuring peer gnunet-cadet generates a GNUnet echo request packet containing an identifier and dummy data and passes it to the CADET service (destination: target peer)
2. The CADET service encrypts the packet, stores an **encryption delay** record and transmits it to the target peer
3. At the target peer the CADET service decrypts the incoming packet and passes the plaintext to the echo service (within gnunet-cadet)
4. The echo service copies the packet and passes it to the CADET service (destination: measuring peer)
5. The CADET service encrypts the packet and sends it back to the measuring peer
6. The CADET service at the measuring peer decrypts the incoming packet, stores a **decryption delay** record and passes it to the application (gnunet-cadet)
7. gnunet-cadet calculates the time since the last GNUnet echo request was generated and stores an **RTT** record

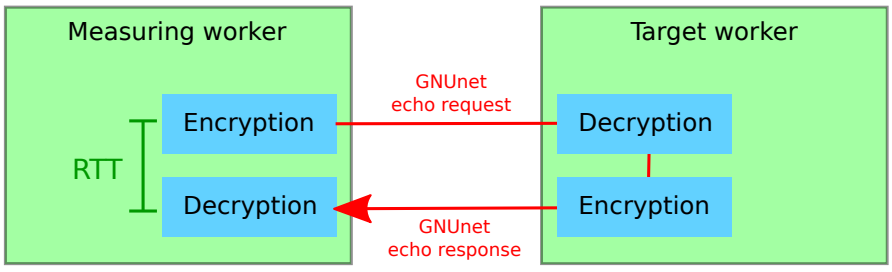


Figure 4: Setup for RTT measurements on CORE and CADET layer

This means each RTT measured on the CADET layer contains the encryption and decryption delays of both the measuring worker and the target worker. In Chapter 6 where the measured delays are analyzed the cryptography delays were analyzed separately from the remaining components of the measured RTTs. Section 5.2 describes the math behind this separation.

For measuring the encryption and decryption delays at the measuring worker, the code parts responsible for encrypting and decrypting a CADET packet were identified. The delays those parts cause were measured using the timestamp data type (`GNUNET_TIME_Absolute`) and the function `GNUNET_TIME_absolute_get_duration()` from GNUnet's `TIME` library[17]. The timestamp datatype stores an absolute timestamp in μs as 64-bit unsigned integer and the function calculates the difference of two such integers. The measured encryption and decryption delays in μs were logged into separate files. These modifications of the CADET service can be disabled at compile time by removing `#define MEASURE_DELAY 1` from `cadet.h`.

5.1.5 CORE service

Analog to the encryption/decryption delay measurements in the CADET service, encryption/decryption delays of CORE's OTR implementation was measured in the CORE service. Removing `#define MEASURE_DELAY 1` in `core.h` disables these measurements at compile time.

5.2 Separating cryptography delays from the measured RTTs

The cryptography delays were measured separately on the CADET and CORE layer as described in the previous sections. For plotting and analyzing them as component of the overall delay, twice the encryption delay and twice the decryption delay had to be subtracted from each measured RTT value as described in Section 5.1.4.

As for all RTT measurements on the CADET layer the same payload sizes were used and the same is true for the CORE layer, the encryption and decryption delay was assumed to be independent from the network shaping. This allowed calculating means for encryption and decryption delays over all measurement values separately for CADET and CORE (the resulting means are listed in Section 6.1.1), and subtracting them from each RTT. This method was used for all plots in Chapter 6 showing separated encryption and decryption delay.

5.3 Packet sizes

In order to obtain comparable results for delays measured on different layers, the GNUnet echo request packets sent over the network have to be the same size for all measurements. This means for each layer a different payload size had to be determined which considers the header sizes of all involved layers and encryption overhead (when encryption is involved).

The payload sizes were determined using the command-line tool `gnunet-statistics` which displays usage information that the GNUnet services provide to the STATISTICS service. Most services provide the sizes of the payloads they send and receive. Table 1 lists the resulting packet sizes.

GNUnet layer	Payload size in Byte	Definition
CONVERSATION	292	gnunet-conversation.c
CADET	300	gnunet-cadet.c
CORE	445	gnunet-core.c
TRANSPORT	663	gnunet-transport.c

Table 1: Payload sizes of GNUnet echo request packets on the different GNUnet layers

[17] `https://gnunet.org/doxygen/d9/d7d/group__time.html`

6 Measurements

In this chapter the conducted measurements are described and the results are analyzed. First it is analyzed what shares of the total delay the different GNUnet layers have under different conditions. Section 6.1 describes delay measurements between two peers, with focus on the network parameters jitter, packet loss and bandwidth. The delays on the CADET layer are treated separately. In Section 6.2 two different methods for measuring RTT delay on the CONVERSATION layer are compared. The evaluation of the subjective assessments of the speech recordings can be found in Section 6.3.

6.1 Latency by Layer

Round-trip time measurements were conducted between two virtual machines. In parallel cryptography delays on the CORE and CADET layer and audio codec delays on the CONVERSATION layer were measured. These measurements are described in detail in Section 5.1.

For each experiment 1000 values of each measured quantity were recorded, e.g. in an experiment on the CORE layer 1000 network RTT values, 1000 encryption delay values and 1000 decryption delay values were recorded.

The delay measurements on the CONVERSATION layer differ from those on the other layers. As described in Section 5.1.1 delay was measured with two different methods: Using ping packets like on the other layers and during a call using audio data recorded by the virtual microphone. This was done in order to assess how realistic the results of the ping measurements are compared to real conversations. Measuring with both methods resulted in twice the number of RTT values on the CONVERSATION layer compared to the other layers. In the plots where different layers are compared the CONVERSATION delay always stems from the ping method.

In addition to the RTT measurements separate experiments were conducted for subjective QoE assessment, hence the `[qoe_]` infix in the CONVERSATION column of Table 2. For subjective assessment decoded audio data sent over the network *once* (in contrast to packets sent back by an echo service without decoding) is needed. Thus, in contrast to the RTT measurements described in Section 5.1.2, in these experiments the audio was recorded on the target worker (the machine receiving the call).

Table 2 lists the IDs of all conducted 'latency by layer' experiments. The experiment IDs which are referenced in the plots and the analysis below correspond to the file names of the experiment specifications in the YAML format described in Section 4.2.

The emulated parameters delay, delay variance and packet loss were chosen in order to meet scenarios in which devices are communicating over WiFi and broadband connections. E.g. delay was varied in the range $20ms..100ms$ which are typical delays between devices that are both connected over broadband technologies such as DSL or cable. Packet loss rates in the range 1%..5% may occur in WiFi environments under bad conditions.

The range of the emulated bandwidth was chosen by making test calls. The calls showed good quality at 200 kbps and bad quality at 140 kbps.

Default values were used for shaped bandwidth and delay, that is in experiments where bandwidth emulation is not specified, an emulated bandwidth of 100 Mbit/s was used, not specifying delay emulation means that an emulated delay of 20 ms was used.

In the measurement scripts it can be specified whether GNUnet should be configured to use exclusively either UDP or TCP. In order to not leave the decision up to GNUnet ATS (see section 3.3) and risk switching from one transport protocol to the other during an experiment, all experiments were conducted with either exclusive UDP or exclusive TCP

usage. Considering the separate experiments for subjective QoE assessment this leads to a total number of 200 experiments: 20 TRANSPORT experiments + 20 CORE experiments + 20 CADET experiments + 40 CONVERSATION experiments, all conducted with both TCP-only and UDP-only configuration.

			GNUnet layer			
			TRANSPORT	**CORE**	**CADET**	**CONVERSATION**
		N/A	t_ideal_0	cr_ideal_0	cd_ideal_0	cv_[qoe_]ideal_0
network shaping	**delay**	20ms	t_delay_0	cr_delay_0	cd_delay_0	cv_[qoe_]delay_0
		40ms	t_delay_1	cr_delay_1	cd_delay_1	cv_[qoe_]delay_1
		60ms	t_delay_2	cr_delay_2	cd_delay_2	cv_[qoe_]delay_2
		80ms	t_delay_3	cr_delay_3	cd_delay_3	cv_[qoe_]delay_3
		100ms	t_delay_4	cr_delay_4	cd_delay_4	cv_[qoe_]delay_4
	PDV	5%	t_pdv_0	cr_pdv_0	cd_pdv_0	cv_[qoe_]pdv_0
		10%	t_pdv_1	cr_pdv_1	cd_pdv_1	cv_[qoe_]pdv_1
		15%	t_pdv_2	cr_pdv_2	cd_pdv_2	cv_[qoe_]pdv_2
		20%	t_pdv_3	cr_pdv_3	cd_pdv_3	cv_[qoe_]pdv_3
		25%	t_pdv_4	cr_pdv_4	cd_pdv_4	cv_[qoe_]pdv_4
	packet loss	1%	t_loss_0	cr_loss_0	cd_loss_0	cv_[qoe_]loss_0
		2%	t_loss_1	cr_loss_1	cd_loss_1	cv_[qoe_]loss_1
		3%	t_loss_2	cr_loss_2	cd_loss_2	cv_[qoe_]loss_2
		4%	t_loss_3	cr_loss_3	cd_loss_3	cv_[qoe_]loss_3
		5%	t_loss_4	cr_loss_4	cd_loss_4	cv_[qoe_]loss_4
	bandwidth	200kbps	t_bandwidth_3	cr_bandwidth_3	cd_bandwidth_3	cv_[qoe_]bandwidth_3
		180kbps	t_bandwidth_7	cr_bandwidth_7	cd_bandwidth_7	cv_[qoe_]bandwidth_7
		160kbps	t_bandwidth_8	cr_bandwidth_8	cd_bandwidth_8	cv_[qoe_]bandwidth_8
		140kbps	t_bandwidth_9	cr_bandwidth_9	cd_bandwidth_9	cv_[qoe_]bandwidth_9

Table 2: experiments conducted to determine delay by GNUnet layer. Default network shaping parameters: 100 Mbit/s bandwidth, 20 ms delay, no delay variation, no packet loss. The prefixes *t*, *cr*, *cd*, *cv* stand for the GNUnet layers *TRANSPORT*, *CORE*, *CADET* and *CONVERSATION*, the following expression is the emulated metric that was varied

6.1.1 Determining the encryption and decryption delay

As described in Sections 5.1.4 through 5.2 the measured RTT delays contain encryption and decryption delays and thus in order to calculate the network delays for plotting it was necessary to measure those delays separately and subtract them from the measured delays.

The encryption and decryption delays were recorded on the measuring worker in all RTT experiments on the CORE and CADET layers (see Section 5.1.4). A mean and standard deviation was calculated over all those values. For both CORE and CADET these sum up to 40000 encryption delay values and 40000 decryption delay values (20 experiments using UDP only, 20 experiments using TCP only, 1000 measuring values per experiment).

Table 3 shows the results. The standard deviations of CADET's encryption and decryption are higher than expected. A look into the dataset showed outliers with both significantly higher and lower delays. Presumably these are caused by packets other than the GNUnet echo requests, e.g. packets responsible for CADET's re-keying described in Section 3.5.2. These may have different packet sizes than the GNUnet echo requests and

thus may cause different encryption and decryption delays.

GNUnet layer	Encryption delay	Decryption delay
CADET	$5.980ms \pm 1.587ms$	$1.377 \pm 0.673ms$
CORE	$0.496ms \pm 0.057ms$	$0.501 \pm 0.065ms$

Table 3: Mean and standard deviation of encryption and decryption delays calculated over 460460 delay values for both CADET and CORE

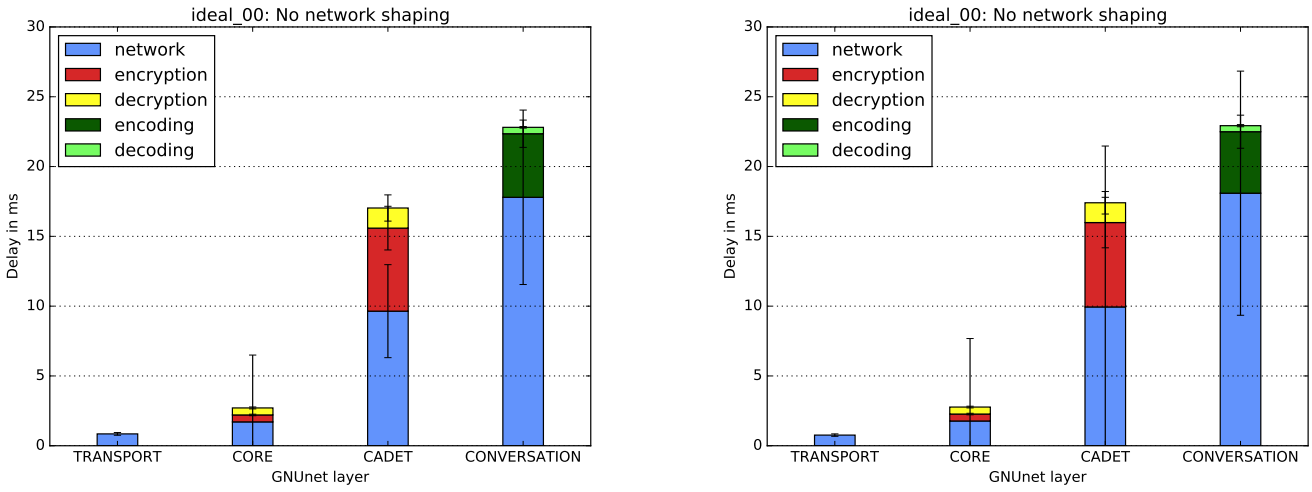
6.1.2 No network shaping

The first line in table 2 lists delay measurements on the four layers without any network shaping. As the two virtual machines used for these experiments are on the same host machine and communicate over a virtual network bridge the physical network delay is expected to be very low. An RTT measurement using ping without GNUnet involved was done as shown in listing 8. The resulting delay ($RTT/2$) is 0.230 ms with a standard deviation of 0.029 ms (12.4%).

```
$ ping -c 1000 -s 725 172.29.0.5
[...]
rtt min/avg/max/mdev = 0.309/0.460/0.939/0.057 ms
```

Listing 8: Ping measurement with ICMP ECHO_REQUESTs using a count of 1000. The packet size 725 equals the one used in gnunet-transport (see section 5.3)

The TRANSPORT layer is expected to add a constant (that is low standard deviation) processing delay to this. ATS as a potential source of unpredictable delays is prevented from switching transport protocols by the GNUnet configuration.



(a) TCP only

(b) UDP only

Figure 5: Measured delay (mean and standard deviation) without network shaping

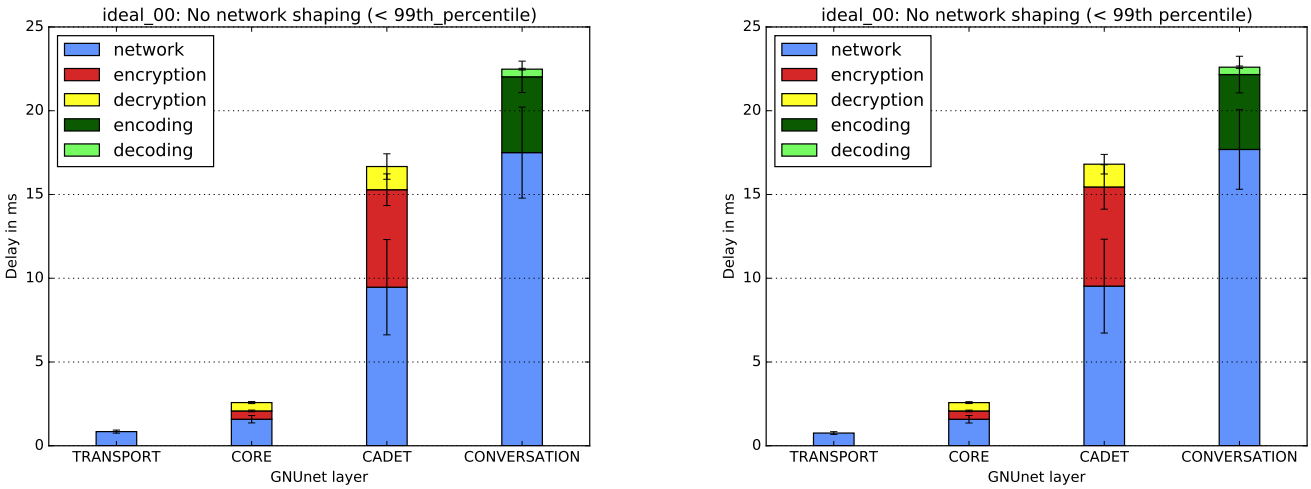
Figure 5a shows the means and standard deviations of the measured delays using TCP only. The delay TRANSPORT added to the network delay measured with ping is approximately 0.6 ms (0.85 ms in total) which is low compared to the 22.81 ms total

delay. This can be be considered as processing delay, e.g. inter-process communication as described in section 5.1.1. The delay increase from TRANSPORT to the CORE layer is roughly 2.1 ms. Half of this increase is caused by OTR encryption and decryption. These delays sum up to approximately 1 ms. The remaining 1.1 ms may again result from IPC communication. The most drastic increase happens from CORE to CADET. CADET adds 13.5 ms to the total CORE delay. This is the main part of the overall delay seen on the CONVERSATION layer. Double-Ratchet cryptography delays sum up to around 7.4 ms. The remaining 6.1 ms are analyzed in chapter 6.1.3. A surprising observation is that the encryption on the CADET layer takes longer than the decryption. As described in section 3.5.2 this is not expected, but could be explained by parallelization. This is also investigated in chapter 6.1.3. The network delay measured on the CONVERSATION layer differs only by roughly 1 ms from the overall CADET delay which is in the same order as the presumable IPC delays on the other layers. CONVERSATION adds an audio codec delay of 4.8 ms.

In both the TCP-only (Figure 5a) and the UDP-only (Figure 5b) plot the network delays show extreme standard deviations with significant differences between TCP and UDP: e.g. on the CADET layer the standard deviation is approximately 3 ms for TCP but almost 12 ms for UDP. In cases where UDP shows a higher standard deviations also the mean is slightly higher for UDP. This might be a hint that both effects are caused by extreme outliers.

Indeed extreme outliers were found in the datasets: e.g. for UDP at approx. 510 ms (CADET) and 390 ms (CONVERSATION) and for TCP at approx. 260 ms (CONVERSATION). Audio packets with such an extreme delay can't be used for an active audio call and thus would probably be dropped by the codec.

By plotting only the values below the 99th percentile in the figures 6a and 6b it can be estimated how the delays would look like without the outliers. Here both the TCP-only case and the UDP-only case show very similar means standard deviations between roughly 0.3 ms (CORE) and 2.5 ms (CONVERSATION). Thus both differences in the means between TCP and UDP and the extreme standard deviations seem to be caused by extreme outliers.



(a) TCP only

(b) UDP only

Figure 6: Measured delay (mean and standard deviation) without network shaping (values < 99th percentile)

In most delay measurements extreme outliers were found. Many were in the order of seconds. It can be speculated that they are caused by CADET's re-keying described in Section 3.5.2. As no clear evidence for this has been found yet, it has to be left as future work.

6.1.3 A closer look at CADET's delay

As discussed in the previous section the delay increase from CORE to CADET is mainly caused by encryption and decryption. As described in Section 3.5.2 the encryption delay being much higher than the decryption delay is unexpected but might be explainable with parallelized decryption.

The functions responsible for encryption and decryption using AES and TWOFISH, `GNUNET_CRYPTO_symmetric_encrypt ()` and `GNUNET_CRYPTO_symmetric_decrypt ()` are implemented in GNUnet's `CRYPTO` library[18]. They are wrapping calls to `libgcrypt`[19]. A program, that calls these functions in two loops was written in C. The input to the encrypt function was 256 Byte of dummy data. The resulting cyphertext was then passed to the decrypt function. The time differences these loops cause were measured using `GNUNET_TIME_absolute_get_duration()` from GNUnet's TIME library. In a measurement on one of the virtual machines with 10000 iterations per loop, it was obvious that both functions cause equal delay (see Table 4).

Function	**Data size**	**Delay**
`GNUNET_CRYPTO_symmetric_encrypt`	256 byte	$0.063ms$
`GNUNET_CRYPTO_symmetric_decrypt`	256 byte	$0.061ms$
`GNUNET_CRYPTO_ecdhe_key_get_public`	N/A	$4.3ms$

Table 4: Mean over 10000 delay measurements for GNUnet functions

A closer look into the source code revealed that during encryption (in function `GCT_send`, `gnunet-service-cadet_tunnels.c`) the function `GNUNET_CRYPTO_ecdhe_key_get_public` is called, which extracts the *Diffie-Hellman ratchet public key (DHr)*, defined in [17], from the corresponding private key. A delay measurement was implemented in C like for the encryption and decryption functions. The result is listed in Table 4: One call to `GNUNET_CRYPTO_ecdhe_key_get_public` takes 4.3 ms. This matches roughly the difference between encryption and decryption delay (4.6 ms).

`GNUNET_CRYPTO_ecdhe_key_get_public` is called for each outgoing packet although the Diffie-Hellman ratchet key changes after 64 packets have been sent or under the conditions described in Section 3.5.2. Thus storing the public key instead of calculating it on-the-fly would eliminate the delay caused by this function for most packets.

6.1.4 Emulate delay

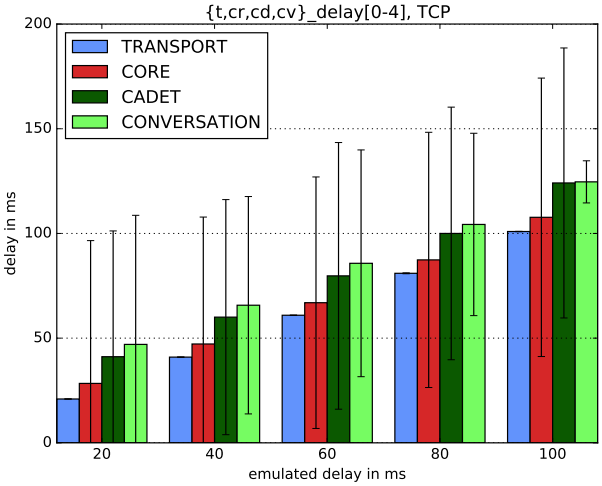
The figures 7a and 7c show a collection of measurement results from all experiments where delay was emulated, that is the emulated delay was varied from 20 ms to 100 ms. The plotted delays contain the same components as the total delays in figure 5a, that is they include cryptography and codec delays. For plotting a total mean and a total standard deviation, consisting either of network delay and encryption / decryption delay

[18] `https://gnunet.org/doxygen/d5/dfc/group__crypto.html`
[19] `https://www.gnupg.org/related_software/libgcrypt/`

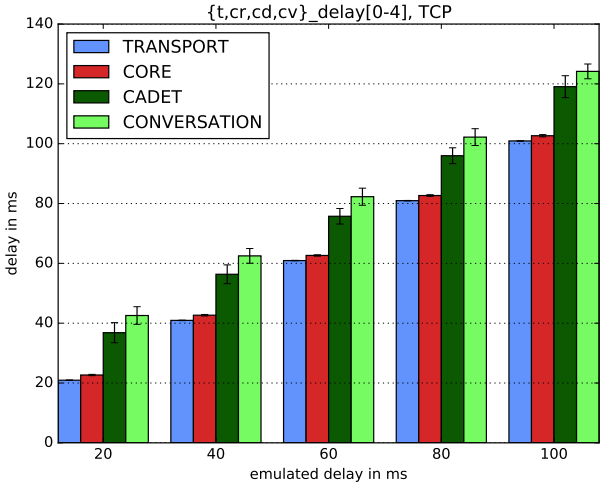
or of network delay and encoding / decoding delay, had to be calculated using Equations 2a and 2b. In both cases three delays were involved, so N is 3.

$$\mu_{total} = \sum_{i=0}^{N-1} \mu_i \tag{2a}$$

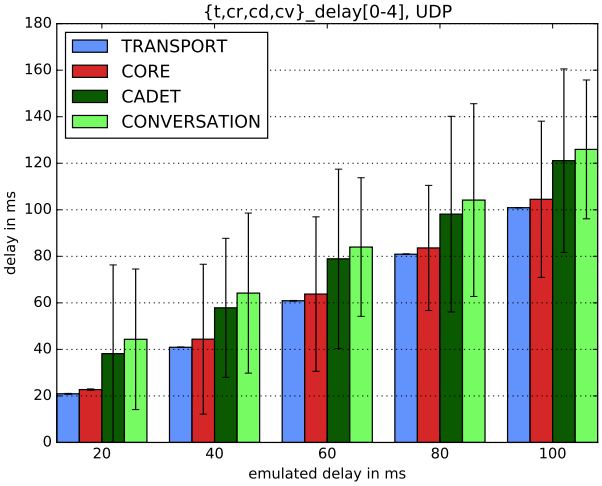
$$\sigma_{total} = \sqrt{\sum_{i=0}^{N-1} \sigma_i^2} \tag{2b}$$



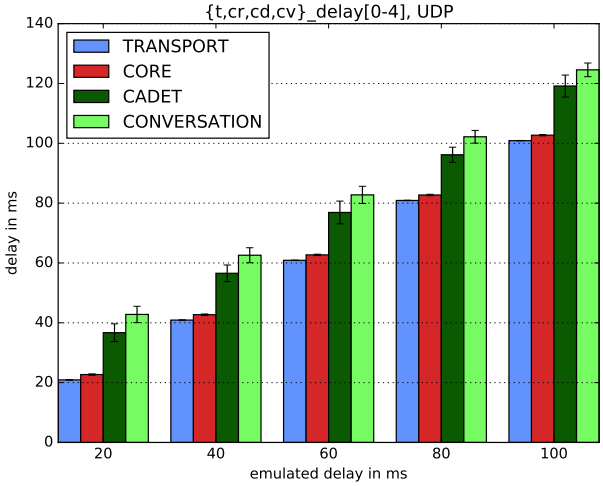
(a) TCP only



(b) TCP only, values < 99th percentile



(c) UDP only



(d) UDP only, values < 99th percentile

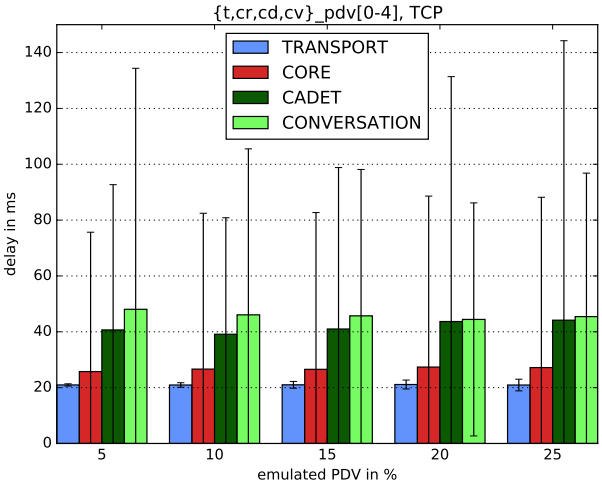
Figure 7: Measured delay (mean and standard deviation) emulating delay, bandwidth: 100 Mbit/s

As shown in section 6.1.1 the delays caused by cryptography and codec are not varying much over the course of the experiments. This and visual clarity are the reasons why they are not listed separately in the collective plots in this chapter.

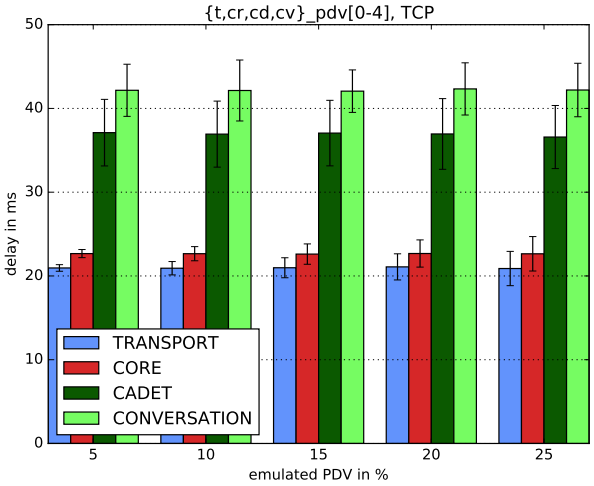
The plots show a linear increase of the measured delays on all four layers for both TCP and UDP. An emulated delay in the range 0 to 100 ms (which is ideal for voice communication) does not result in unexpected behaviour but each layer only adds the constant delay described in section 6.1.2. It seems that the difference between TRANSPORT and CORE is higher than in the previous plots. The 99th-percentile plot shows very similar proportions to the experiment without network shaping. Thus, the higher mean on the CORE layer is again caused by outliers.

6.1.5 Emulate packet delay variation

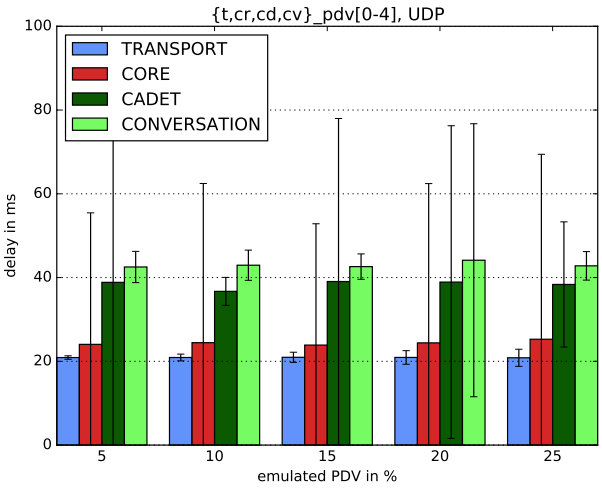
When measuring delay with emulated PDV it is expected that the standard deviation of the measured values increases with the emulated PDV increase. The figures 8a (TCP only) and 8c (UDP only) show that this is indeed the case on the TRANSPORT layer. For both cases the standard deviation increases steadily from 0.4 ms at 5% emulated PDV to 2.1 ms at 25% emulated PDV. The same cannot be said for the other layers. For example the standard deviation on the CADET layer in the UDP case is much higher at 5% PDV (43 ms) than at 10% PDV (2.8 ms). In the TCP case for CONVERSATION the standard deviation even seems to decrease from 86 ms at the lowest PDV to 41 ms at 20% PDV. Like the high standard deviation in the experiment without network shaping this is caused by outliers. In the described CONVERSATION case one outlier with a value of more than 2 seconds exists. Again only the values below the 99th percentile were plotted (figures 8b and 8d). Without the highest percentile the plots show a steady increase of the standard deviations on the layers TRANSPORT and CORE. CADET and CONVERSATION seem to be affected little or not at all by this increase on the lower layers because these standard deviations are small compared to the delay increase both CADET and CONVERSATION cause.



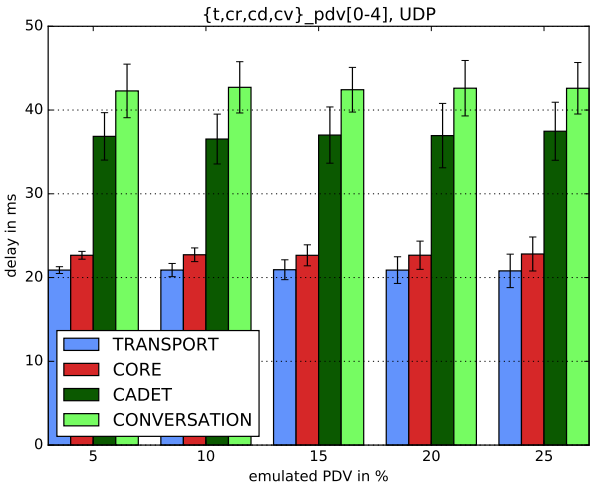
(a) TCP only



(b) TCP only, values < 99th percentile



(c) UDP only

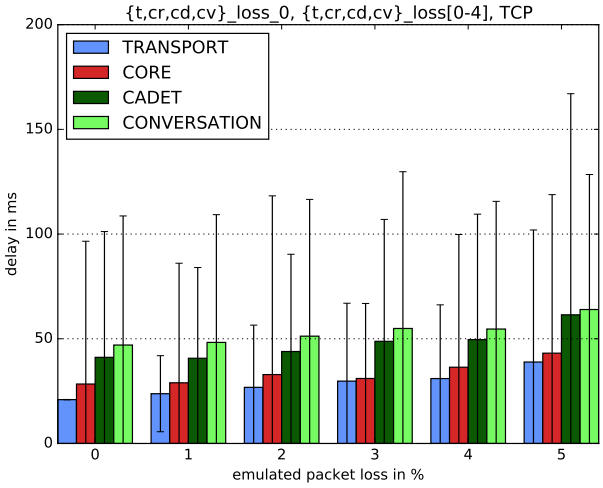


(d) UDP only, values < 99th percentile

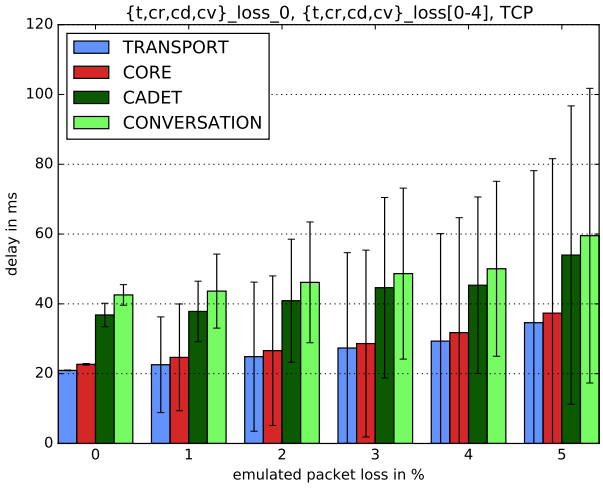
Figure 8: Measured delay (mean and standard deviation) emulating PDV

6.1.6 Emulate packet loss

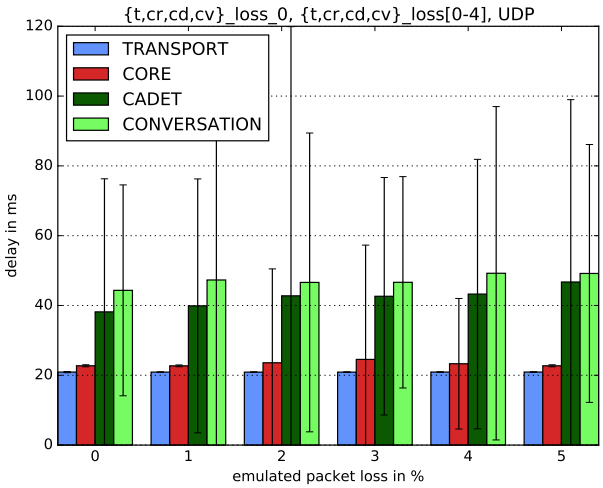
Emulated packet loss was varied from 1% to 5%. The figures 9a through 9d show the measured delays including the results without emulated packet loss taken from the experiments *delay_tcp_0* and *delay_udp_0*. In both the TCP and the UDP plots a linear increase of the measured delays can be observed. For TCP the increase per % packet loss on all layers is roughly 12%, the increase on the CADET layer being a little more fluctuating than on the other layers.



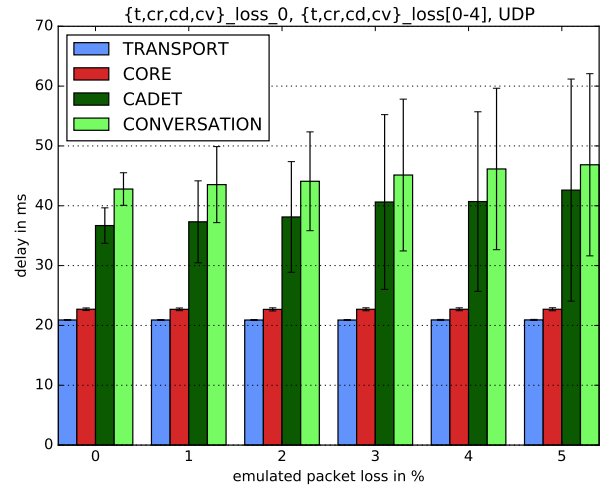
(a) TCP only



(b) TCP only, values $<$ 99th percentile



(c) UDP only



(d) UDP only, values $<$ 99th percentile

Figure 9: Measured delay (mean and standard deviation) emulating packet loss

The boxplots in Figure 10 show a lot of outliers, that is measured values outside of 1.5 IQR (inter-quartile range). For TRANSPORT there are 1129 outliers and for CORE there are 1241 outliers from 10000 measured values each. The values of most of the outliers are not as extreme as seen in the previous experiments. The vast majority is below 300 ms. So in contrast to the previously discussed experiments the outliers are an essential share of the measured values and a mean may not be the best choice for getting reliable information about datasets with such properties.

Another observation are the extreme standard deviations on all layers (TCP) and on the layers CADET and CONVERSATION (UDP). As TCP is reliable, it will retransmit lost packets which causes high variance for TRANSPORT and all layers above.

The UDP-only plots have a very low standard deviation for TRANSPORT and a much lower standard deviation for CORE than the TCP plots. This is because neither these two layers nor UDP provide reliability. gnunet-transport and gnunet-core detected the lost packets after the timeouts described in Section 5.1.3 but no retransmissions that

could have increased the variance were done. CADET's reliability functionality is in effect though, because the CONVERSATION service activates it as described in Section 3.6. Thus when CADET detects a lost packet, it initiates a retransmission. This explains that the standard deviation is high on the CADET and CONVERSATION layer.

The fluctuations in the standard deviations can be explained again with extreme outliers: the 99th percentile plots for both UDP and TCP show steady increasing standard deviations.

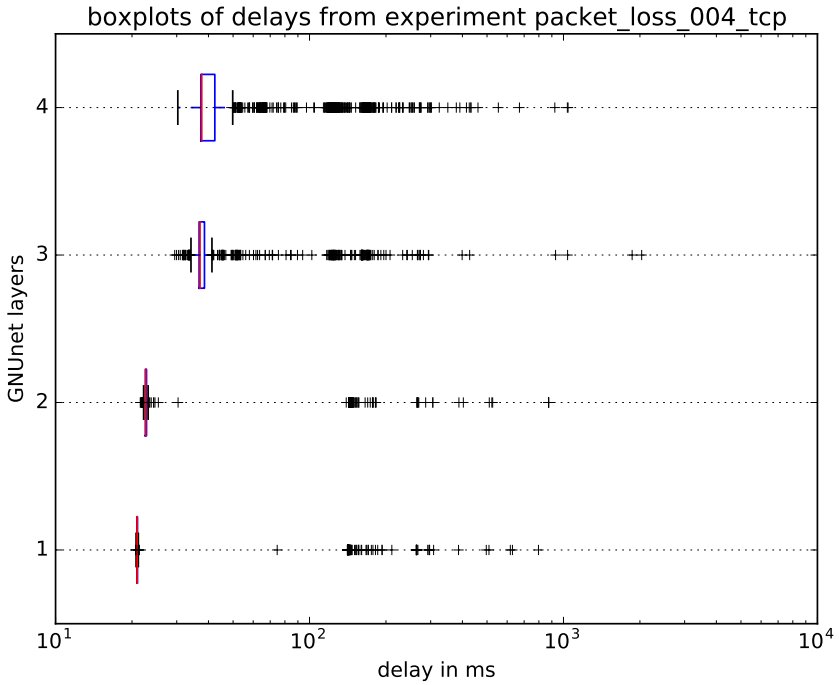


Figure 10: Boxplot of the measured network delays at 5% emulated packet loss; **1:** TRANSPORT, **2:** CORE, **3:** CADET, **4:** CONVERSATION

6.1.7 Emulate low bandwidth

Low bandwidth was emulated in the range 140 Kbit/s to 200 Kbit/s in steps of 20 KBit/s. Measurements were done like for the other metrics but the plots are omitted for the following reason.

In all plots (TCP-only, UDP-only, both 100th-percentile and 99th-percentile plots) the mean of the TRANSPORT delay was higher than for the next higher layer CORE. This clearly indicates a measurement error. As every plots was affected and a very low standard deviation existed for TRANSPORT and CORE, outlier cannot explain the effect. As the bandwidths are very low considering the codec's bit rate of 48 Kbit/s and the packet header and encryption overhead of the four layers it is possible that slight differences in the effective bit rate can lead to different queueing delays. Such differences between the layers can only exist if either the ping packets were sent in different rates or if the ping packets had different sizes. Different rates are not possible because the same hardcoded interval (300 ms) was used for all command-line tools, using the same implementation for enforcing that interval. So the second statement must be true: The ping packet sizes of different command-line tools differ in size.

The packet sizes listed in Table 1 were determined using `gnunet-statistics`. This

seems to be an unreliable method because the number of bytes displayed may be the sum of payloads of multiple packets. E.g. on the CORE and CADET layer key exchange packets may be sent in between the application data. This may have been summarized into one statistics output.

The measurement errors result from wrongly determined packet sizes. This has an effect on the experiments with low bandwidth emulation because here queueing delays increase rapidly when the effective bit rate comes close to or exceeds the available bandwidth. In the other experiments a large bandwidth of 100 Mbit/s was available, so the effect of the wrong packet sizes should not be significant.

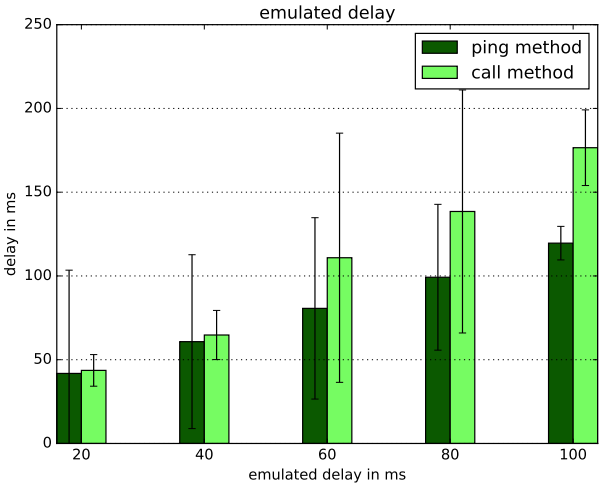
6.2 CONVERSATION ping measurements vs. call delay

In order to determine how close the RTT measurements are to a realistic scenario, that is a call with speech recorded by a microphone, the RTTs measured using the *ping method* were compared with RTTs measured using the *call method* (both defined in Section 3.6).

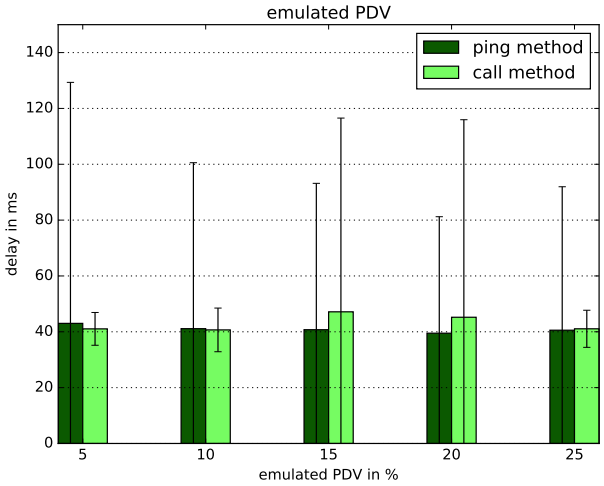
Figure 11 shows the comparison for the TCP-only experiments discussed in the previous sections. For an emulated delay of 20 ms the means for both methods are equal as the plots 11a and 11b show. For increasing emulated delay the delays from both methods increase linearly, but with a slightly higher slope for the call method. From the lowest to the highest emulated delay the measured delays increase by 286% for the ping method and by 405% for the call method. This can be explained by higher queueing delays because of the much higher packet rate. For an emulated delay of 100 ms, the measured delay for call method exceeds the recommended the recommended maximum mouth-to-ear delay (see Section 2.2). Thus, bad QoE is expected.

The higher delay values in the packet loss plot can be explained with higher queueing delay aswell.

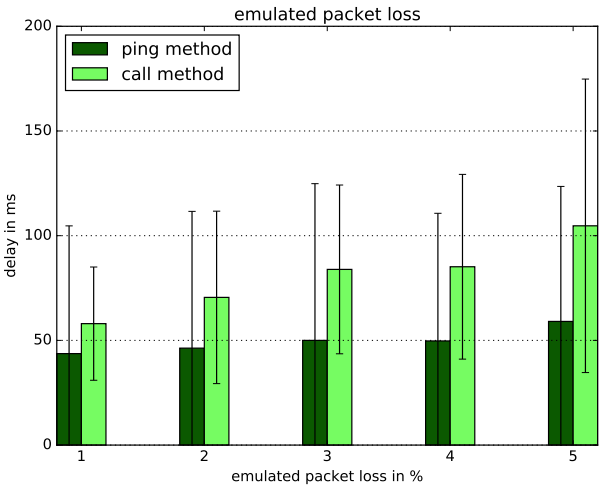
The biggest difference between the two methods can be seen in the bandwidth plot. For dereasing emulated bandwidths the delays increase only slightly for the ping method but in an extreme way for the call method. Between an emulated delay of 200 Kbit/s and 180Kbit/s the increase is 525%. This means that while still potentially suitable for good QoE at 200 Kbit/s available bandwidth, for little less than 200 Kbit/s available bandwidth the queueing delay becomes unsuitable according to the ITU-T recommendations (Section 2.2). This sudden delay increase is an indication for packet loss because of insufficient bandwidth. QoE is expected to be bad for bandwidths of 180 Kbit/s or less.



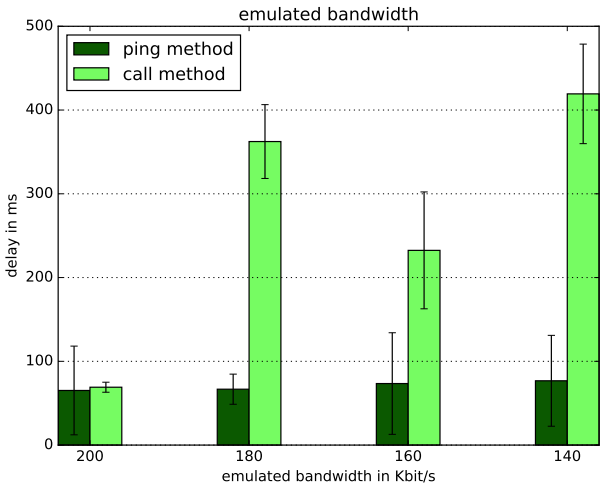
(a) parameter: delay



(b) parameter: PDV



(c) parameter: packet loss



(d) parameter: bandwidth

Figure 11: CONVERSATION delays: Comparing ping method and call method (TCP only)

6.3 Evaluation of the recorded calls

The experiments on the CONVERSATION layer for subjective QoE assessment as described in Section 2 were done separately. Of each experiment two recordings were assessed in order to determine QoE tendencies. The audio file (reference audio) used was a poem recorded in a studio by a professional speaker. At the target peer one minute of the call was recorded.

In a first evaluation four kinds of behaviour was observed. Some recordings had very short artifacts, distributed over the whole duration of the recording. These, if not occurring too often, left the speech understandable (as in every word could be understood). Other recordings showed longer silence periods up to roughly 3 seconds. This lead to four disturbance categories to be examined:

1. **up to 5 artefacts of** $< 1s$: Every word could be understood, this might still be a satisfying QoE for many users

2. **up to 20 artefacts of** $< 1s$: Almost all words could be understood, but the artefacts occur to often to provide a satisfying QoE

3. **up to 5 silence periods of** $< 5s$: Words were left out, not a satisfying QoE

4. **more artefacts / silence periods**: The disturbances occur to often to understand the speech content

Table 5 show the assessment results. It is obvious that only very few recordings fall into the first category. In the delay category only a UDP-only experiment falls into this categoriy. The bandwidth category shows that with the same delay and even a lower bandwidth TCP may perform similarly, though. Thus without assessing a greater amount of recordings it cannot be concluded, whether UDP or TCP performs better at low delays.

		Type of disturbance			
		up to 5 artefacts of $< 1s$	**up to 5 silence periods of** $< 5s$	**up to 20 silence periods of** $< 1s$	**more silence periods**
delay	0	U	T		
	1		U	T	
	2				T, U
	3				T, U
	4				T, U
PDV	0		T, U		
	1		T, U		
	2		T, U		
	3	T	U		
	4	T	U		
packet loss	0	U		T	
	1		U	T	
	2				T, U
	3				T, U
	4				T, U
bandwidth	3	T		U	
	7				T, U
	8				T, U
	9				T, U

Table 5: Subjective QoE assessment, **T**: TCP-only, **U**: UDP-only; the serial numbers correspond to those in the experiment IDs (see Table 2)

A low packet loss rate (1 or 2%) seems to have less negative effects, when UDP is used. This makes sense as TCP reliability features will increase the delay as shown in Section 4.3.2.

For emulated PDV all recordings contain at most 5 silence periods, for two of them at the highest emulated PDV (20 and 25%) show even small artefacts (first category). Thus PDV seems to affect QoE the least.

In general it can be said that bad QoE, that is more than 20 artefacts of less than 1 second or more than 5 silence periods of more than 5 seconds, will definitely occur for delays of 60 ms or more, for packet loss of 3% and for bandwidths of 180 Kbit/s or less.

7 GNUnet optimization

As shown in Section 6.1.3 an optimization with estimated big effect is avoiding calling the function GNUNET_CRYPTO_ecdhe_key_get_public during encryption for each message. Instead it should only be called once a Diffie-Hellman ratchet step was done and stored for future messages. On the virtual machines this would save 18.6% (4.3 ms) of the overall mouth-to-ear delay.

Moreover a very important task left to future work is determining the GNUnet implementation details that cause the extreme outliers described throughout Section 6 and find measures to mitigate the outliers.

Another optimization measure would be to implement the *per-packet RELIABLE / UNRELIABLE* transport for CADET. This would make it possible to send audio packets unreliably while continuing to send control packets reliably without the extra implementation overhead of a second CADET channel. A task linked to this would be to implement a way for CADET to influence the underlying transport selection. Currently the RELIABLE / UNRELIABLE option in the CADET API only enables or disables CADET's reliability mechanisms but has no effect on whether ATS selects an unreliable (UDP) or a reliable transport (TCP/HTTP, etc.). An application should be enabled to pass a preference about its reliability and performance requirements to ATS through the API. In the ATS API [20] a function for expressing performance preferences already exists and could be extended.

[20] https://gnunet.org/doxygen/d8/d82/group__ats.html

8 Conclusion

In this thesis delays were measured on the four GNUnet layers the voice application GNUnet conversation employs. The fully automated measurements were done under different network conditions: A network emulator was used to vary the metrics delay, packet delay variation, packet loss and bandwidth. GNUnet's services and command-line tools were extended to provide measurement features. The security architecture of GNUnet required the measurement of encryption and decryption delays on the layers CORE and CADET. Delay caused by the audio codec was measured on the CONVERSATION layer.

Audio recordings were done in parallel in all experiments that involved CONVERSATION calls. Separate audio recordings were done for subjective QoE assessment.

The evaluation of the measured delays showed, that on average GNUnet conversation causes 23 ms of mouth-to-ear delay on top of existing network delay, with otherwise ideal network conditions, that is, a sufficient bandwidth and no jitter or packet loss. In this case the largest part (approx. 60%) is caused by CADET. CADET's delay is mostly encryption and decryption delay (Double-Ratchet cryptography). We were able to show that optimization is possible in the encryption part of this delay by not calling the expensive function `GNUNET_CRYPTO_ecdhe_key_get_public` for every message. On systems with a performance similar to the machines used this would save more than 18% of the mouth-to-ear delay most of the time.

Although for increasing delay at the underlying network layer the average delays at the GNUnet layers were only increased by a constant and for increasing loss of IP packets only a slight delay increase could be measured (caused by TCP's and CADET's reliability mechanisms), the subjective QoE assessment mostly showed bad QoE for typical network conditions. The reasons are extreme outliers in the order of 1 second occurring in many measurements. The GNUnet implementation details responsible for this are yet to be determined.

Recommendations for good QoE with the current CONVERSATION implementation could be determined: CONVERSATION cannot provide good QoE if the delay to the other end is 60 ms or more or the available bandwidth is below 200 Kbit/s. Furthermore good QoE could only be observed with a packet loss rate of 1% or less.

Important future work includes analyzing CONVERSATION's behaviour in different network topologies, such as routes over multiple GNUnet peers or parallel routes with different network conditions. Examining the connection establishment procedure is important for a an extensive QoE analysis too. A task not directly related to this thesis, but which should not be underestimated is to improve CONVERSATION's user interfaces (command-line and graphical) so secure, distributed voice communication finally becomes usable.

References

[1] UN General Assembly. "Universal declaration of human rights, art. 12". In: *UN General Assembly* (1948).

[2] Salman A Baset and Henning Schulzrinne. "An analysis of the skype peer-to-peer internet telephony protocol". In: *arXiv preprint cs/0412017* (2004).

[3] Jan A Bergstra and CA Middelburg. "ITU-T Recommendation G. 107: The E-Model, a computational model for use in transmission planning". In: (2003).

[4] Daniel J Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* (2012), pp. 1–13.

[5] Wlodzimierz Bielecki and Dariusz Burak. "Parallelization of Standard Modes of Operation for Symmetric Key Block Ciphers". In: *Biometrics, Computer Security Systems and Artificial Intelligence Applications*. Springer, 2006, pp. 101–110.

[6] Peter Bright. *Skype finalizes its move to the cloud, ignores the elephant in the room*. URL: `http://arstechnica.com/information-technology/2016/07/skype-finalizes-its-move-to-the-cloud-ignores-the-elephant-in-the-room` (visited on 2016-10-27).

[7] Jonathan Davidson et al. *Voice over IP Fundamentals (2nd Edition) (Fundamentals)*. 2nd ed. Cisco Press, 2006. Chap. 7. ISBN: 1587052571.

[8] Morris Dworkin. *Recommendation for block cipher modes of operation. methods and techniques*. Tech. rep. DTIC Document, 2001.

[9] Nathan S Evans and Christian Grothoff. "R5n: Randomized recursive routing for restricted-route networks". In: *Network and System Security (NSS), 2011 5th International Conference on*. IEEE. 2011, pp. 316–321.

[10] *GNUnet*. URL: `https://gnunet.org` (visited on 2016-10-27).

[11] Christian Grothoff, Bart Polot, and Matthias Wachs. *A Tutorial for GNUnet 0.10.x (C version)*. URL: `https://gnunet.org/svn/gnunet/doc/gnunet-c-tutorial.pdf` (visited on 2017-06-23).

[12] Stephen Hemminger et al. "Network emulation with NetEm". In: *Linux conf au*. 2005, pp. 18–23.

[13] Rec ITU-T and I Recommend. "G. 114". In: *One-way transmission time* 18 (2000).

[14] P ITU-T RECOMMENDATION. "Subjective video quality assessment methods for multimedia applications". In: (1999).

[15] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International journal of information security* 1.1 (2001), pp. 36–63.

[16] Ewen MacAskill et al. *GCHQ intercepted foreign politicians' communications at G20 summits*. URL: `https://www.theguardian.com/uk/2013/jun/16/gchq-intercepted-communications-g20-summits` (visited on 2017-05-30).

[17] Moxie Marlinspike and Trevor Perrin. *The double ratchet algorithm*. 2016.

[18] Morgan Marquis-Boire, Glenn Greenwald, and Micah Lee. *XKeyScore - NSA's Google for the World's Private Communications*. URL: `https://theintercept.com/2015/07/01/nsas-google-worlds-private-communications/` (visited on 2017-05-30).

[19] Muhammad Amir Mehmood. "Impact of network effects on application quality". In: (2012).

[20] *netem (linux network emulator)*. URL: https://wiki.linuxfoundation.org/networking/netem (visited on 2016-10-27).

[21] Laura Poitras, Marcel Rosenbach, and Holger Stark. *GCHQ and NSA Targeted Private German Companies and Merkel*. URL: http://www.spiegel.de/international/germany/gchq-and-nsa-targeted-private-german-companies-a-961444.html (visited on 2017-05-30).

[22] Bartlomiej Polot and Christian Grothoff. "Cadet: Confidential ad-hoc decentralized end-to-end transport". In: *Ad Hoc Networking Workshop (MED-HOC-NET), 2014 13th Annual Mediterranean*. IEEE. 2014, pp. 71–78.

[23] Martin Schanzenbach. "Design and Implementation of a Censorship Resistant and Fully Decentralized Name System". MA thesis. TU München, 2012.

[24] Bruce Schneier. "Metadata= surveillance". In: *IEEE Security & Privacy* 12.2 (2014), pp. 84–84.

[25] *The Linux kernel 3.16*. URL: https://cdn.kernel.org/pub/linux/kernel/v3.x/linux-3.16.44.tar.xz (visited on 2017-06-24).

[26] Jean-Marc Valin, Koen Vos, and T Terriberry. *RFC 6716. Definition of the Opus audio codec*. Tech. rep. 2012.

[27] Matthias Wachs. *GNUnet's HOSTLIST subsystem*. URL: https://gnunet.org/gnunets-hostlist-subsystem (visited on 2017-06-23).

[28] Charles V Wright et al. "Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations". In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 35–49.

A GNUnet configuration used for measurements

```
[arm]
SYSTEM_ONLY = NO
USER_ONLY = NO

[transport]
PLUGINS = tcp
# PLUGINS = udp
# PLUGINS = tcp udp

[peerinfo]
USE_INCLUDED_HELLOS = NO

[transport-udp]
BROADCAST = NO

[hostlist]
AUTOSTART = NO
FORCESTART = NO

[fs]
AUTOSTART = NO
FORCESTART = NO

[nat]
ENABLE_UPNP = NO

[nse]
AUTOSTART = NO
FORCESTART = NO

[revocation]
AUTOSTART = NO
FORCESTART = NO

[set]
AUTOSTART = NO
FORCESTART = NO

[vpn]
AUTOSTART = NO
FORCESTART = NO

[PATHS]
GNUNET_HOME = ./test

[conversation]
LINE = 2
```

Listing 9: The GNUnet configuration without restricted transport (for UDP-only or TCP-only uncomment the respective line)

B Usage of the measurement scripts

```
Usage: mc.py [options]

Options:-
  h, --help            show this help message and exit-
  c CONFIG, --config=CONFIG
                       config file path-
  r, --reinstall-gnunet
                       push local gnunet changes to the workers and initiate
                       rebuild / reinstall-
  m, --manual          configure and start the peers and shape the network as
                       described in the given experiment file-
  e EXPERIMENT, --experiment=EXPERIMENT
                       configure and start the peers and conduct the
                       experiment discribed in the given file-
  n NUMBER, --number=NUMBER
                       run the experiment n times-
  t TRANSPORT, --transport=TRANSPORT
                       configure gnunet to use the specified transport
                       protocol (tcp/udp) only-
  w TIMEOUT, --timeout=TIMEOUT
                       configure the timeout for one experiment, default: 600
```

Listing 10: Usage of the measurement controller script

```
usage: mw.py [-h]
             {call,experiment,init-gnunet,shutdown-gnunet,set-workers,update-
                 gnunet,record-conversation,unshape}
             ...

positional arguments:
  {call,experiment,init-gnunet,shutdown-gnunet,set-workers,update-gnunet,record-
      conversation,unshape}
    call                 call a GNUnet peer using conversation
    experiment           start an experiment
    init-gnunet          start GNUnet with the config file from the environment
    shutdown-gnunet      stop gnunet
    set-workers          store information about workers
    update-gnunet        recompile and reinstall gnunet if changes exist
    record-conversation
                         start conversation with --auto-accept and record to
                         output.wav as soon as the call is accepted
    unshape              remove all network shaping

optional arguments:-
  h, --help            show this help message and exit
```

Listing 11: Usage of the measurement worker script

C Output files

GNUnet component	Output location	Description
gnunet-conversation	conversation_rtt_ping.csv	CONVERSATION RTT values (ping method)
CONVERSATION service	conversation_rtt_call.csv	CONVERSATION RTT values (call method)
gnunet-helper-audio-record	conversation_encode_delay.csv	OPUS encode delay
gnunet-helper-audio-playback	conversation_decode_delay.csv	OPUS decode delay
gnunet-cadet	stdout	CADET RTT values
CADET service	cadet_encryption_delay.csv	double ratchet encryption delay
	cadet_decryption_delay.csv	double ratchet decryption delay
gnunet-core	stdout	CORE RTT values
CORE service	core_encryption_delay.csv	OTR encryption delay
	core_decryption_delay.csv	OTR decryption delay
gnunet-transport	stdout	TRANSPORT RTT values

Table 6: The measured delays are either printed to stdout or written to a CSV file (filename currently hardcoded); the delays are logged in microseconds, lost packets are logged as -1

D GNUnet command-line tools

```
gnunet-transport
Direct access to transport service.
Arguments mandatory for long options are also mandatory for short options.-
  a, --all                  print information for all peers (instead of only
                                connected peers)-
  b, --benchmark            measure how fast we are receiving data from all
                                peers (until CTRL-C)-
  c, --config=FILENAME      use configuration file FILENAME-
  D, --disconnect           disconnect from a peer
  -E, --echo activate echo mode-
  e, --events               provide information about all connects and
                                disconnect events (continuously)-
  h, --help                 print this help-
  i, --information          provide information about all current connections
                                (once)-
  L, --log=LOGLEVEL         configure logging to use LOGLEVEL-
  l, --logfile=FILENAME     configure logging to write logs to FILENAME-
  m, --monitor              provide information about all current connections
                                (continuously)
  -N, --number=NUMBER number of RTT measurements-
  n, --numeric              do not resolve hostnames-
  P, --plugins              monitor plugin sessions-
  p, --peer=PEER            peer identity
  -r, --measure-rtt measure rount-trip time by sending packets to an
                                echo-mode enabled peer-
  s, --send                 send data for benchmarking to the other peer
                                (until CTRL-C)-
  V, --verbose              be verbose-
  v, --version              print the version number
  -w, --timeout=SECONDS timeout for each RTT measurement
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/s/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

Listing 12: gnunet-transport command-line options, newly-implemented options in red

```
gnunet-core
Print information about connected peers.
Arguments mandatory for long options are also mandatory for short options.-
  c, --config=FILENAME      use configuration file FILENAME
  -e, --echo activate echo mode-
  h, --help                 print this help-
  L, --log=LOGLEVEL         configure logging to use LOGLEVEL-
  l, --logfile=FILENAME     configure logging to write logs to FILENAME-
  m, --monitor              provide information about all current connections
                                (continuously)
  -n, --count=COUNT number of RTT measurements
  -p, --peer=PEER peer identity
  -r, --measure-rtt measure round-trip time by sending packets to an
                                echo-mode enabled peer-
  v, --version              print the version number
  -w, --timeout=SECONDS timeout for each RTT measurement
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/s/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

Listing 13: gnunet-core command-line options, newly-implemented options in red

```
gnunet-cadet (OPTIONS | PEER_ID SHARED_SECRET)
Create tunnels and retrieve info about CADET's status.
Arguments mandatory for long options are also mandatory for short options.-
  C, --connection=CONNECTION_ID
                              Provide information about a particular connection-
  c, --config=FILENAME       use configuration file FILENAME-
  d, --dump                  Dump debug information to STDERR-
  e, --echo                  Activate echo mode-
  h, --help                  print this help-
  L, --log=LOGLEVEL          configure logging to use LOGLEVEL-
  l, --logfile=FILENAME      configure logging to write logs to FILENAME
  -n, --count=COUNT number of RTT measurements-
  o, --open-port=SHARED_SECRET
                              Listen for connections using a shared secret
                                 among sender and recipient-
  P, --peers                 Provide information about all peers-
  p, --peer=PEER_ID          Provide information about a patricular peer
  -r, --measure-rtt Active RTT measurements-
  T, --tunnels               Provide information about all tunnels-
  t, --tunnel=TUNNEL_ID      Provide information about a particular tunnel-
  v, --version               print the version number
  -w, --timeout=SECONDS timeout for each RTT measurement
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/s/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

Listing 14: gnunet-cadet command-line options, newly-implemented options in red

```
gnunet-conversation
Enables having a conversation with other GNUnet users.
Arguments mandatory for long options are also mandatory for short options.
  -a, --auto-accept automatically accepts all incoming calls (for
                                 measurement purposes)-
  c, --config=FILENAME       use configuration file FILENAME
  -E, --echo activate echo mode-
  e, --ego=NAME              sets the NAME of the ego to use for the phone
                                 (and name resolution)-
  h, --help                  print this help-
  L, --log=LOGLEVEL          configure logging to use LOGLEVEL-
  l, --logfile=FILENAME      configure logging to write logs to FILENAME
  -n, --count=COUNT number off RTT measurements-
  p, --phone=LINE            sets the LINE to use for the phone
  -r, --measure-rtt activate RTT measurement-
  v, --version               print the version number
  -w, --timeout=SECONDS timeout for each RTT measurement
Report bugs to gnunet-developers@gnu.org.
GNUnet home page: http://www.gnu.org/s/gnunet/
General help using GNU software: http://www.gnu.org/gethelp/
```

Listing 15: gnunet-conversation command-line options, newly-implemented options in red