BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2016

Bc. Martin Jaroš

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

DISTRIBUOVANÉ MULTIMEDIÁLNÍ SLUŽBY

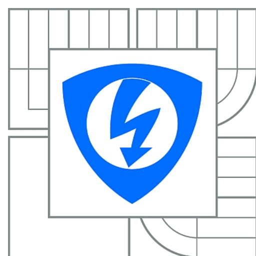
DISTRIBUTED MULTIMEDIA SERVICE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Martin Jaroš**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Petr Číka, Ph.D.**
SUPERVISOR

BRNO 2016



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Martin Jaroš	***ID:***	146847
Ročník:	2	***Akademický rok:***	2015/2016

NÁZEV TÉMATU:

Distribuované multimediální služby

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte komunikační protokol vhodný pro přenosy v reálném čase prostřednictvím peer-to-peer multimediálních sítí. Pro vyhledávání uživatelů v rámci sítě využijte distribuovanou hešovací tabulku. Navrhněte kryptografický systém pro ověření uživatelů a zabezpečení přenášeného obsahu. Metody implementujte a vytvořte aplikaci vhodnou pro distribuované streamování multimédiálního obsahu. Aplikaci doplňte o uživatelské rozhraní, analyzujte chování celého systému.

DOPORUČENÁ LITERATURA:

[1] GHODSI, Ali. Distributed k-ary System: Algorithms for Distributed Hash Tables. The Royal Institute of Technology. 2006.
[2] BERNSTEIN, D. Curve25519: New Diffie-Hellman speed records. Proceedings of PKC 2006.

Termín zadání:	1.2.2016	***Termín odevzdání:***	25.5.2016

Vedoucí práce: Ing. Petr Číka, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Diplomová práce se zabývá návrhem a realizací distribuované komunikační služby v reálném čase, jejím cílem je sestavit základní strukturu pro aplikace digitálního přenosu zvuku a videa v rámci klient-klient topologie sítě. Práce definuje komunikační protokol a popisuje jeho implementaci. Důraz je také kladen na zabezpečení pomocí moderních kryptografických metod. Realizace navazuje na stávající řešení přičemž využívá především svobodného softwaru.

KLÍČOVÁ SLOVA

Multimédia, komunikace v reálném čase, klient-klient

ABSTRACT

This work focuses on design and realization of a distributed real-time communication system, the primarily goal is creation of a generic framework for digital audio and video broadcasting within a peer-to-peer network. The work defines a communication protocol and describes its implementation. Security is also taken into account with usage of modern cryptography methods. Realization is built on top of existing solutions preferring free and open-source software.

KEYWORDS

Multimedia, real-time communications, peer-to-peer

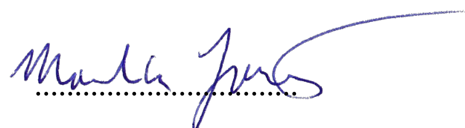
JAROŠ, M. *Distribuované multimediální služby*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 53 s. Vedoucí dimplomové práce Ing. Petr Číka, Ph.D..

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma Distribuované multimediální služby jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno 25.5.2016



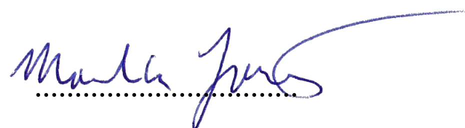
....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce Ing. Petrovi Číkovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

Brno 25.5.2016



....

(podpis autora)

research centre
sensor, information and communication systems

Faculty of Electrical Engineering
and Communication

Brno University of Technology
Technicka 12, CZ-61600 Brno, Czechia

http://www.six.feec.vutbr.cz

The research described in this diploma thesis has been done in laboratories supported by Sensor, Information and Communication Systems Research Centre (SIX) project; registration number CZ.1.05/2.1.00/03.0072 Operational Program Research and Development for Innovation (operační program Výzkum a vývoj pro inovace).

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

2007–13
OP Výzkum a vývoj pro inovace

Contents

1 Project overview **9**

2 Overlay network **11**

2.1 Lookup algorithm . . . 12

2.2 Protocol specification . . . 17

2.3 Protocol implementation . . . 21

3 Security model **25**

3.1 Cryptographic primitives . . . 26

3.2 Protocol mapping . . . 29

3.3 Security consideration . . . 30

4 Streaming framework **32**

4.1 Streaming protocol . . . 32

4.2 GStreamer integration . . . 36

5 Application **43**

Conclusion **51**

Abbreviations **52**

References **53**

List of Figures

1.1 Distributed topology model 9
2.1 DHT buckets example 11
2.2 Session negotiation 20
3.1 Key exchange and derivation 26
4.1 Pipeline construction 37
4.2 RTP session management 40
5.1 Main application window 43
5.2 Aliases editor 44
5.3 Automatic text completion 45
5.4 Status window 47
5.5 Network preferences 48
5.6 Audio preferences 49
5.7 Video preferences 50

List of Tables

2.1 State management of DHT nodes 14
2.2 Summary of DHT system parameters 17
3.1 Comparison of Curve25519 implementations 26
4.1 Payload types of the RTP profile 33
4.2 GStreamer hardware video acceleration support 39

1 Project overview

The main goal of this project is to design and implement an alternative model to the server-client based VoIP architecture, that is fully distributed between its clients and does not rely on centralized servers. Advantages of such model are simplified deployment, reduced maintenance costs, higher reliability and security. Currently widespread VoIP systems use a Session Initiation Protocol for session signaling directed by a centralized SIP server. The proposed system replaces this protocol with the implementation of a logical overlay network. The overlay network is a peer-to-peer network with a mesh topology that allows communication between its nodes independently on the underlaying physical network infrastructure.

Each node chooses a randomly generated private key. A globally unique value is derived from this key, that becomes the public node identifier. The overlay network and its protocol defined within this document provides a mechanism for maintaining a routing table which maps the node identifiers to the respective physical network addresses. The following diagram illustrates the proposed architecture.

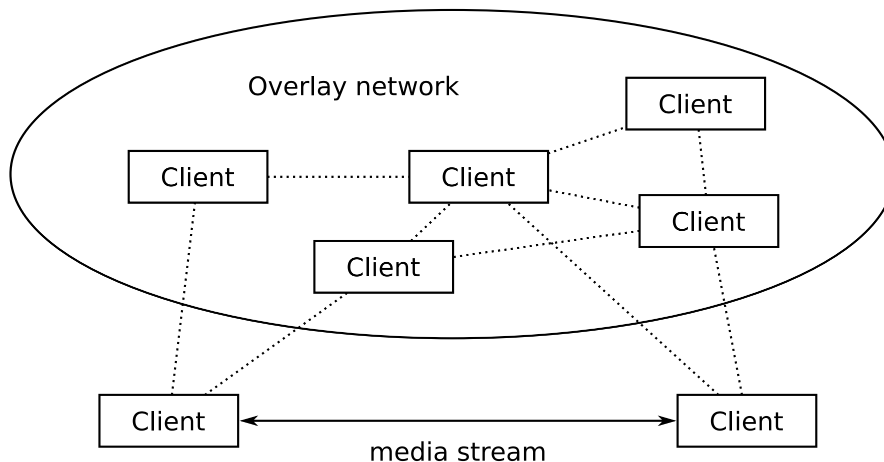


Figure 1.1: Distributed topology model

The overlay network is a logical topology built on top of the physical network, that allows communications between clients based on unique node identifiers. Any node connected to the network may create a connection to any other node by executing an ID lookup procedure that yields the physical address. The connection is tunneled through the physical network which allows multimedia transfer between the nodes. The same payload protocols as in the traditional VoIP model may be used thanks to the per-session tunneling.

This work is divided into four chapters; each chapter begins with a generic introduction to the specific topic and definition of requirements for this project. Designed solutions

are presented in consecutive sections.

The Overlay network (2) chapter describes the proposed distributed architecture. A specific lookup algorithm was designed and is presented together with its networking protocol. The protocol itself is designed from scratch for best fit of the proposed algorithm. The reference implementation is described in the last section of the chapter.

The Security model (3) chapter specifies the cryptographic suite. Individual primitives are selected and discussed within proposed constructions. Implementation of the security model is provided and analyzed. This work uses already existing and proved primitives, a mapping of the primitives to the proposed protocols is described.

The Streaming framework (4) chapter describes the media payload. The protocol mapping to existing standards is defined and the selected protocols are discussed. A real-time pipeline is proposed and integrated with existing software blocks. Specific audio and video codecs are selected and compared, specific audio and video backends are also discussed. An integration of the complete streaming layer is implemented, specific features as audio video synchronization and hardware acceleration are considered.

The Application (5) chapter documents the implemented application. A complete graphical user interface frontend is described within this chapter. All previous conclusions are integrated together into a single application. The application is designed to be a full featured implementation of a multimedia communication client for widespread use.

The implementation design is focused on the GNOME platform which makes it portable to most operating systems including Linux and Windows. The program uses exclusively free software and itself is free software under the GNU General Public License. The application is written in the C language using the native GLib / GObject framework for object-oriented high level approach. The advantage of using such framework is direct extendability and interoperability with high level languages such as Python, Vala, Java or C# while keeping the performance of plain C. There are also many libraries built on top of the same framework such as the GTK+ user interface library and the GStreamer multimedia library.

The user interface is provided in native English and localized Czech translation. The final product is distributed via GitHub [1].

Special care is taken about hardware acceleration support. The application runs on Intel x86, x86_64 and ARM AArch32 architectures, both SSE and NEON vector floating point units are considered. Dedicated audio and video accelerators are also taken into account.

Both POSIX socket and WinSocket networking APIs are supported. The cryptographic framework features state-of-the-art algorithms developed by Daniel Bernstein [2][[3][4] and published into the public domain.

2 Overlay network

The overlay network provides mapping of logical identifiers to physical network addresses, this mapping is shared between nodes with a distributed hash table (DHT) [5]. Each node has only a limited knowledge of the DHT and queries its neighboring nodes to find the requested target. There is a notation of logical distance of the nodes that serves as a metric during the lookup process, nodes closest to the target are recursively queried until the target is finally found. This metric is defined as a bitwise XOR of the node IDs, it is symmetrical and has the triangle property

$$x \oplus x \;=\; 0, \quad x \oplus y \;=\; y \oplus x, \quad (x \oplus y) + (y \oplus z) \;\geq\; x \oplus z. \tag{2.1}$$

A routing table is maintained by each participating node, it has form of a binary search tree based on the metric. The table is split into buckets of maximum size k, the n-th bucket contains up to k nodes which first n bits of their ID is same as the first n bits of the local node ID. Initially there is only one bucket in the routing table, which gets updated dynamically as the traffic flows. When the number of nodes hits k, the bucket is split into two and nodes are redistributed. This distribution is illustrated by the following diagram.

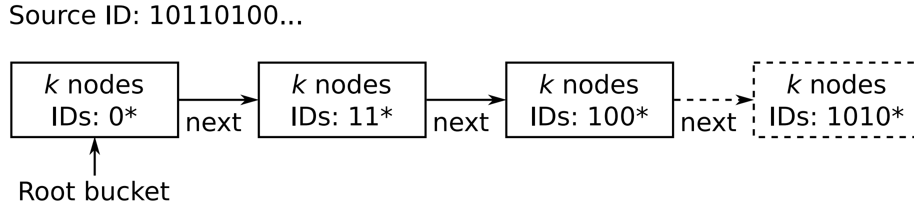


Figure 2.1: DHT buckets example

Total size of the routing table is based on the value of k which is system wide parameter

$$N \;=\; k \cdot \log_2 n - k \cdot \log_2 k, \quad \text{for } n > k \tag{2.2}$$

where n is a number of nodes within the system and N is a number of nodes in the routing table. The ID is of a finite size which limits the maximum number of nodes within the system and maximum routing table size. As there should be a significant margin to prevent hash table collisions a 20-byte ID was chosen for the protocol, however this value is application specific and may be changed if necessary. This gives a theoretical maximum number of nodes in the network of 2^{160} and with a k value of 16, the maximum number of nodes in the routing table of

$$16 \cdot \log_2 2^{160} - 16 \cdot \log_2 16 \;=\; 16\,(160 - 4) \;=\; 2496. \tag{2.3}$$

During lookup for a target node, the local node first examines its routing table to get k nodes closest to the target. Then it sends lookup request message to the closest node it knows about and caches the result, this step is repeated recursively until the target node is found or until the k closest nodes were queried. When a node receives the request it searches its own routing table for the target ID and send a lookup response message containing up to k closest nodes to the target. The local routing tables are also refreshed with each message to contain up-to-date information.

Each message sent should be assigned to a timer, when this timer expires or an error is received over the network, the queried node is considered dead and is removed from consideration. The lookup process continues as if the node responded with zero entries. There may be multiple messages send in parallel per lookup step to speed up the search by filling timeout gaps, the maximum number of parallel messages shall be limited by a concurrency parameter α. Higher α value reduces lookup latency but increases network bandwidth usage. The reference implementation of this project uses a concurrency parameter $\alpha = 4$, that is 4 request are sent per recursion step.

During startup the routing table needs to be pre-filled by a bootstrap process before an automated refresh may take place. The bootstrap process sends request to a network addresses specified by user to discover the initial nodes. These nodes are special and should have static public network addresses known to the user. In the case of a local area network the bootstrap process may use broadcast to automatically discover its neighbors.

2.1 Lookup algorithm

The routing table is represented as a linear doubly linked list of buckets defined as

```
struct bucket
    pointer to previous bucket
    pointer to next bucket
    array of nodes
```

where the bucket is a single list item holding up to k nodes in its array. The node is represented as

```
struct node
    node ID
    network address
    network port
    alive flag
    last seen timestamp
```

When a message is received from remote peer the routing table is updated with sender information:

- The node ID taken from message body
- The network address and port of the message origin
- The current time (using monotonic clock)

The system may be configured to use either IP version 4 or IP version 6 address. A source metric is calculated as bitwise XOR of the remote node ID and the local node ID. The following algorithm is used to find an actual bucket within the routing table in which the node belongs

```
nbits = 0
bucket = first bucket

while ( bucket.next not null ) and not
      ( metric[nbits div 8] & (0x80 >> (nbits mod 8)) )

    nbits = nbits + 1
    bucket = bucket.next
```

The selected bucket is then searched for the node ID. If found the node is just updated and flagged as "alive"; otherwise the node is inserted into the array (if there are less then k elements). If the array is already full, the bucket is either split into two or the node replaces another node or the node is discarded. Splitting of tables is done if the current bucket is also the last bucket in the linked list, a new bucket is allocated and inserted as the last list item and nodes are redistributed based on the previous algorithm.

Empirical tests of distributed networks [6] have shown that the probability of nodes leaving the network decreases with time, during which they stayed alive in within the network. Therefore it is always preferable to retain alive nodes in the routing table, even if a more recent node was found as replacement. If there is a node in the bucked not flagged as "alive", it is replaced by the new node; otherwise the new node is discarded. The node is flagged as potentially not "alive" when it fails to respond in $t_{latency}$ to a sent request. The node is retained in the routing table, until it is replaced by another node. The latency parameter $t_{latency}$ defines the maximum time to wait for a node to respond and should be set to the highest expected network latency. The suggested value is 1 second.

If the linger time of a node is higher that t_{linger}, the node is considered "dead" and skipped during the lookup algorithm. The linger time of a node is the difference between

current time and the node timestamp, this is the time since the last received message from that particular node. Timestamps of nodes are updated with each received message. Time is always measured with monotonic system clock, this clock starts from zero at boot time and runs uniformly. System clock adjustments may modify monotonic clock rate to fix drifts introduced by oscillator imperfections, but will never directly change the time value which would otherwise produce time discontinuities. The following table summarizes discussed node states

Last query	Last response	Search action	Update action
responded	$< t_{linger}$	query	keep
responded	$> t_{linger}$	query	keep
timed-out	$< t_{linger}$	query	replace
timed-out	$> t_{linger}$	skip	replace

Table 2.1: State management of DHT nodes

Servicing of lookup requests uses a similar but more generic form of the algorithm that iterates buckets in both directions to get k nodes closest to the destination node. The destination metric is calculated as bitwise XOR of the destination node ID and the local node ID.

```
dir = 1
nbits = 0
bucket = first bucket
result = empty array

while length(result) < K_parameter

    if not ( metric[nbits div 8] & (0x80 >> (nbits mod 8)) ) == not dir
        for node in bucket.array
            if node.is_alive or (current_time - node.timestamp < T_linger)
                append result with node

    if dir == 1
        if bucket.next not null
            nbits = nbits + 1
            bucket = bucket.next
        else
```

```
            dir = 0
    else
        if bucket.prev not null
            nbits = nbits + 1
            bucket = bucket.prev
        else
            break
```

Running this algorithm will therefore yield an array of nodes from the routing table that are logically closest to the requested ID. In order to lookup target node, one would remotely call this function recursively until the target node is actually found. There may be multiple parallel lookups happening at once; each lookup is distinguished by its destination ID. The lookup itself is handled in parallel so it makes no sense to run more than one lookup with the same destination ID.

The state of a lookup process is defined by the following structure

```
struct lookup
    destination ID
    query cache
```

where the query cache is a sequence of queries sorted based on the metric against destination node ID, lowest value first. This sequence should be implemented with a balanced binary tree for high performance and scalability. Multiple lookup states should be stored in a hash table with destination ID used as a key. Since the IDs themselves are random and the XOR metric sorts by most significant bits first, the last n bits of the destination ID may be used directly as the hash key, where n is the hash key size. Therefore when a lookup response message is received the implementation can quickly find the corresponding lookup state.

The structure of a single query is

```
struct query
    metric
    network address
    network port
    finished flag
    alive flag
    timeout
```

Metric is the XOR of the node ID and the destination ID. The metric is used to sort queries within the sequence. The implementation starts by performing a lookup locally

on its own routing table and continues by dispatching cashed queries. The number of concurrent messages sent is limited by the concurrency parameter α.

When a response is received, the query cache is updated with new queries created from the nodes contained in the response. Network address and port is copied and the metric is calculated as a bitwise XOR of the node ID and lookup destination ID. The query is flagged as "alive" if a response from that ID is received or as not "alive" if the query times out. In either case the finished flag is set.

This process is repeated until a address for the destination ID is found or until all of the k closest nodes to the target have responded. That is the first k items in the query cache are flagged as finished and "alive", in this case the lookup terminates with and error. Note that if a lookup to randomly chosen ID is executed, it is unlikely that the target node exist, however at least k closest nodes to the chosen ID will be queried. This is useful as a refresh feature because a bucket containing the selected ID is updated during the process.

Buckets are periodically refreshed based on the $t_{refresh}$ parameter by executing a random lookup with randomly generated destination ID. The generator should follow the logarithmic distribution of buckets in the routing table. The following algorithm is proposed:

```
let n = uniform 0 to N
let r = uniform 0 to 2^(N - n) - 1
refreshID = ownID & (~0 << n) | r
```

where `N` is the number of buckets in the routing table. These periodic lookups will update routing tables across the whole network and are necessary for detecting nodes leaving the network. The optimal value of $t_{refresh}$ depends on size of the routing table, the suggested value is 1 minute.

A bootstrap process is used for initial setup of the routing table. This is a done with a normal refresh lookup, whose destination ID is set to the source ID of the node performing the bootstrap. As the local routing table is empty during the bootstrap, the initial lookup state does not contain any queries. Instead a user supplied list of static IP addresses is used to send the initial requests and a standalone timer is created to measure the $t_{latency}$ timeout until the first response is received. Upon reception of the first response the lookup may continue as usual, the k nodes closest to the source ID will be discovered. From this state the periodic refresh lookups can take over and keep the routing table up-to-date. Premature expiration of the bootstrap timer without any response should be reported to the user, in this case we failed to join the distributed network and the bootstrap process must be repeated.

This is a summary of the defined system parameters and their suggested values

Parameter	Description	Value
k	bucket size	16
α	concurrency	3
$t_{latency}$	query timeout	1 second
$t_{refresh}$	refresh period	1 minute
t_{linger}	node timeout	1 hour

Table 2.2: Summary of DHT system parameters

2.2 Protocol specification

The communication protocol is built on top of UDP/IP networking stack and uses remote procedure call like messages. The goals of the protocol are:

- Distributed hash table lookup
- Session key exchange
- NAT traversal

Each message of the protocol is identified by the first byte. The 4 most significant bits specify the protocol type while the 4 least significant bits specify the message type. RTP is used for the underlaying payload, which sets its first two bits to the RTP version. In order to easily distinguish control messages from payload the first two bits are set to one (11_b), while RTP uses (10_b). The remaining bits are used to separate IPv4 and IPv6 addressing modes

- $C0_{16}$ - IPv4 addressing mode type mask
- $D0_{16}$ - IPv6 addressing mode type mask

There are 4 message types defined

- 0 - lookup request
- 1 - lookup response
- 2 - connection request
- 3 - connection response

The lookup request message has the following structure:

```
Message type | Source ID | Destination ID
```

Message type is $C0_{16}$ for IPv4 or $D0_{16}$ for IPv6.

Source ID is an 20-byte identification number of the message sender.

Destination ID is an 20-byte identification number of the lookup target.

The lookup response additionally specifies target nodes:

```
Message type | Source ID | Destination ID | Node 1 | Node 2 | Node N
```

Message type is $C1_{16}$ for IPv4 or $D1_{16}$ for IPv6.

The actual number of nodes may be from 1 to k and is determined by the message length. Each of the nodes has the following inner structure:

```
Node ID | Port number | IP address
```

Node ID is an 20-byte identification number of the node.

Port number is a 2-byte port number in network byte order.

IP address is either an 4-byte IPv4 address or an 16-byte IPv6 address in network byte order.

The address family of the IP address must much the protocol type of the message. It is possible to specify an IPv4 address as an IPv6 mapped address with a $FFFF_{16}$ prefix. This will cause IPv6 message to be tunneled over IPv4 connection, which still requires support of the IPv6 addressing mode on both endpoints. The advantage of the pure IPv4 addressing mode are smaller messages, however it is perfectly possible to use the IPv6 addressing mode ($D0_{16}$ mask) over IPv4 sockets. The implementation may accept either of the addressing modes or both. If a particular address family is not supported by the socket, the message should be discarded. In particular the IPv4-only node cannot work with IPv6 addresses which are not IPv4-mapped addresses for obvious reasons.

Lookup messages are always sent from a single bound but unconnected datagram socket. This makes full-cone NAT traversal possible as address translation bindings should remain constant for the lifetime of the socket. Foreign nodes will see the source node by its public address:port tuple and propagate this information through the distributed hash table. In the case of port restricted or address restricted NAT a port forwarding must be set up to allow incoming requests get through the NAT. Given the nature of the protocol it is not possible to extend NAT traversal to a symmetrical NAT.

The connection request message has the following structure:

```
Message type | Source ID | Nonce | Application profile
```

Message type is $C2_{16}$ for IPv4 or $D2_{16}$ for IPv6.

Source ID is an 20-byte identification number of the message sender.

Nonce is a 32-byte unique number used to identify this connection session.

Application profile (optional) is an ASCII string identifying an application profile requested for this connection. The application profile name is a hint for the receiver to enable arbitrary extensions of the stream protocol. For the Real-time Transport Protocol audio video profile as defined in the Streaming framework (4) chapter a literal `"rtp-avp-1"` string is used.

There may be multiple concurrent connections. Each connection is uniquely identified by its *nonce*, which is a randomly generated number. The connection request is sent from the same socket as used for lookup messages. The destination address is the result of the lookup process. There is a unique socket pair created for each connection. Each peer participating in the connection creates a new socket, binds it to a random source port and sends a connection response to the distributed peer address. As all messages are send to the statically bound port, the dynamically created ports are forwarded through the NAT. The connection response has the following structure:

```
Message type | Public key | Source nonce | Peer nonce
```

Message type is $C3_{16}$ for IPv4 or $D3_{16}$ for IPv6.

Public key is a 32-byte public key of the sender as defined in Security model (3) chapter.

Source nonce is a 32-byte nonce generated by message sender.

Peer nonce is a 32-byte nonce matching the peer's request.

After the exchange, both endpoints should have received the address of the peer's socket and the sockets can be connected together. Each endpoint also knows its peer public key a nonce which is used in key derivation according to the Security model (3) chapter. The routing table must not be updated with per-connection socket addresses, that is the source address of the connection response message. The routing table gets updated from the initial connection request message.

The summary of the complete session negotiation is:

1. Send connection request to the destination address discovered through DHT lookup
2. The peer allocates new socket and sends a connection response
3. Create a new socket and send a connection response

These messages are matched be the nonces and resembles a 3-way handshake. The implementation should maintain a per-connection states in a hash table keyed by the nonce. The nonce or any part of it may be directly used as a hash key. When a connection request message is received a new connection state is created containing:

- nonce
- peer ID
- peer nonce
- socket

When a connection response message is received the implementation should get the connection state from the table based on the peer nonce field of the message. The public key must be verified against the peer ID according to the Security model (3). If the implementation did not yet sent a response it should send one by filling its public key and nonce and copying the peer nonce.

Each message is assigned a timeout, the connection fails with error if the peer does not respond in time. In this case the connection should be repeated with new nonces. If the handshake is successfully finalized, further payload may be sent freely on the resultant socket pair. The uniqueness of the nonce is critical for security and its generation is described in further detail in Security consideration (3.3) section.

The following diagram illustrates the complete dataflow.

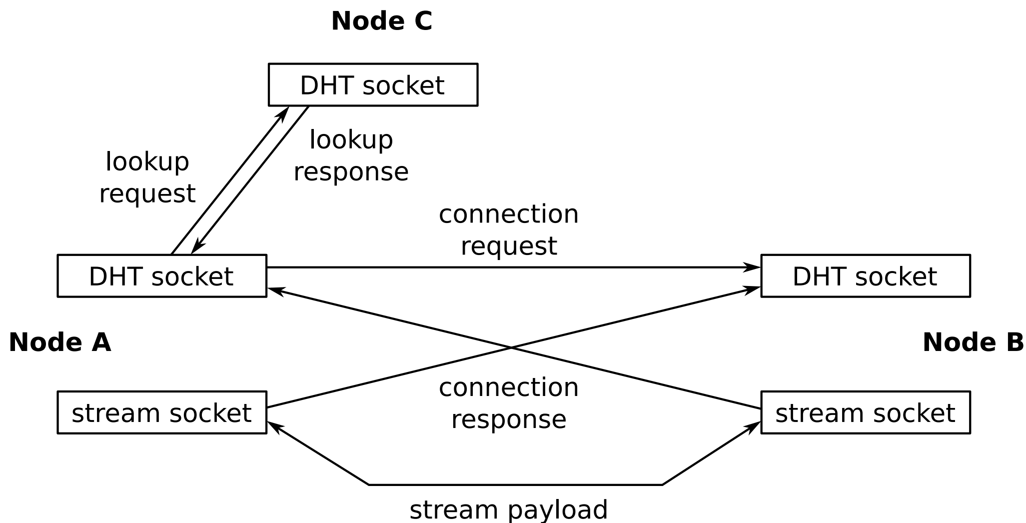


Figure 2.2: Session negotiation

The protocol of the payload is defined by the application profile negotiated during the connection handshake. The application profile itself is not authenticated nor encrypted and so shall not contain any sensitive information. The payload and its profile is

authenticated with a first packet being transfered by a profile-defined authentication scheme.

As an alternative to the RTP audio video profile that is discussed in detail further in the work, lets define a simple text based messaging protocol for a basic chatting service. Text messages are different in nature than the audio or video payload as they do not form a continuous stream. A text transfer protocol requires confirmation of message reception and re-transferring of missing packets. Lets define a text based application profile identified by a literal string `"text-utf8"`. The application message has the following form

ACK | Sequence number | Payload | Authentication tag

ACK is an acknowledgement bit flag. This flag is zero for normal messages and one for confirmation messages.

Sequence number is a 31-bit sequential number of the message. The first message sent has zero sequence number and the value is linearly incremented with each message. Confirmation messages uses the sequence number of the message being confirmed.

Payload is the encrypted, UTF-8 encoded text to be transferred. The payload has variable length deducted from the datagram size. A single message length is limited to the socket MTU, larger messages are split into multiple packets. The payload size may be zero, this is particularly useful for keep-alive messages. Confirmation messages has no payload (zero length), their payload is defined by the message they confirm by a sequence number.

Authentication tag is a cipher-defined MAC. This field is 16-bytes long for the chacha20-poly1305 encryption scheme defined in the Cryptographic primitives (3.1) chapter. Messages are encrypted using the same primitives as for the RTP, the nonce is the concatenation of the ACK bit and the sequence number forming a 32-bit value.

The receiver sends a confirmation for each received message. The confirmation messages are formed by setting the ACK bit to one, stripping the payload and recalculating the authentication tag using the receivers key and the new nonce value. The sender caches the messages and awaits their confirmation, timed-out messages are resent. The connection is kept open until a per-connection timeout, the timeout is prolonged with periodic keep-alive messages with no payload.

2.3 Protocol implementation

The implementation uses an event driven model, its API is asynchronous and non-blocking. For the sake of clarity the following text will annotate asynchronous results with `->` symbol. These results may be delivered via an asynchronous callback or by other facility provided by the target language.

The API provides a class `Client` and two global constants:

- `ID_LENGTH` - Length of the ID string
- `KEY_SIZE` - Size of the private key in bytes

IDs are exposed in human readable form as Base64 encoded strings, a 20-byte ID is 28 characters long. Keys are exposed as raw 32-byte objects, the application is supposed to read the key from a binary file. The `Client` class has the following constructor:

```
Client(family: SocketFamily, port: int, key: bytes)
```

where `family` is either `SocketFamily::IPV4` or `SocketFamily::IPV6`, `port` is the local port number to which the client should bind to and `key` is a 32-byte private key. The key is an optional parameter; the implementation will generate a random key if one was not specified. If the port is set to zero a random port will be selected and bound. The `Client` class needs to be initialized with the method

```
Client.bootstrap(host: string, port: int)
    -> [result: bool]
```

where `host` is either a host name or address and `port` is the destination port number. This method bootstraps the client by sending a request to the specified host. The host name is resolved via a domain name service to the selected network family addresses. This method may be called repeatedly, until at least one peer is found. The bootstrap process may fail, the result is delivered asynchronously. Successful invocation yields `true`, possible errors are

- `HOST_UNREACHABLE` - The host cannot be reached
- `TIMED_OUT` - The request timed out

The client will automatically take over with its own internal periodic refreshing after the bootstrapping is finished. The application may then proceed with the method

```
Client.lookup(id: string)
    -> [addr: SocketAddress]
```

where `id` is the destination ID. This method finds host by ID and yields its network address to which the application may connect via

```
Client.connect(id: string, addr: SocketAddress)
    -> [sock: Socket, peer: SocketAddress, enckey: bytes, deckey: bytes]
```

where `id` is the target node ID, `addr` is the target node address, `sock` is a per-connection socket, `peer` is an address of the per-connection socket of the peer, `enckey` is a derived encryption key, `deckey` is a derived decryption key. The keys are symmetric and unique for each connection, the encryption key of one peer matches the decryption key of the other peer and vice versa. The application may request any number of connection, each connection will have its own socket and network address. Note that the peer address is unique and not the same as the originally provided address.

Both methods may work in parallel and asynchronously queues the operations. Errors either raises synchronous exceptions or are delivered asynchronously

- `HOST_NOT_FOUND` - Lookup for a host with the specified ID failed
- `HOST_UNREACHABLE` - The host address cannot be reached
- `CONNECTION_REFUSED` - The connection was rejected
- `TIMED_OUT` - The connection timed out

The application listens for incoming connections

```
Client.listen(accept: function<[id: string], [result: bool]>)
    -> [sock: Socket, peer: SocketAddress, enckey: bytes, deckey: bytes]
```

where `accept` is a function that takes an ID and returns `true` if the connection should be accepted or `false` if the connection should be rejected. This method yields a new connection for each accepted request. The callback function is invoked whenever a connection request is received and it is up to the user to decide which remote IDs he trusts and accepts. The yielded result is the same as for the `connect` method.

The `Client` class exposes the following properties:

`id` string (read-only)

> Client ID, 20 bytes, Base64 encoded. This is the format used by all public methods. It is readable by humans, usable as string and with minimal overhead. The application should present this ID to the user.

`key` bytes (read-only)

> 32-byte private key. This property may be used to retrieve a randomly generated key. The key may be saved to a file and later loaded at startup.

`key-size` int (read/write)

> Size of the derived keys passed via `connect` and `listen` methods. The application may set any size as needed, 32 bytes by default.

Additional statistics are gathered and published as

`peers` Number of alive peers the client knows about

`last-seen` Time of the last received message

`packets-received` Total number of received packets

`packets-sent` Total number of sent packets

`bytes-received` Total number of received bytes

`bytes-sent` Total number of sent bytes

The implementation is internally bound to an event loop. The event loop should be integrated with the application to be executed within a single thread. The computations are mostly reactive, IO related and reentrant access would produce unnecessary overhead without considerable performance advantage. Hereby the API can be directly used in a GUI environment and the program can run a single event loop. There are two events for which the implementation needs a monitor:

- socket I/O, only a single socket is monitored
- timer expiration, multiple timers are needed

The private structure of the `Client` class is

`bucket_list` Doubly-linked list of buckets

A bucket is a vector of nodes updated with according to previously discussed algorithms.

`lookup_table` A hash table of lookup states

Each destination ID has its own lookup state. The hash table allows to quickly fetch the right object based on the destination ID of a received message.

`connection_table` A hash table of connection states

Each connection maintains a unique state. The hash table allows fast matching of peer's nonce to client's nonce.

The client runs a state machine servicing I/O events coming from the main event loop. Messages are dispatched according to the specific hash table and results are forwarded.

3 Security model

Security is a key aspect of any public network. The security model should define a common set of rules for all users, be transparent, easy to understand and implement, but hard to breach. This work focuses on using modern cryptographic primitives with security margin that scales well into the future.

The goals for the security model are:

- unique, uniformly distributed and unforgeable user identifiers
- public key exchange
- authentication and confidentiality of user payload

Each user shall own a private key d, this is a random number generated and securely stored by the user for its lifetime. A public key is computed from this number with the use of elliptic curve cryptography as a scalar multiplication with base point

$$Q = d\,G \tag{3.1}$$

where Q is the public key and G is the curve base point. The unique user identifier is calculated from this public key with a one-way pseudo-random hash function and published via DHT. It identifies the user within the overlay network and authenticates its public key.

The public key exchange resembles a Diffie-Hellman type algorithm using elliptic curves and static keys. Let Alice key pair be (Q_A, d_A) and Bob key pair be (Q_B, d_B). Both users exchange their public key keys together with some randomly generated nonces. Alice and Bob computes

$$(x_k,\ y_k) = d_A\,Q_B, \tag{3.2}$$

$$(x_k,\ y_k) = d_B\,Q_A \tag{3.3}$$

where x_k is the derived shared secret. As the private and public keys are constant (for the two parties), the shared secret is also constant and may be cached for future uses. Each party then derives encryption keys from the shared keying material, for Alice this would be

$$k_{rx} = \mathrm{KDF}(x_k,\ n_B \mid n_A), \tag{3.4}$$

$$k_{tx} = \mathrm{KDF}(x_k,\ n_A \mid n_B) \tag{3.5}$$

where k_{rx} is a receive key, k_{tx} is a transmit key, n_A is Alice's nonce, n_B is Bob's nonce and KDF denotes a cipher suite specific key derivation function. The transmit key is used to encrypt and authenticate the payload before sending. The receive key is used to verify and decrypt the payload after reception. Peers are authenticated with the first encrypted message being transfered. The following diagram illustrates the key exchange and derivation scheme.

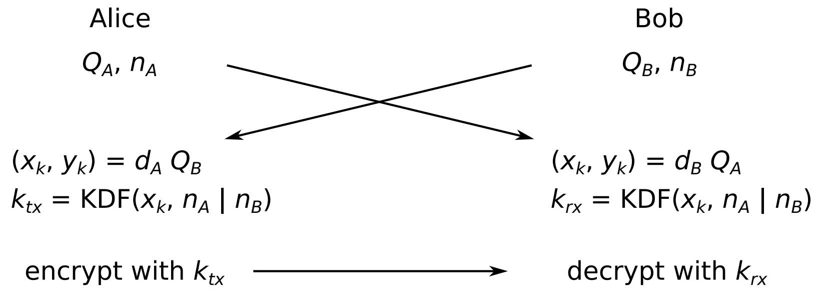


Figure 3.1: Key exchange and derivation

3.1 Cryptographic primitives

The **Curve25519** [2] was chosen for the elliptic curve Diffie-Hellman key exchange. This is a very fast modern elliptic curve developed by Daniel Bernstein, it is not encumbered by any patents and is without any known vulnerabilities or backdoor suspicions. It is a variant of Montgomery curve defined as

$$y^2 = x^3 + 486662x^2 + x \tag{3.6}$$

over the prime field defined by the prime number $2^{255} - 19$ and the base point 9. There are multiple high speed reference implementations provided, the design of the curve itself was optimized for speed efficiency. An optimized implementation for x86, 64-bit C source and generic C source code are available according to the following table.

Implementation	Platform	Author	32-bit speed	64-bit speed
curve25519-ref	x86 32-bit	D. J. Bernstein	265μs	—
curve25519-donna-c64	64-bit C	A. Langley	—	215μs
curve25591-donna	Portable C	A. Langley	2179μs	610μs

Table 3.1: Comparison of Curve25519 implementations

The core function takes a 32-byte integer and a 32-byte group element to compute a 32-byte group element. The public key is computed as

$$Q \;=\; \text{Curve25519}(d,\ 9) \tag{3.7}$$

while the shared secret is

$$x_k \;=\; \text{Curve25519}(d_A,\ Q_B). \tag{3.8}$$

The design of the curve allows using any 32-byte string as a public key without validation and is resilient to random number generator deficiencies. Therefore the implementation is simple and does not have worry about some common pitfalls and side channel attack. Note that the shared secret computation may fail if the provided public key is invalid (has smaller order). In this situation the result has zero value and the implementation must handle this condition as error.

The **BLAKE2b** [7] hash function was chosen for hashing user identifiers and key derivation. The requirements for the hash function given by the previously defined security model are collision resistance and second preimage resistance. BLAKE was one of the candidates for the SHA-3 competition and is a faster, more efficient yet highly secure alternative to the SHA-3 finalist, Keccak. The BLAKE2 algorithm is an improved version of BLAKE optimized for higher speed with slightly reduced number of rounds. It is a pseudo-random function (PRF) that computes an arbitrary long secure hash from arbitrary long input. The BLAKE2 core is a ChaCha20, the same core that is used for authenticated encryption described below. BLAKE2 additionally supporting keying implicitly making it a HMAC algorithm. The function is highly customizable, the reference implementation in C language comes in two flavors:

- BLAKE2b optimized for 64-bit systems and SIMD accelerated platforms (such as ARM/NEON), producing 1 to 64 bytes output in 12 rounds
- BLAKE2s optimized for 8- to 32-bit platforms, producing 1 to 32 bytes output in 10 rounds

The BLAKE2b implementation was chosen, it is faster then widely used but broken MD5 and SHA-1 yet with much higher security comparable to SHA-3. For ID computation a 160-bit variation of BLAKE2b was chosen as an analogue to the SHA-1 function:

$$ID \;=\; \text{blake2b160}(Q) \tag{3.9}$$

where Q is a 32-byte input string (the public key) and ID is a 20-byte result. For key derivation a 256-bit keyed variation is used, this is an analogue to HMAC-SHA256:

$$\mathrm{KDF}(x_k,\ n):\ k\ =\ \mathrm{blake2b256}(n,\ x_k) \tag{3.10}$$

where n is a 64-byte input string (the combined nonce), x_k is a 32-byte key (the shared secret) and k is a 32-byte result (the secret key).

The **ChaCha20** family [4] of stream ciphers was chosen for the authenticated encryption. ChaCha20 is an improved variant of a Salsa20 cipher family [3] with increased diffusion per round. The ChaCha20 core function maps a 64-byte input to a 64-byte output using add-rotate-xor operations. In streaming operation ChaCha20 generates 64-byte pseudo random blocks

$$B_i\ =\ \mathrm{chacha20}(n,\ i,\ k) \tag{3.11}$$

where n is a 12-byte (96-bit) per-message unique nonce, i is a 4-byte (32-bit) stream position and k is a 32-byte secret key. This limits the cipher stream length available for single message to 274 gigabytes which is well enough for practical network protocols.

The reference implementations come with support for SSE/AVX and NEON accelerators and are generally faster then the traditional substitution-permutation network based ciphers such as AES (with a notable exception of AES-NI acceleration). The latest cryptanalysis were able to break the first 8 rounds of Salsa20 core out of 20, thus providing reasonable security margin.

The authenticated encryption with additional data (AEAD) construction is based on [8]. The Poly1305 authenticator takes the first block of the ChaCha20 output stream as a one-time key, the rest of the output stream is XOR-ed with the data to be encrypted. The authenticated portion of the message is then passed to the authenticator to produce a 16-byte message authentication code (MAC)

$$c_i\ =\ m_i\ \oplus\ B_{i+1}, \tag{3.12}$$

$$MAC\ =\ \mathrm{poly1305}(a\ |\ m,\ B_0) \tag{3.13}$$

where a_i are authenticated only data and m_i are authenticated and encrypted data.

During the decryption process the MAC is computed and verified first, then (if valid) the encrypted portion is XOR-ed again with the following key-stream blocks to get the plain data. A single secret key is used in a single secure session, multiple protocols should divide the nonce number space (2^{64}), the implementation must assure that the nonce is always unique within the session.

The `libsodium` [9] open-source cryptographic library was chosen for the implementation, it features all primitives selected for this project in a portable way.

3.2 Protocol mapping

There are two protocol mappings proposed for this project:

- SRTP/SRTCP according to RFC [10]
- modified SRTP/SRTCP to use chacha20/poly1305 encryption

The RFC compliant encryption for RTP may be used, with a master key set to the results of the key derivation process. Size of the key depends on the selected cipher mode and must be properly configured, the proposed KDF is well capable of variable key derivation. However this would require using an external library and does not provide high enough security margin in comparison to the proposed system.

The proposed extension to the SRTP/SRTCP protocol uses the previously established authenticated encryption scheme. This extension is fully compatible with the standard and uses the same packet structure. The MKI is omitted and the authentication tag is set to the 16-byte MAC. The 96-bit nonce for AEAD construction is computed in the following way

$$n \;=\; 2^{64} \cdot ssrc \;+\; i \tag{3.14}$$

where $ssrc$ is a 32-bit synchronization source identifier and i is a 64-bit packet index.

The usage of $ssrc$ allows multiplexing of multiple RTP streams within one cryptographic context, as each stream has unique $ssrc$ value. RTP and RTCP payload can also be multiplexed together, the separation is done with different packet index range. For RTCP the packet index is explicitly included in the packet. The packet index starts at zero and is linearly incremented with each packet sent by the transmitter.

For RTP the packet index is calculated using the 32-bit rollover counter ROC, based on the SRTP specification, but shifted by a constant offset according to the following formula

$$i \;=\; 2^{31} + 2^{16} \cdot ROC + seq \tag{3.15}$$

where seq is the sequence number; refer to the Streaming framework (4) chapter for detailed packet structure. The sender should initialize the ROC to zero, and increment the value every time the sequence number wraps around to zero. On the receiver side, the ROC value is determined by the following algorithm

```
if packet.is_first
    prev_seq = packet.seq
```

```
    prev_roc = 0

roc = prev_roc
if prev_seq < 0x8000
    if prev_seq + 0x8000 < packet.seq
        roc = prev_roc - 1
else
    if prev_seq - 0x8000 > packet.seq
        roc = prev_roc + 1

if packet.decrypt(roc)
    prev_seq = packet.seq
    prev_roc = roc
```

The whole RTP/RTCP packet is authenticated but only its payload is encrypted, the SRTP packet structure is

```
RTP header | encrypted payload | MAC
```

SRTCP uses the explicit packet index, the packet structure is

```
RTCP header | encrypted reports | E | index | MAC
```

where E is a 1-bit encryption flag (1) and the *index* is a 31-bit integer.

The index number space is split from 0 to $7FFF\ FFFF_{16}$ for RTCP and from $8000\ 0000_{16}$ to $1\ 0000\ FFFF\ FFFF_{16}$ for RTP. The remaining number space of $1\ 0001\ 0000\ 0000_{16}$ up to $2^{64} - 1$ is left reserved for possible future extension. The session is limited to 2^{31} RTCP packets and up to 2^{48} RTP packets, when this threshold is reached the session must be dropped and renegotiated.

3.3 Security consideration

To assume an identity of another user, the attacker would have to either discover user's private key or find a second preimage to the ID. However even if the preimage is found, the attacker would still need to solve the discrete logarithm problem for a private key, assuming that the preimage is already a valid curve point. The same rule applies to random collisions which may (unlikely) naturally occur, in this situation the colliding IDs will fail to authenticate with no recovery other then regenerating new identity. Users are authenticated upon reception of the first encrypted message, that is after the initial key exchange.

The session security is primarily based on the nonce uniqueness as the shared secret is expected to be hard to crack. This would not be easy in an environment with a limited entropy pool. For this case the following scheme is proposed to generate a pseudo-random non-repeating value N_n:

$$N_n = \left(g^{f(n)} \mod p\right) \oplus c \tag{3.16}$$

where p is a prime number, g is a generator for $\mathbb{Z}_p$ and $f(n)$ is a linear congruential generator

$$f(n+1) \equiv a \cdot f(n) + b \mod m \tag{3.17}$$

with the values of a, b, c, m and f_0 being randomly seeded.

Each user has to decide which peers to trust, it is possible to create a PKI system by introducing a certification authority which signs trusted users public keys. These signatures may be appended to the public keys and verified by peers. Curve25519 cannot be used directly for ECDSA, however the curve points may be transformed from ed25519 twisted Edwards curve with

$$x_{montgomery} = \frac{1 + y_{edwards}}{1 - y_{edwards}}. \tag{3.18}$$

Certificates are issued to the users by a Certificate Authority. The connection response message is extended with a certificate chain

```
Connection response | Certificate chain
```

Certificate chain is the binary X.509 user's certificate followed by issuer's certificates up to the root certificate. The receiver first ensures that the peer's public key does match the top level certificate, then it verifies the certificate chain against a list of trusted certificates. This proposal is fully compatible with the previously described protocol and may be used within the same network.

4 Streaming framework

The streaming framework defines a payload transfer on top of the socket connection described in Overlay network (2) chapter. There may be multiple elementary streams multiplexed together, such as an audio stream and a video stream. The payload format must support inter-stream identification and synchronization. There are multiple codecs that needs to be distinguished, the streams should be self-descriptive as there is no session initiation or description layer. All signaling has to be done in-band, a simple call management is summarized by the following list

1. Caller creates a connection and starts transmitting an audio stream
2. Callee notifies the user and starts transmitting a ringing tone
3. User accepts the call, ringing tone is replaced by the actual audio stream
4. (optionally) A video stream is started by the peers
5. User hangs-up the call, the stream is terminated via a timeout

The ringing / ring-back tone is a generated 425 Hz sine wave with 1 second on-time and 4 seconds off-time, which overrides the audio stream of the callee until the user accepts the call. This works as a simple audio cue to the calling party.

The protocol selected for the streaming framework is the RTP [11]. RTP header specifies a 7-bit payload type and a 32-bit unique identifier for each stream. The unique identifiers are primarily used for encryption to partition the nonce number space as described in chapter 3.2. This leaves only the 7-bit payload type for codec information. The RTP specification is designed for dynamic payload description, however this application does not need a huge codec pool, so a fixed table is defined for the dynamic payload types in chapter 4.1.

4.1 Streaming protocol

The RTP packet header has the following format:

```
Version | P | X | CC | M | PT | Sequence number | Timestamp | SSRC
```

Version is a 2-bit value indicating the protocol version (2).

P and *X* are padding and extension bit flags (not used).

M is a marker bit which is set for the first packet of a video frame, the remaining packets of the frame has the marker bit cleaned. As the video frame size may be higher than the packet MTU a single frame is fragmented into multiple packets with distinct

sequence numbers but a common timestamp. The marker bit indicates start of a new video frame, for audio this bit is not used.

CC is a 4-bit number of CSRC identifiers (not used).

PT is a 7-bit payload type defined in a table below.

Sequence number is a 16-bit number used to reorder packets and detect losses.

Timestamp is a 32-bit clock value used for synchronization.

SSRC is a 32-bit clock identifier used for synchronization.

The RTP header is followed by an encrypted payload, the packet is appended with an authentication tag. The implementation may use a RTCP signaling for synchronization, elementary streams (audio, video, RTCP) are multiplexed into a single physical stream. The following definitions assume RTCP usage, an lightweight alternative without the RTCP stream is described and the end of this chapter. The RTP and RTCP streams are multiplexed according to [12]. That is, based on the payload type *pt*, RTP for $(pt < 64) \cup (pt \geq 96)$ and RTCP for $(pt \geq 64) \cap (pt < 96)$. Audio and video payload is distinguished by payload type, each elementary stream uses a unique SSRC identifier. There is no session description protocol, the payload is fully described by a predefined RTP profile and *pt* value. Dynamic payload types are defined according to the following table.

Payload type	**Name**	**Clock rate**	**Description**
96	opus	48,000 Hz	Xiph.Org Opus audio
97	H264	90,000 Hz	MPEG-4 Part 10 video
104	VP8	90,000 Hz	Google VP8 video
105	VP9	90,000 Hz	Google VP9 video

Table 4.1: Payload types of the RTP profile

Opus [13] was chosen as primary generic audio and speech encoder. The assigned *pt* value is 96. Opus works in hybrid mode consisting of SILK and CELT coders, features its own packetizer with packet loss concealment and forward error correction. Opus streams are self-descriptive without any need for additional out-of-band parameters and can change bandwidth and bitrate on the fly. The RTP clock rate is fixed at 48kHz, however the actual bandwidth may be anywhere between 8kHz to 48kHz with one or two coupled channels. SILK is a linear prediction coding based voice codec, that is responsible mainly for the low 8kHz band of the encoded signal spectrum. SILK encoding is multiplexed in Opus bitstream and is mostly used during voice activity

and in low bitrate / low bandwidth operation. CELT (Constrained Energy Lapped Transform) is a modified discrete cosine transform based generic audio codec operating with relatively small, overlapping blocks, which leads to a small latency of 20 ms. CELT operates with adaptive bitrate of 24kbps to 128kbps per channel in full-band frequency range. It is used primarily in high bandwidth, low delay operation and during no voice activity. A reference implementation of the Opus codec is available in C language and is optimized for embedded platforms, both the specification and implementation are royalty free and open source. The application interface is highly sophisticated and automatically adjusts codec parameters based on specified bitrate ceiling and real-time stream condition.

Incoming RTP packets are buffered by their SSRC value with an internal 48kHz clock synchronized to each SSRC. Packets are reordered based on their sequence numbers; when a packet loss is detected the condition is passed to the Opus decoder which then performs loss concealment. These events are monitored in the RTP session and reported with RTCP, the session manager implementation should maintain per-session quality of service statistics and adapt encoder bitrate ceiling. The packets are decoded synchronously based on their timestamp, thus producing a continuous playback stream.

Video support is provided by several codecs. As there are many platforms with hardware support for H.264 AVC encoding a binding is provided for *pt* 97. Many portable cameras provide M-JPEG encoded streams, these may be transmitted directly without the need for transcoding with *pt* 26. The recommended video codec is VP9 that is provided via open source package *libvpx* bundled with the previous VP8 version; assigned *pt* values are 104 and 105 respectively. Video streams have clock rate of 90kHz and unique SSRC identifiers. Buffering of video packets is performed in a similar way as for audio. Synchronization of audio and video streams is done by recovering common wall clock from RTCP sender report and synchronizing session clocks. The sender report packed header has the following format:

```
Version | P | RC | SR | Length | SSRC
```

Version is a 2-bit value indicating the protocol version (2).

P is a padding bit (not used).

RC is a 5-bit number of receiver reports in this packet.

SR is an 8-bit packet type (200).

Length is a 16-bit number indicating total packet length in 32-bit words minus one.

SSRC is a 32-bit clock identifier used to generate this report.

The header is followed by sender block:

```
NTP timestamp | RTP timestamp | Packet count | Byte count
```

NTP timestamp is a 64-bit Network Time Protocol timestamp, consisting of 32-bit integer part and 32-bit fractional part. NTP is used to synchronize the wall clock across different machines.

RTP timestamp corresponds to the 32-bit SSRC clock value and is used to synchronize the session clock to the wall clock. Both NTP and RTP timestamps are sampled upon transmission time of a RTCP sender report packet.

Packet count and *Byte count* are 32-bit counters of RTP packets used to measure throughput and error rate.

The sender block is followed by receiver blocks, one for each remote SSRC in a session. The receiver block is used as a feedback to measure quality of service and adjust streaming parameters. It contains following 32-bit fields:

```
SSRC | Loss count | Sequence number | Jitter | Timestamp | Delay
```

SSRC is a source clock identifier of this report block.

Loss count is a number of packets lost with 8-bit fractional part and 24-bit cumulative part; used for dynamic encoder bitrate adjustment to avoid network congestion.

Sequence number is the highest RTP sequence number received so far; used for loss calculation.

Jitter is an inter-arrival time calculated as mean deviation of transit time; used to adjust the RTP buffer size and control playback latency.

Timestamp is the most recently received NTP timestamp while *Delay* is the time since then; used to calculate propagation delay and synchronize playback time.

To synchronize the streams a time difference is calculated as

$$t_{\Delta} \;=\; (t_{now} - t_{NTP}) \;-\; (t_{run} - t_{RTP}) \tag{4.1}$$

where t_{now} is the current time (real-time clock), t_{NTP} is the NTP timestamp of the synchronization packet, t_{run} is the current stream time (RTP clock) and t_{RTP} is the RTP timestamp of the synchronization packet. Relative offset of a stream is described by t_{Δ}, such an offset is calculated for each elementary stream and the result is applied to the stream time

$$t_{\Delta run} \;=\; t_{\Delta} \;-\; t_{\Delta min} \tag{4.2}$$

where $t_{\Delta run}$ is a correction value for the stream clock and $t_{\Delta min}$ is the minimum offset value across all synchronized streams.

An RTP session manager collects the following per-stream statistics:

- bitrate
- fraction loss
- round trip time
- jitter time

These values are used at both the encoder and the decoder to optimize the parameters for stream quality. If the statistics collection is not needed by the application and only simple lip synchronization is necessary, the RTCP signaling may be omitted altogether. A lightweight alternative synchronization algorithm uses a common clock base for audio and video streams. The transmitter calculates timestamps as

$$t_{RTP} = (t_{base} + t_{run})\ f_{RTP} \tag{4.3}$$

where t_{base} is a common randomly generated base time shared by both streams, t_{run} is the running time of the stream and f_{RTP} is clock rate of the particular stream. The stream running time is recovered at the receiver by

$$t_{run} = t_{0\,run} + \frac{t_{RTP}}{f_{RTP}} - \frac{t_{0\,RTP}}{f_{0\,RTP}} \tag{4.4}$$

where $t_{0\,run}$ is a reference running time, $t_{0\,RTP}$ is a reference timestamp and $f_{0\,RTP}$ is a reference clock rate. The reference values are sampled for a first received buffer, the running time is then adjusted for clock drifts over time. The pipeline running time should be either a low granularity fixed-point value or a floating-point value so the clock precision is not lost during the calculations, for example a 64-bit nanosecond counter is appropriate.

4.2 GStreamer integration

An implementation of the RTP manager, synchronization and integration with codecs is done with GStreamer [14] framework. GStreamer is an open source set of libraries and utilities used for multimedia processing and real-time streaming. It has a modular interface and large collection of plugins for system integration as well as codecs and processing tools. The framework is built on top of GObject class system in objective oriented manner; the classes are exported through multiple language bindings and are portable to most hardware and software platforms.

GStreamer provides an extensible framework which allows application to create a streaming pipeline by connecting several dynamically loadable elements. Elements are classes, they have configurable attributes and emits signals to report event to the application. A typical processing element will provide a sink pad, on which it consumes buffers and a source pad, on which it produces buffers. These pads are linked together to create the pipeline. Together with buffers, there are events which may travel through the pipeline. Some of the important events are

- stream-start
- end-of-stream (EOS)
- segment
- latency
- caps

These events are used to transfer stream specific metadata, for example the caps event describes the format of the stream payload. GStreamer features dynamic format negotiation, the format is defined by caps in a MIME like format: `audio/x-raw, format={ F32LE, F64LE, S8, S16LE, S24LE, S32LE }, rate=[ 1, 2147483647 ], channels=[ 1, 2147483647 ], layout=interleaved`.

It is clear that there are multiple format options and a wide range of supported sampling rate and number of channels, the actual format is dynamically negotiated when pads are linked together. This is done via queries, which are performed on the linked pads. An application may also force the format by inserting caps filer such as `audio/x-raw, format=S16LE, rate=41000, channels=1`.

The pipeline is processed in a separate system thread, communication between the elements and the application is done via a GStreamer bus. Elements asynchronously post messages from the streaming thread and the application synchronously retrieves the messages from withing its main thread. Errors, warnings and run-time stream information is delivered to the application via this messaging system.

The following diagram describes a simple pipeline construction for streaming audio.

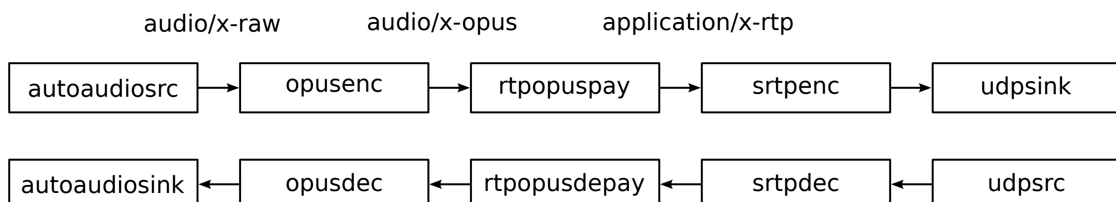


Figure 4.1: Pipeline construction

Each pipeline is scheduled in a separate system thread so encoding end decoding is evenly spread across CPU cores. The `autoaudiosrc` and `autoaudiosink` elements use a feature called auto-plugging. GStreamer inspects the system to find an appropriate element at runtime by scanning all elements that match the requested class (Source/Audio or Sink/Audio) and selecting the highest ranked element. Elements are ranked by their relevance to the working environment and available hardware acceleration. This means that if the application runs on a specialized hardware with GStreamer drivers installed, that driver will be prioritized by GStreamer auto-plugging mechanism. The auto-plugging also hides platform specific details and makes the application portable across different operating systems and devices. Typical audio backends are ALSA, PulseAudio, OpenAL or DirectSound.

ALSA (Advanced Linux Sound Architecture) is a native Linux kernel interface for direct audio hardware access. Most specialized hardware devices and SoC solutions have an ALSA module available that ships with that particular device. Advantages of ALSA are primarily low latency and low overhead with full hardware acceleration including direct memory mapping of physical audio ring buffers. ALSA is ideal for embedded solutions build on Linux platform.

Pulse is a high level audio server that features audio mixing, streaming and per-application configuration such as volume adjustment and resampling. Pulse is a multiplatform solution that is build on top of ALSA on a Linux system. Most desktop distributions are shipped with Pulse as a default sound server, GStreamer will automatically select Pulse driver on these platforms.

OpenAL and DirectSound interfaces are useful mostly outside Linux, DirectSound is a Microsoft specific implementation while OpenAL is a portable audio solution that works on almost any system. GStreamer will use these backends only if no other platform specific drivers are available.

The encoder and decoder element of the shown pipeline is Opus provided via `libopus` library, the respective stream caps are `audio/x-opus` and
`application/x-rtp,encoding-name=OPUS`.

Video support may be provided explicitly, for example with `vp9enc` and `vp9dec` elements using a `libvpx` library. Additionally `encodebin` and `decodebin` elements may be used to dynamically auto-plug codecs. The `decodebin` element will find a decoder suitable for its input format and output a raw media stream. The `encodebin` element will find an encoder based on an application provided encoding profile, which specifies the output caps. GStreamer selects an optimal codec automatically, same ranking rules applies, codes with hardware acceleration will be prioritized. GStreamer includes support for technologies such as VDPAU, NVENC, VAAPI, OpenMAX or DCE. The following table lists accelerated codecs provided by some of the GStreamer plugins.

	gst-plugins-bad	gstreamer-vaapi	gst-omx
JPEG	—	VAAPI	OpenMAX
H.264	VDPAU, NVENC	VAAPI	OpenMAX
VP8	—	VAAPI	OpenMAX

Table 4.2: GStreamer hardware video acceleration support

VAAPI (Video Acceleration API) is open-source X Window System Unix-based specification supported by GPU manufacturers such as Intel, Imagination Technologies or S3 Graphics. Accessible to GStreamer through the gstreamer-vaapi plugin.

VDPAU and NVENC are currently supported only by NVidia for their family of GPUs. GStreamer integrates these technologies in gst-plugins-bad package.

OpenMAX is an open-source specification managed by the technology consortium Khronos Group. It is a “royalty-free, cross-platform set of C-language programming interfaces with varying hardware support mostly in embedded devices. Accessible to GStreamer through the gstreamer-omx plugin.

DCE (Distributed Codec Engine) is an open-source library provided by Texas Instruments targeted at embedded ARM platforms. In GStreamer this library is wrapped via the gstreamer-ducati plugin.

A similar auto-pluging functionality is provided for upstream video via `autovideosrc` and `autovideosink` elements. Typical video backends are V4L2, XVideo, OpenGL/EGL or DirectDraw.

V4L2 is a native video driver of the Linux kernel supporting a wide range of devices. V4L2 allows interfacing to hardware encoders of the camera hardware, which is very useful as many USB camera devices already provide JPEG or H.264 encoded video for free.

XVideo is a direct video rendering extension for the X window system, complemented with cross-platform OpenGL and EGL drivers. DirectDraw is a Microsoft specific driver for their family of systems.

The payload is transfered via `udpsrc` and `udpsink` elements operating over a single physical socket. Each stream (RTP and RTCP separately) shall have its own UDP element with a shared socket. These elements can synchronize the output streams just before transmitting and allows per-stream configuration.

The session manager is implemented with RTP specific elements: `rtpsession`, `rtpssrcdemux`, `rtpptdemux` and `rtpjitterbuffer`.

The `rtpsession` element manages a session context and generates RTCP packets. The session context contains source information for each stream together with gathered quality of service statistics.

The `rtpjitterbuffer` element implements a buffer queue with packet reordering and inter-stream synchronization.

The `rtpssrcdemux` element separates the incoming packets into elementary streams based on the SSRC value.

The `rtpptdemux` element provides downstream caps signaling and negotiation of the payload format based on the *pt* value and allows the dynamic auto-plugging.

The following diagram illustrates interconnections within the session manager. Solid lines represent RTP packet flow (`application/x-rtp`) while dashed lines represent RTCP packet flow (`application/x-rtcp`).

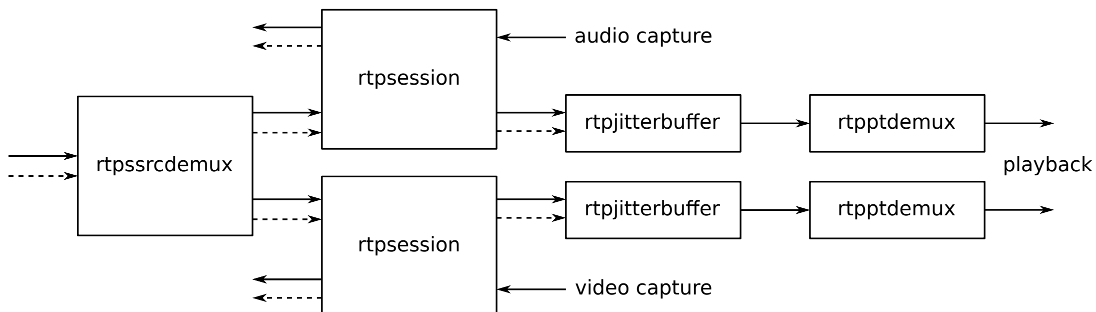


Figure 4.2: RTP session management

Incoming traffic from the network flows from left to right of the diagram, outgoing traffic flows from right to left. The leftmost element is the `rtpssrcdemux`, which sinks the incoming network stream, decodes the SSRC field of each packet and sources elementary streams with their unique SSRC values. This de-multiplexing is done for both RTP and RTCP packets. Each stream is then fed into a `rtpsession` manager which processes the RTCP reports and generates RTCP feedback. The session manager handles two RTP sources (internal and external) as it is provided with both received and transmitted RTP packets. Both sender and receiver reports are generated by the element, statistics are accumulated for each source. The application may then retrieve the stream statistics and take appropriate measures such as latency or bitrate adjustment.

The next element is the `rtpjitterbuffer` which sinks both the RTP and RTCP packets, which are used for timestamp recovery. The jitter buffer calculates clock skew of the transmitter and receiver clocks and generates smoothed skew-adjusted timing. Buffering of the packets increases latency of the stream, the latency is signaled with

a latency event by the jitter buffer. These latency events are collected by the sink elements which then synchronizes playback of each stream.

The trailing `rtpptdemux` element parses the *pt* field of RTP packets and extends the caps with stream specific fields:

- `media` - either audio or video
- `payload` - payload type number
- `encoding-name` - name of the payload encoding
- `clock-rate` - RTP clock rate

This metadata is sent downstream via a caps event to be used by depayloders and decoders, thus enabling the dynamic auto-plugging pipeline construction.

GStreamer features several plugins for raw audio and video processing and is easily extensible with user defined filters. It is possible to split a single pipeline into multiple execution threads by inserting the `queue` element which works as a FIFO queue and defines the new thread boundary. On a multi-core machine this may overcome the limitation of an processing element exceeding the 100% utilization of a single CPU core at the cost of higher latency and scheduling overhead.

Some basic audio processing filters are:

- `volume` - linear scaling
- `audioconvert` - format conversion such as sample width or channel count
- `audioresample` - sampling rate adjustment
- `audiofirfilter`, `audioiirfilter` - FIR and IIR filters with custom kernels

Some basic video processing filters are:

- `videoconvert` - color space conversion
- `videoscale` - rescaling of the video frame
- `videorate` - frame rate adjustment
- `videobalance` - brightness, contrast, hue and saturation adjustment

Additional processing is provided by audio and video backends. For example PulseAudio server features echo cancellation filter via `echo-cancel` item in its `stream-properties`. V4L2 devices exposes many hardware settings such as exposure time or backlight compensation.

A custom audio filter may be implemented by deriving the `GstAudioFilter` class. The base class have two virtual methods `setup()` and `transform_ip()`. The `setup`

method is called whenever the stream format is changed and is passed a `GstAudioInfo` structure as argument. This structure describes sampling rate, bits per sample, channel count and layout of the buffers. The `transform_ip` method is then called for each buffer being processed containing the individual samples. This method modifies the buffer in-place by directly mapping and modifying its contents.

A tone generator used for in-band signaling is implemented using this base class and registered to GStreamer core. The tone generator sinks the input audio from user microphone and if activated, overwrites it with generated tones such as ringback or hangup tones. This implementation is more simple and straightforward then creating a separate signaling stream which would need demultiplexing at the receiver end.

The RTP encryption elements also needs to be implemented manually. GStreamer exposes its RTP packet management utilities via a `gstreamer-rtp` library integrated within the framework. The implementation is derived directly from the `GstElement` class which has a `chain()` virtual method that is used for downstream chaining of buffers. The buffers can be mapped via `rtp_buffer_map()` and `rtcp_buffer_map()` methods and processed by cryptographic primitives provided by a separate library.

The RTP encryption model as described in the Protocol mapping (3.2) chapter uses IETF compatible AEAD construction. This construction is provided for example via the `libsodium` [9] library as `crypto_aead_chacha20poly1305_ietf_encrypt()` and `crypto_aead_chacha20poly1305_ietf_decrypt()` functions.

The integration of a video output with the application GUI is done via a `GstVideoOverlay` interface which is implemented by all video sink elements. This interface specifies `prepare_window_handle()`, `set_window_handle()` and `set_render_rectangle()` abstract methods. The `prepare_window_handle` method post an element message on the pipeline bus informing the application that a new video renderer is about to be created. The application that invokes the `set_window_handle` method passing its platform native window handle. The `set_render_rectangle` method is then used to specify the drawing region within the specified window used by the renderer. This interface is platform independent, the window handles are opaque pointers that varies across the actual implementations. As the video output is usually hardware accelerated these method just provide a negotiation mechanism between the window manager and the rendering hardware.

5 Application

A complete end-user application was implemented and published [1]. The application is written completely in C and uses the GTK+ and GStreamer frameworks. The GUI is designed to be clear, simple and minimalistic. Upon startup the user is greeted with a main application window shown in the picture below.

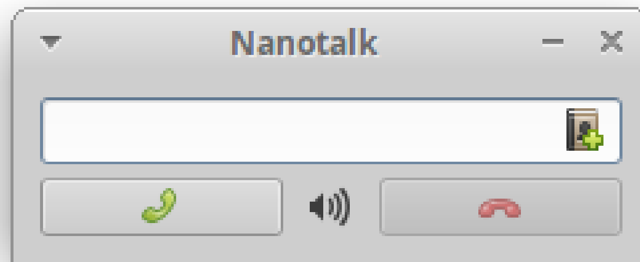


Figure 5.1: Main application window

The main window contains a search entry widget with an editor launcher, dial and hangup buttons and a volume control widget. The respective GTK widgets are `GtkEntry`, `GtkButton` and `GtkVolumeButton` packed in a `GtkGrid` container. The window and its widgets use a system defined visual theme and native named icons. The entry box is a place where the called party name is specified. There are two naming schemes, node identifiers and identifier aliases. A node identifier is the raw DHT node ID encoded into a Base64 string, that is a string of 27 a-z and A-Z characters (case sensitive) and a trailing equal sign character. An identifier alias is an UTF-8 encoded, user defined string that is mapped to a node ID. The aliases are fully in the user's control and may be used instead of the cumbersome node identifiers. They are locale independent, may have any length and are stripped of any leading and trailing whitespaces. If the user enters an alias into the text entry, it is transparently translated to the actual node ID by the application. During an incoming call the remote peer ID is also backward-translated to an alias if there is any.

Each user maintains his/her own list of aliases with an editor available via the editor launcher button at the right of the text widget. This opens a dedicated editor window which allows the user to edit and save the aliases list. The list is saved to a hard drive in a plain text form (UTF-8 encoded) and parsed by the application. Each line represents a single alias mapping, beginning with the actual node ID followed by a user defined text identifier. This textual format is parsed into a `GtkListStore` object which implements a `GtkTreeModel`. The `GtkTreeModel` is an abstract data model interface implemented by `GtkTreeModelFilter` and `GtkTreeModelSort` classes allowing searching and sorting items in the data model.

The following picture shows the editor window.

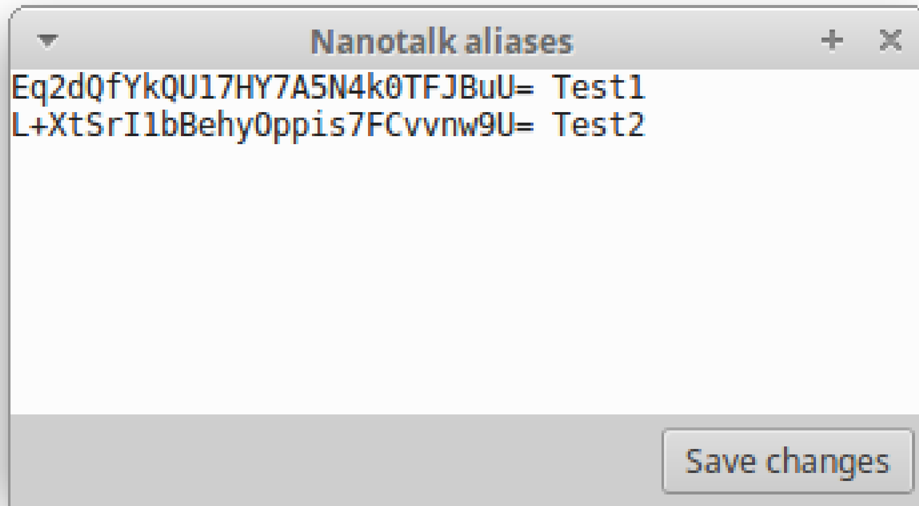


Figure 5.2: Aliases editor

The window has a `GtkTextView` widget in a `GtkScrolledWindow` container and a `GtkButton` widget in a `GtkButtonBox` container. These containers allows dynamic resizing of the window. Scrollbars are shown as needed and the inner text area can be scrolled if the window is too small to fit. The text view uses a system default monospace font so the IDs line up perfectly. The `GtkTextView` widget features standard text editing and presentation such as copy and paste clipboard support. The actual text is held in a `GtkTextBuffer` object that implements the `GtkTextIter` interface used for parsing the text into the `GtkTreeModel`. The `GtkTextIter` is a text iterator, it supports word and line navigation as well as searching within the iterated text. The parser is quite simple, each line represents a data model row, the first word relates to the first column of the data model while the rest of the line, stripped of any leading and trailing whitespace characters relates to the second column of the data model. The text buffer internally tracks made changes. When modified the text is saved to a file upon closing the editor window and the data model is updated.

A `GtkEntryCompletetion` manager is attached to the text entry of the main window. It uses the `GtkTreeModelFilter` and `GtkTreeModelSort` implementations of the `GtkTreeModel` interface to provide interactive auto-completion support to the user via `GtkCellRenderer` widgets. The second column of the data model is used i.e. the identifier alias, the filter is case insensitive. The auto-completion is automatically triggered when the user starts typing and displays available aliases matching the written text. The individual matches are displayed in a drop-down menu and may be individually selected by the user. Both inline completion and inline selection is supported with

the arrow keys, so the user can interactively navigate through the displayed list. The picture below illustrates this process.

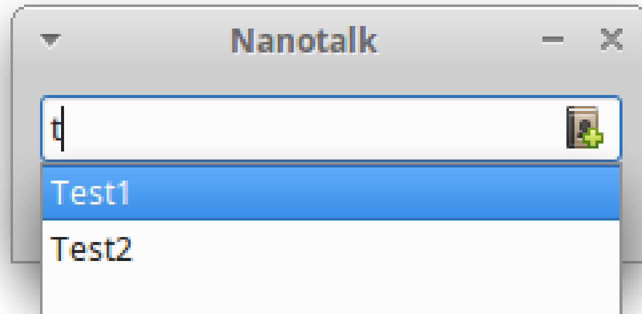


Figure 5.3: Automatic text completion

The dial and hangup buttons of the main window are mutually exclusive and may be operated directly from the text entry. When a submit key is pressed the currently sensitive button is activated. The dial button initiates a DHT lookup and prepares the streaming framework, the hangup button terminates the stream and reverts to an idle state. The streaming is started when a new connection is established the DHT client. Upon reception of an incoming call the text entry is filled with the calling party ID (or alias if available). A notification is presented to the user together with an event sound according to the system theme. The user has an option to either accept or reject the incoming call, the stream is muted in the meantime and a ringback tone is generated for the caller as an indication.

GStreamer pipelines are created and started as described in the GStreamer integration (4.2) chapter. The output volume of the playback stream is interactively manageable with the volume widget. In the case of a video capable stream a new window is created holding the video frame and configured for the GStreamer video overlay.

The application implements an extensible preferences system. There are three configuration files used by the application

- `user.cfg`
- `user.key`
- `aliases.txt`

The default location of these files is in the current user's application data directory. The paths are relocatable via a command line interface implemented with a `GOptionContext` manager.

The `aliases.txt` file contains the aliases map and is directly editable with the internal editor or by an external program.

The `user.key` file is a binary file holding the user's private key, this key is automatically generated by the application.

The `user.cfg` file is key-value configuration file managed by a `GKeyFile` parser. The file contains several preferences groups each having a dictionary of values. An example configuration file with default values is shown below.

```
[network]
enable-ipv6=false
local-port=5004
bootstrap-host=
bootstrap-port=5004

[audio]
echo-cancel=false
latency=200
bitrate=64000
enable-vbr=false
```

The network group contains preferences for the DHT client, such as port numbers or bootstrap addresses. The audio and video groups contains preferences for the GStreamer pipeline such as encoder bitrates. This file is loaded by the `GKeyFile` parser which provides getters and setters for the application. The getters retrieve the values from the configuration and parses them into their native type (string, integer, boolean). The setters updates the configuration with serialized values, it is also possible to modify the configuration by hand as it is contained in a plain text file.

The application features a configuration window where the user may edit the preferences in a graphical way. Preference groups are presented with the `GtkNotebook` widget, each notebook page uses a `GtkGrid` container with the respective controls. Each control is displayed a name with the `GtkLabel` widget and has a longer description in the tooltip hint shown on mouse hover. The widgets used for the controls are

- `GtkSwitch` - boolean value
- `GtkEntry` - string value
- `GtkSpinButton` - integer value
- `GtkComboBoxText` - enumerated value

The first page of the window displays a real-time status of the application as shown in the image below.

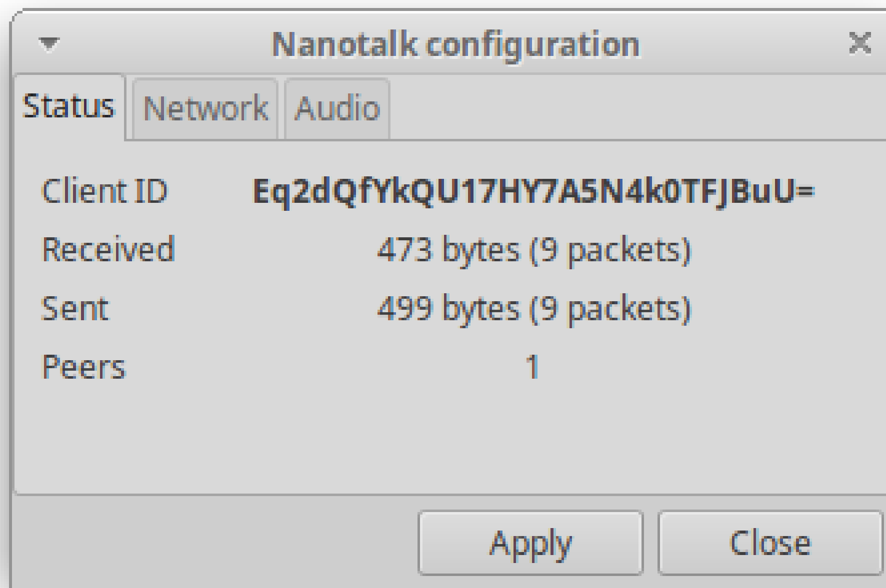


Figure 5.4: Status window

On the top is the ID of the user's key, by which the user is identified within the network. This ID has to be manually distributed to the foreign clients. It is customary that other users will create and save an alias for this ID.

Network statistics are shown below the ID, this includes the number of received and sent packets and their cumulative size. This provides the user with an overview of the network activity and serves as a indication of properly configured network. The current number of alive peers known by the DHT client is also shown as a quick feedback on the bootstrap process and current state of the distributed system. These statistics are automatically updated in real-time by a periodic background timer.

The following pages represent the preference groups. There is a `GtkButtonBox` container in the bottom of the window with "Apply" and "Close" buttons. The apply button propagates the changes by modifying properties of DHT client and GStreamer elements. If the new configuration is valid, the `GKeyFile` instance is updated and saved to a file. The video implementation is compiled into a runtime loadable plug-in to reduce program dependencies and size on a system, where video support is not feasible. The plug-in library is loaded on demand when a video capture hardware is detected on the running platform. The video preferences pages is not displayed if there is no video support available.

Network preferences are configured on the second page as shown in the image below.

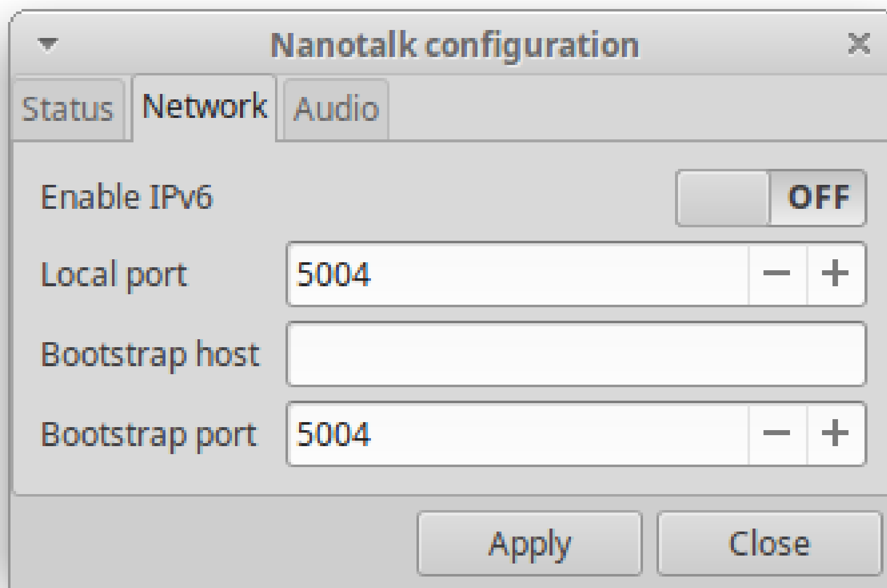


Figure 5.5: Network preferences

The IPv6 and local port preferences control the parameters to the DHT client constructor. When modified, a new client instance must be created. New network socket is bound to the specified local port and the network statistics are reset. The user is responsible for providing the required NAT forwarding rules for the specified port as discussed in the Protocol specification (2.2) chapter. It should be noted that when the IPv6 support is enabled, the client is able to communicate only with peers which also have the IPv6 support enabled, due to the limitation of the protocol implementation. However the IPv6-enabled clients work on both IPv4 and IPv6 networks due to the backward compatibility of the Internet protocol.

The bootstrap preferences control the parameters to the `bootstrap()` method. The host field accepts either an IP address (in dotted decimal or hexadecimal format) or a host name which is automatically resolved in the background. When modified, the method of the current client instance is invoked with the new address. The bootstrap process may be repeated any times, the user has a direct feedback in the status window. When the number of peers rises above zero and the number of received packets starts climbing up the configuration is proved to be working.

The next configuration page sets the audio preferences.

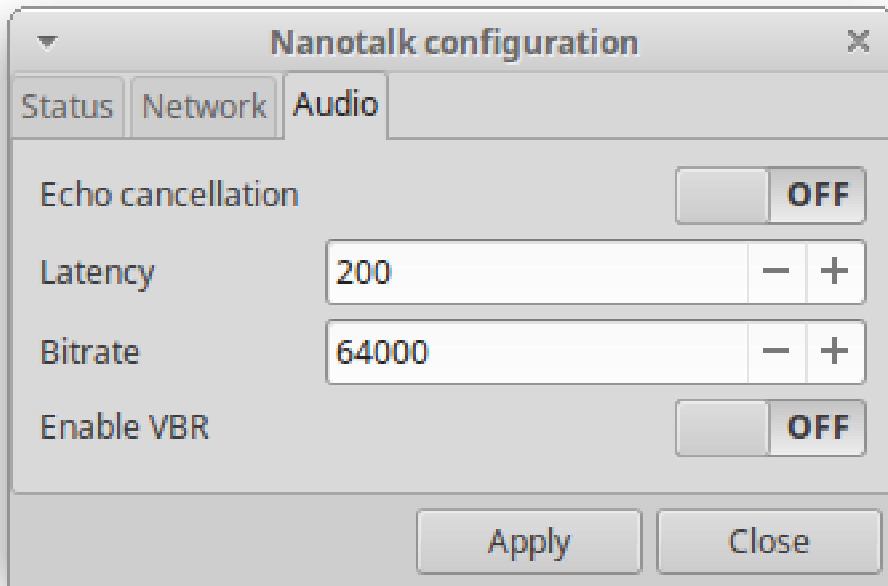


Figure 5.6: Audio preferences

Echo cancellation is implemented via a filter on the system sound server. The option controls the `echo-cancel` field of the `stream-properties` as discussed in the previous chapter. The switch widget is disabled on a system where this feature is not available.

The latency controls the `rtpjittterbuffer` latency in milliseconds. This control is shared with the video pipeline. Both streams have the same latency because of the synchronization, but the option is presented here as the video page is not displayed on systems without the video support. Setting the latency lower than the network jitter will cause quality degradation. The RTP session manager is able to detect this situation via the RTCP feedback, but this process takes some time while already streaming. Note that this is not the total latency of the pipeline, additional latency is introduced by the sink element, which depends on the selected audio backend.

Bitrate is set as a parameter to the Opus encoder. The VBR option controls the bitrate mode; if enabled a constrained variable bitrate mode is used, otherwise a constant bitrate mode is used. In constrained-VBR mode, the encoder uses the specified target bitrate as a ceiling, but does not fully utilize it when not necessary (for example during a period of silence).

This is the payload bitrate without encapsulation. The total bitrate is increased by inserting the RTP header (12 bytes per packet) and the authentication tag (16 bytes per packet). Additional 28 bytes per IPv4 + UDP headers or 48 bytes per IPv6 + UDP headers are inserted by the system networking stack.

The video preferences page is shown in the following image.

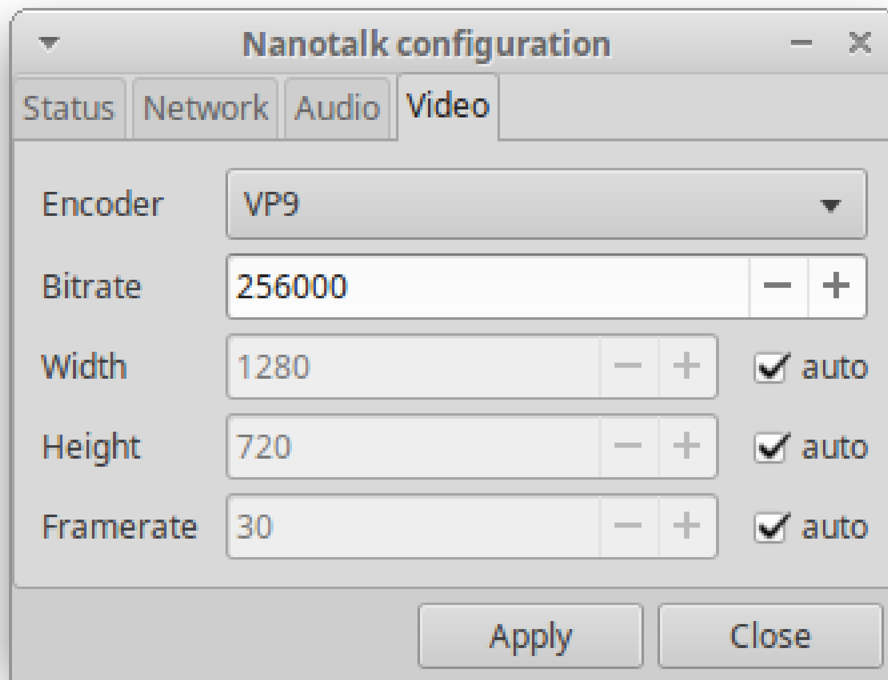


Figure 5.7: Video preferences

These preferences are used to setup the video encoding profile for the `encodebin` element. That is the encoder name, the bitrate property and optionally frame size and rate constrains. The "auto" checkbox controls whether the respective constraint should be explicitly included in the encoding profile. The `encodebin` element will automatically pick the best parameters if the encoding profile does not enforce them otherwise.

GStreamer will first try to configure the specified format directly on the capture device (many cameras already provide JPEG or MPEG output). If the capture device is not capable of producing the needed payload, a viable encoder is selected and inserted into the pipeline by the `encodebin` element. The `encodebin` element can additionally insert a `videoconvert`, `videoscale` or `videorate` element to match the input requirements of the encoder. This solves the problem of incompatible color spaces, resolutions or framerates respectively.

The application implements a language translation system using runtime text substitution provided through the GTK+ framework. Czech localization is provided and automatically selected based on the current system locale. The language packs are stored in separate dictionary files and loaded on startup, this process is fully automated by the GTK / gettext system.

Conclusion

This work presents a complete solution for a distributed multimedia service. The proposed lookup algorithm and the networking protocol solution is included. The algorithm was tested in practice and the functionality was proven under typical conditions. A protocol specific dissector for the Wireshark software was created for real-time benchmarking of the application. The distributed hash table algorithm scales well up to thousands of clients, allowing large service deployment. Both IPv4 and IPv6 networking protocols are supported for both Linux and Windows networking stacks. The software runs on both desktop and ARM based embedded systems, hardware specific features of each platform were taken into account. A complete security model was designed and implemented providing both user authentication and content confidentiality. The cryptography suit involves only a little processing overhead, the algorithms are optimized for each architecture. Modern cryptographic primitives were selected, the security strength of the protocol was designed with reasonable margin into distant future. The streaming framework was designed for integration of existing multimedia codecs; several possible solutions are presented. The implementation uses a sophisticated system of plugins which allows future extensions and support of new audio and video drivers.

Overall this work delivers a fully funtional end-user product and provides a core building block for future projects focused on the topic of distributed real-time systems. The application itself was tested in practice and tuned for best user experience. The user interface is simple, intuitive and configurable. English and Czech localizations were implemented. The user has control over most stream specific parameters, such as bitrate or resolution. Advanced filters such as acoustic echo cancellation are also available.

The software distribution is targeted primarily for the GNOME desktop, following the design guidelines and theming, but is portable to most available systems. Basic audio functionality can be established even on very resource constrained hardware with a 500MHz CPU clock and as little as 32MB RAM. With proper hardware acceleration, the video support does not provide considerable performance hit even at high resolutions if provided by the accelerator. However, on older systems with poor hardware acceleration capabilities, the video encoding becomes very CPU intensive and is better avoided altogether. The streaming framework does consume about 10% CPU time (load is spread evenly across multiple cores) on a typical middle-end desktop machine with fully featured desktop environment. The idle CPU usage is negligible, but the idle network overhead may exceed several hundreds of bits per second.

Abbreviations

DHT distributed hash table

NAT network address translation

PRF pseudo-random function

KDF key derivation function

AEAD authenticated encryption with additional data

MAC message authentication code

MTU maximum transmission unit

GPU graphics processing unit

GUI graphical user interface

References

[1] JAROS, Martin. *Nanotalk distributed multimedia service* [online]. [cited 25 Apr 2016]. Available from: https://github.com/martinjaros/nanotalk2

[2] BERNSTEIN, Daniel. *Curve25519: New Diffie-Hellman speed records*. 2006. Proceedings of PKC.

[3] BERNSTEIN, Daniel. *The Salsa20 family of stream ciphers*. 2007. The University of Illinois. Chicago.

[4] BERNSTEIN, Daniel. *ChaCha, a variant of Salsa20*. 2008. The University of Illinois. Chicago.

[5] GHODSI, Ali. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. The Royal Institute of Technology (KTH), 2006.

[6] MAYMOUNKOV, Petar and MAZIERES, David. *Kademlia: A peer-to-peer information system based on the XOR metric*. Springer, 2002.

[7] SAARINEN, M and AUMASSON, J. *The BLAKE2 Cryptographic Hash and Message Authentication Code*. 2015. IETF. RFC7639.

[8] NIR, Y and LANGLEY, A. *ChaCha20 and Poly1305 for IETF Protocols*. 2015. IETF. RFC7539.

[9] DENIS, Frank. *libsodium: A high-security, cross-platform, easy-to-use crypto library* [online]. [cited 25 Apr 2016]. Available from: https://www.gitbook.com/book/jedisct1/libsodium

[10] BAUGHER, M, MCGREW, D, NASLUND, M, CARRARA, E and NORRMAN, K. *The Secure Real-time Transport Protocol*. 2004. IETF. RFC3711.

[11] CASNER, S, FREDERICK, R and JACOBSON, V. *RTP: A Transport Protocol for Real-Time Applications*. 2004. IETF. RFC3550.

[12] PERKINS, C and WESTRRLUND, M. *Multiplexing RTP Data and Control Packets on a Single Port*. 2010. IETF. RFC5761.

[13] VALIN, J and VOS, K. *Definition of the Opus Audio Codec*. 2012. IETF. RFC6716.

[14] TAYMANS, W, BAKER, S, WINGO, A, BULTJE, R and KOST, S. *GStreamer Application Development Manual* [online]. [cited 1 Dec 2015]. Available from: http://gstreamer.freedesktop.org/documentation