Project Specification

IN2194: P2PSEC, SoSe 2015
Chair of Network Architectures and Services
TUM

Sree Harsha Totakura   Prof. Dr.-Ing Georg Carle

May 8, 2015

1 Project Description

The aim of the project is to build an anonymous VoIP application using a P2P architecture. We assume that users willing to use this application also participate in the P2P network by running their respective peers. Their peers contribute bandwidth (for routing other peers traffic), storage (for storing other peers' data), and computational power (needed for doing cryptography while routing traffic).

The underlying concept behind this project stems from Drac[1] which is based on APFS[3]. Our project builds on these ideas by having the following features/assumptions:

- A DHT is used for distributed data storage.
- Each peer participating in the network has a public-private key pair referred to as *hostkey*. The hostkey is generated with RSA and is 4096 bits long. It is stored on disk in the PEM format. The SHA256 hash of the public key gives the *identity* of the peer.
- Additionally, each peer has many pseudo-identities (also public-private key pairs).

Briefly, our VoIP application works as follows:

1. For each pseudo-identity each peer periodically chooses a random peer as its *exchange point* and builds an onion tunnel to that peer.
2. They then store their respective exchange points signed by their corresponding pseudo-identity into the DHT with the public key of that pseudo-identity as a key.
3. A peer wishing to call will then perform a lookup in the DHT for the exchange point of the caller. The lookup is performed by searching for the public key of the pseudo-identity of the caller.
4. If the peer is unable to find the exchange point, the call fails.

5. If the exchange point is found, the peer then builds an onion tunnel to that exchange point.
6. Call signalling happens via this onion tunnel until the exchange point, and from the exchange point via the onion tunnel the receiving peer has built.
7. After successful call parameter negotiation, the voice stream transmission starts.

The project is divided into sub-projects which focus on implementing the following key modules. Detailed description of these modules is available in Section 3.

DHT: A distributed hash table is needed for storing and searching the pseudo-identity to exchange point mappings. Furthermore, the DHT is also used for finding random peers by the KX and VoIP modules. This is accomplished by asking the DHT module to trace the path taken by the DHT GET request to reach the target responsible for a random key.

KX: Crypto layer implementing key exchange at the exchange points and onion encryption. This module has some functionality similar to Tor[2] for establishing onion tunnels.

VoIP: This module is responsible for call and identity management and also for interfacing with the operating system's audio interface. Call management includes signalling for call placement, hangup, holding, etc. Identity management includes management of pseudo-identities.

Testing: This module ensures the conformance of the above 3 modules. It also implements various attacker models and adds evaluation capabilities to the system.

2 Project Organisation

The magnitude of the implementation effort required to complete this project exceeds the course requirements for a student. For this reason, students are encouraged to form teams of two. If anyone has not formed a team but would like to, we can do team pairing during one of the lecture slots.

The teams will choose a sub-project to work on. The requirements and features to be implemented in these sub-project are listed in Section 3. The teams can use any programming language of their choice to implement these sub-projects. It is advised to choose one carefully after one's experience with it, its support for network communications, and library support. However, we enforce a restriction that your programs should run on GNU/Linux operation systems.

Git repositories will be created for each team to work on during the course of this project. It is advised that the teams commit their work into repositories allocated to them frequently. This helps in assessing the effort spent and also serves as a backup. Towards the deadlines the master branch of the git repository is considered for evaluation.

To avoid frequent merge conflicts, team members are encouraged to employ the following workflow while using the repositories:

1. Each team member starts his/her own working branch (possibly named after their lastname). This branch is to be used exclusively by that team member; the other team member should not rely on this branch, ever. So, if you work on multiple computers (a latop and a desktop), you can use this branch to sync your work.
2. Each team member always commits his work to his working branch first.
3. The current branch is then changed to master and any new incoming commits from the remote are pulled into it.
4. The working branch is then rebased on top of the master branch.
5. The master branch is then fast-forward merged with the working branch.
6. The master branch is then pushed to the remote.

There is a change in the way the repositories are setup. For this, we exploit some of the features provided by Moodle: there is an new activity added to the course which allows you to form groups. Please use it to register your team. You can choose any group from *group01. . . group30*. After setting up the groups, **upload your SSH keys (a key for each team member) as a file upload** in the *SSH Key Upload* assignment. Also **mention your team name in the online text** in the assignment. The repositories will then be created accordingly.

2.1 Reporting

Reporting for this project is accomplished through 3 reports. These reports help in balancing the workload throughout the semester, in encouraging teamwork, to identify any pitfalls in design and rectify them early. The reports also serve as a feedback for us to evaluate your progress and workload. Note that, reporting is different from code documentation; both will be evaluated separately.

The reports are to be submitted by placing them in the `docs` folder in the repositories allocated to you. For submission deadlines please refer to the next section.

The first report should document your implementation approach to your allocated module. It should contain why you have chosen a particular programming language, what your development environment is going to be, what libraries you plan to use and why. It should in essence give us an idea of how to replicate your development environment.

The interim report should document your module's design, the module protocol.

The final report should document any changes you made to the module's design and module protocol and their the corresponding reasons, the final module design and module protocol, implementation details and how to use your module.

Work load justification: We acknowledge that the project workload differs from student to student based on the choices they have made. For example, if there is no compatible cryptography library available for the programming language a team has chosen, it could be a considerable effort to write wrappers for a library available for another language. Also, a student in a team could spend more time in designing the protocols or on reporting than on coding. Such differences are dealt individually and for us to evaluate correctly, we ask you to report the activities you have chosen to do and the time spent for each of them. We believe that this leads to a fair evaluation of your effort.

2.1.1 Reporting Checklists

Before submitting your report please check if you have covered all the points in the following checklists:

Initial approach report

1. Indicate your team name, names of team members and, which sub-project you are working on.
2. Which programming language, operating system you want to use and the reasons for choosing them.
3. What type of build system is used.
4. What measures do you intend to take to guarantee quality of your software.
    - How do you write test cases for your software
    - Quality control. For example: by using Valgrind, cppcheck, etc.
5. What libraries are already available to assist you in your project.
6. Which license do you intend to assign to your project's software and reasons for doing so.
7. Team members' previous programming experience which is relevant to this project
8. How do you plan to share the workload in your team
9. Issues and complains

Yet to be covered: checklists for other reports

2.2 Deadlines

1. Initial approach [Sun 17. May 23:59]
2. Interim report [Sun 7. June 23:59]
3. Code review snapshot [Sun 19. July 23:59]
4. Written exam [2nd week of August]
5. Coding freeze [Sun 23. August 23:59]
6. Final report [Sun 6. September 23:59]

3 Sub-Projects

This section documents the functionality of the sub-projects. Since we expect that the various modules developed need to interface with one another, an interface is defined for every module in the form of network communication protocol that it has to adhere to. We call a module's communication protocol its Application Programming Interface (API).

The modules could be seen as layers similar to those in OSI networking stack. For example, the DHT module provides storage for KX and the KX abstracts building onion tunnels for the VoIP module. Each module on a peer is expected to communicate with other modules of the peer. The modules should also communicate with their counterparts of other peers. For example, the DHT module of a peer connects with the DHT module of peers who are its neighbours. Similarly, the KX module connects to the KX modules of other peers while building onion tunnels. The VoIP module on the caller peer, despite not having to connect to the VoIP module of the callee peer directly, will still need to communicate to the callee's VoIP module through the tunnel. Hence, every module needs a communication protocol to communicate with their counterparts on other peers. We call these protocols *module protocols*. You are free to decide on the module protocol for your module. However, note that the implementation details of the module protocol should be abstracted from the module's API.

Each module is to be implemented as a process which listens on a network socket. This approach helps in two ways: 1. the modules could be programmed in any language and could be running on any operating system which supports network communications; 2. any bugs in a module leading to segmentation faults or crashes could be isolated. The IP address and the port number on which the listening socket of a module is opened should be configurable as command line parameters. The module process is then expected to listen for connections on this socket.

It is a common assumption in P2P systems that misbehaving or malicious peers exist in the network. The core of P2P research lies in developing algorithms to tolerate the existence of such peers without making neither our peer nor the system vulnerable to them. We apply this to our project too and thus assume that all other peers we connect to or which connect to us could be either malicious or misbehaving. Misbehaving peers may not adhere to protocols strictly, or may crash randomly. Malicious peers may drop messages, store and replay them or modify them. Furthermore, they may exploit any vulnerabilities in protocols/implementations to crash or attack peers, *e.g.*, buffer overflow vulnerability due to the usage of `strcpy`. The same applies for the module API communications, although it could be assumed here that the risk is mainly caused due to programming errors and crashes in the other module. Thus, care must be taken while designing and implementing the module protocols to make a module resistant towards such attacks. For example, modules may drop connections if they observe any discrepancy in the communication protocol or after a response timeout.

Messages in all APIs are restricted to 64KB in length and start with a header which has field to store the length of the message (including the header) and the type of the message. The format of the message header is shown in Figure 1.

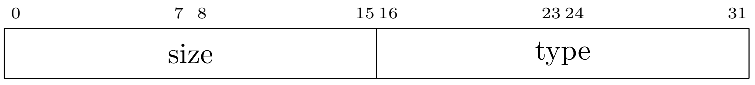

Figure 1: Message header format

Message types for the messages involved in the API have a globally unique number. It is advised that you do not use any of these numbers in your module-internal protocols.

bit formats for various messages involved in the module APIs are given in the following subsections. For illustration purposes (see Figures 1, 2), a message is split into 32 bit fields which are represented as a row. We call each such row a *word*. Words are further splitted to accommodate fields which are smaller than 32 bits. In that case, a vertical line is used inside the word to mark the beginning and ending of the field. Longer fields which typically encompass multiple words are illustrated by a space encompassing upto 3 words. The length of the field is then mentioned after the fields name. Variable length fields are illustrated by drawing a vertical cut through a similar space.

3.1 DHT

The aim of this sub-project is to build a Distributed Hash Table (DHT). You can choose to implement it with any underlying design like Chord, Kademlia, Pastry, etc.

Your implementation should satisfy the following requirements:

1. Support 256-bit keys.
2. The maximum amount of data stored under each key is 64KB.
3. Support storing multiple content under the same key.
4. Support replication; the replication degree should be configurable through a parameter.
5. Content should expire after a predefined time limit of 12 hours if not refreshed.

3.1.1 DHT PUT

This message is used to store some data under a key in the DHT. The message is identified by the message type `MSG_DHT_PUT` and is sent to the DHT module. It contains the key, Time-To-Live (TTL) value expressed in seconds, a replication degree, and the content to be stored under the key. The format of the message is show in Figure 2.

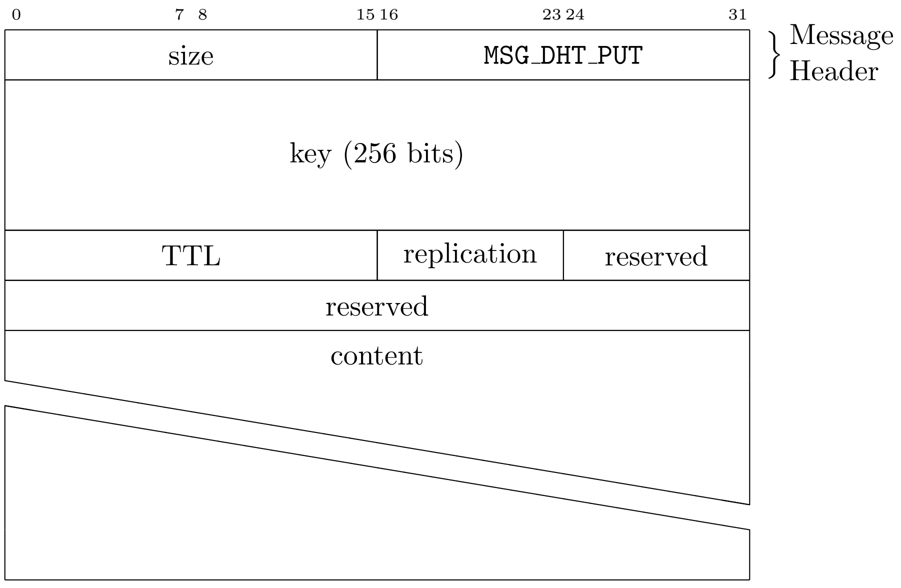

Figure 2: DHT PUT message format

The TTL value specifies the maximum amount of time a peer may store the content for the associated key. The maximum value for the TTL is 43200 (12 hours). If TTL is greater than the maximum, the behaviour is undefined.

The replication degree is a number which specifies the number of peers at which the content has to be replicated. The maximum value for this is 255.

The content to be stored under the given key follows the bits reserved for future extensions.

3.1.2 DHT GET

This message is used to retrieve values of a key from the DHT. The message is identified by the message type `MSG_DHT_GET` and is sent to the DHT module. It contains the lookup key and its format is show in Figure 3.

3.1.3 DHT TRACE

This message is used to trace the path traversed in the DHT implementation to reach the target peer responsible for storing the given key. The message is identified by the message type `MSG_DHT_TRACE` and is sent to the DHT module. It contains the lookup key and its format is show in Figure 4.

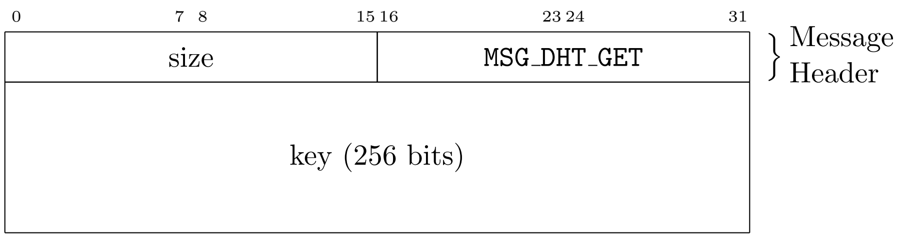

Figure 3: DHT GET message format

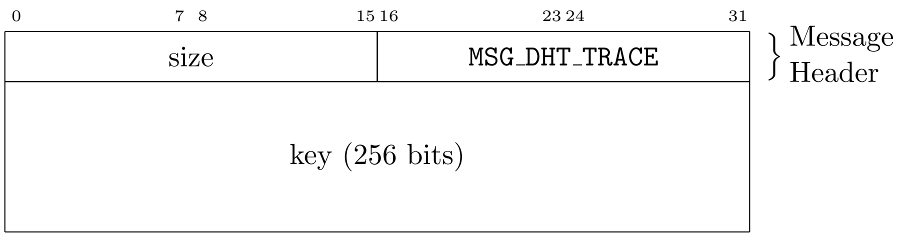

Figure 4: DHT TRACE message format

DHT TRACE is used to find random peers in the network by tracing the path taken for searching a random key. As the key is random, it is not guaranteed that a lookup for it will yield any result. Therefore, upon receiving a TRACE message, the target peer responsible for the random key should nevertheless reply even if it does not have the key stored.

3.1.4 DHT GET REPLY

This message contains the results of a key lookup in the DHT. It is identified by the message type `MSG_DHT_GET_REPLY` and is sent from the DHT module. This is the reply for DHT GET API message. This message contains the lookup key and the associated content. The format of this message is shown in Figure 5.

It is common in the DHT to have multiple content under the same key. It this happens to be the case for the given key, then the DHT module may send multiple of these messages for each retrieved content.

3.1.5 DHT TRACE REPLY

This message contains the results of DHT TRACE. It is identified by the message type `MSG_DHT_TRACE_REPLY` and is sent from the DHT module. This message contains the key for which the trace has been performed and the address information of the hops involved in the path. The address information of the hops is listed in the reverse order with the target peer (the last hop) address information towards the beginning.

The address information of a hop consists of the port number and the IP addresses (both IPv4 and IPv6) through which the hop's KX module could be reached. On a hop, this information is available from the peer's configuration. If the configuration does not

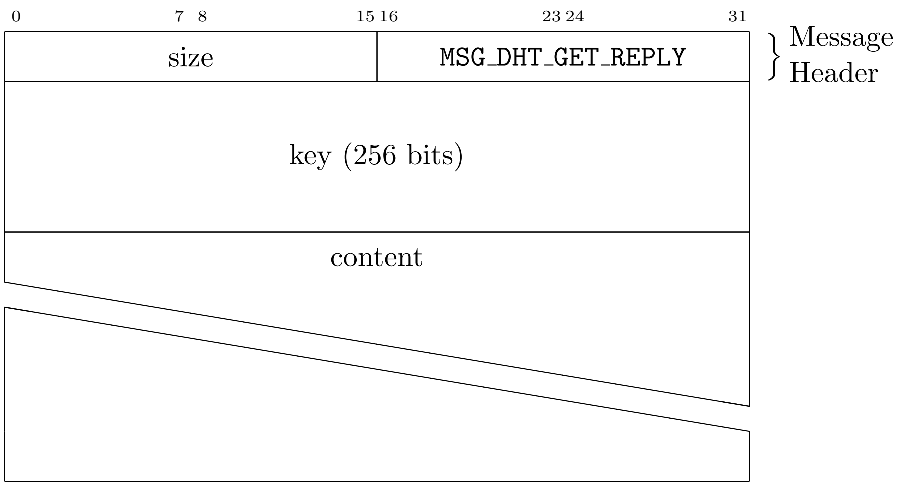

Figure 5: DHT GET REPLY message format

specify the address information of the KX module, the DHT module could assume a default KX port number of 3001 and its own IP addresses as a valid address information for the hop.

The number of hops contained in the message can be calculated from the size of the message. The format for this message is shown in Figure 6.

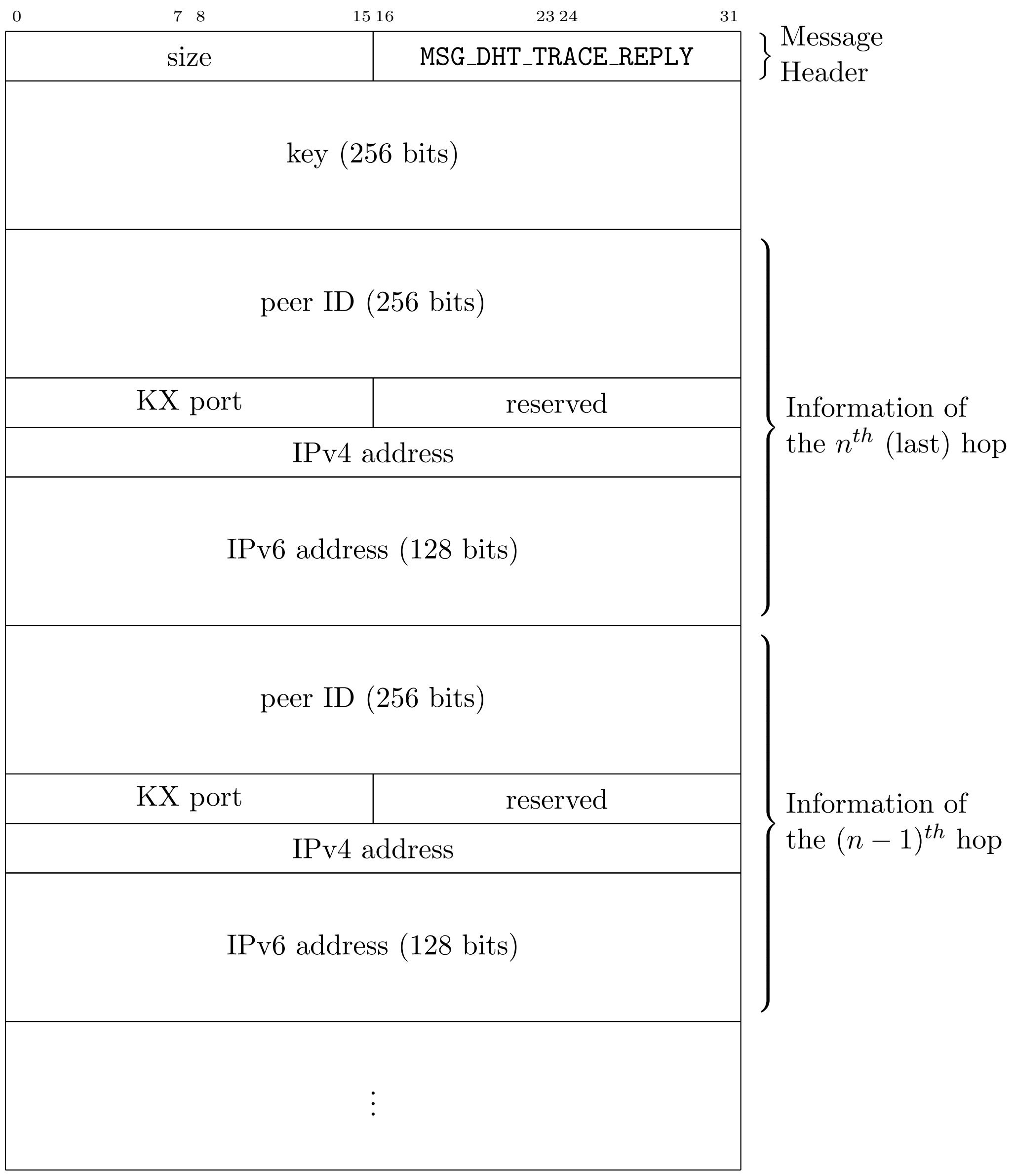

Figure 6: DHT TRACE REPLY message format

3.2 KX

The KX module deals with constructing onion tunnels. We distinguish two types of tunnels here: *incoming* and *outgoing* tunnels. Incoming tunnels are created by a peer to its exchange point to forward traffic destined to the peer. Outgoing tunnels are created by a caller peer to the exchange point of the callee peer. Incoming tunnels provide anonymity for the callee while outgoing tunnels provide anonymity for the caller. The KX module should facilitate building both types of tunnels. It should also handle traffic forwarding from the outgoing tunnel to the incoming tunnel and *vice versa* at an exchange point.

The KX module process should read the peer's hostkey from the file given as a command line parameter. The KX module processes in peers should listen on a predefined port and should accept connections from the KX module processes of other peers. The KX module processes of two peers use their respective hostkeys to form an ephemeral session key which they then use to encrypt all communications between them.

Tunnel building is done iteratively, i.e., a peer O building a tunnel selects the first hop peer H_1 at random, it then forms an ephemeral session key K_1 with the peer. It then uses K_1 to encrypt necessary meta-data to H_1 to instruct it to connect to the next hop peer H_2 and establish an ephemeral session key K_2. Note that K_2 is generated by the hostkeys of O and K_2, so that H_1 does not learn about it. The process continues until the tunnel has enough number of hops.

While building the onion tunnels there is a chance that you randomly pick attacker peers. Building a tunnel through them has the disadvantage that if all of the peers in the tunnel are attacker peers or co-operate, your communications can be deanonymised. For this reason, the more peers you have in the tunnel, the less is the risk as a single honest peer in the tunnel is enough to scramble your communication. On the other hand, having more peers increases the latency of the communication and may effect the call quality. For this reason, we restrict to a minimal hop count of 3 hops in a tunnel. The actual hop count could be parameterised via configuration or command line parameters.

Since we are building an application employing anonymity, it is important to defend against certain types of traffic analysis attacks which make use of packet sizes to determine the communication even though it is encrypted. For this reason, the KX to KX communication should employ packets of fixed sizes. The size of the packet is up to the module developer to decide. It is wise to choose a not too big or too small value. This also means that packets should be padded accordingly at each hop to maintain their fixed size.

The tunnels the KX builds are used exclusively for transmitting data from the VoIP module. Since the VoIP module requires TCP like reliable semantics for call control connections and UDP like semantics for the actual voice data connection, the KX tunnel should support both of them. This task could be achieved by tunnelling either of layer 4, 3, 2 traffic. Tunnelling layer 4 traffic is achieved by reimplementing TCP semantics at the KX module, which is quite a lot of implementation effort whose goals are not aligned with our project. The other alternative to tunnel layer 3 and layer 2 traffic is in this case attractive because we can use the underlying operating system's TCP stack for reliability semantics. Layer 3, layer 2 tunnelling can be achieved with the help of TUN, TAP devices respectively.

The advantages of employing layer 2 tunnelling over layer 3 tunnelling are not relevant here, because the VoIP module uses IP for both control and data connections. Hence the requirements are met by tunnelling layer 3 traffic with TUN devices. However, Windows supports only layer 2 tunnelling with TAP devices [1], so on this environment

[1] `https://stackoverflow.com/questions/12513580/using-tun-driver-in-windows`

we are restricted to using TAP devices. Native support for TUN device is available on GNU/Linux and OS X operating systems.

With TUN/TAP devices, the KX should be able to read IP packets/Ethernet frames from the control and data connections established by the VoIP module. The KX should then forward the read data through the tunnel to the tunnel's other end point (not the exchange point), where corresponding IP packets/Ethernet frames are created.

Before the VoIP module could make use of TUN, TAP devices, the KX module also should negotiate the tunnel IP addresses with the tunnel's end point and associate the corresponding tunnel device with an IP address. Without this, the VoIP module will not be able to know the destination to open the connections to. In the case of TAP device, the KX module may also need to negotiate MAC addresses and associate the corresponding TAP device with a MAC address.

Finally, the tunnels should be setup and torn down periodically. This is to decrease the risk of having an attacker peer in the tunnel and also to enhance load balancing in the system. The transition between an old tunnel and a new tunnel should be made transparent to the VoIP module.

3.2.1 TUNNEL BUILD INCOMING

This message is to be used by the VoIP module to request the KX module to start building an incoming tunnel from the given exchange point. The message is identified by `MSG_KX_TN_BUILD_IN` message type. See Figure 7 for the message format.

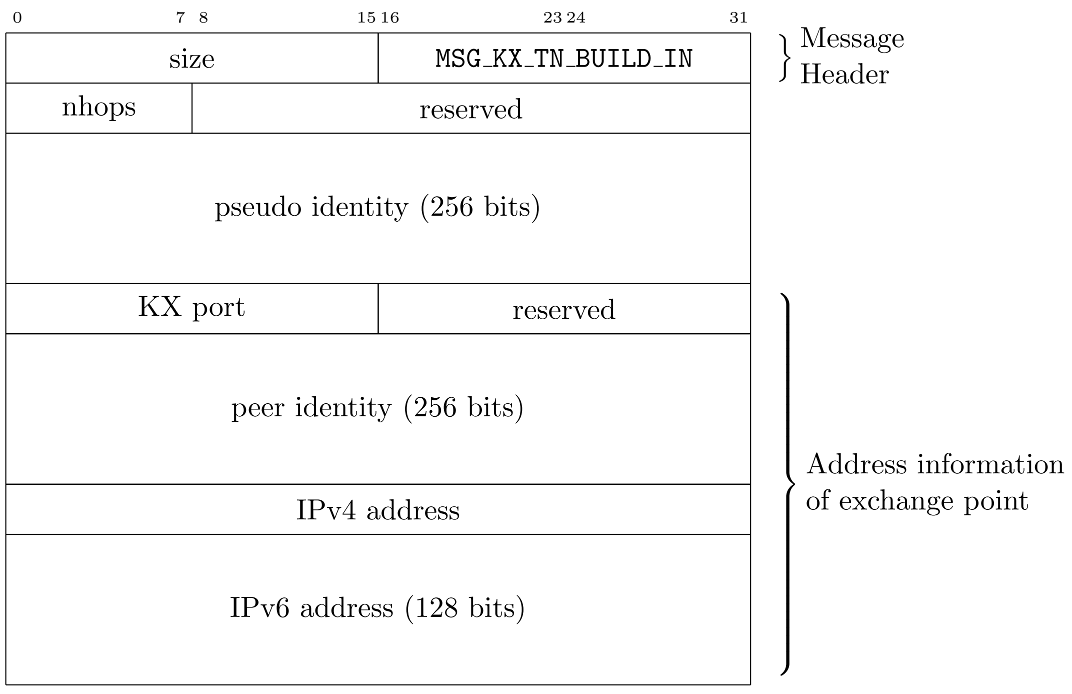

Figure 7: TUNNEL BUILD message

The message contains the required minimum number of hops in the tunnel, the identity of the pseudonym for which the exchange point should forward incoming data and, the address information of the exchange point.

3.2.2 TUNNEL BUILD OUTGOING

This message is identified by the `MSG_KX_TN_BUILD_OUT` message type and is similar to that of TUNNEL BUILD INCOMING message (see Section 3.2.1). This message indicates that the KX should build an outgoing tunnel instead of an incoming one.

3.2.3 TUNNEL READY

This message is identified by the message type `MSG_KX_TN_READY` and is sent by the KX module when the requested tunnel is built. It contains the pseudo identity given in either of the TUNNEL BUILD messages, the IP address of the newly TUN/TAP interface and the path of the TUN/TAP device expressed as ASCII encoded and zero-terminated string. See Figure 9 for message format.

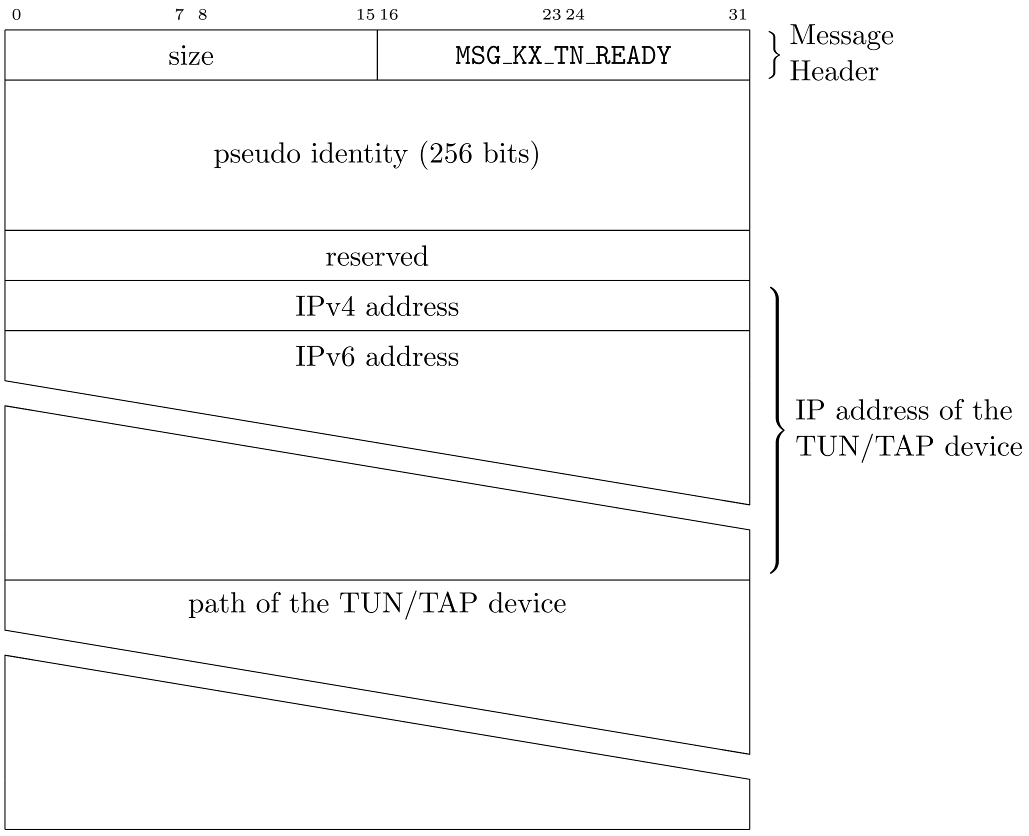
0
7 8
15 16
23 24
31
size
MSG_KX_TN_READY
Message Header
pseudo identity (256 bits)
reserved
IPv4 address
IPv6 address
IP address of the TUN/TAP device
path of the TUN/TAP device

Figure 8: TUNNEL READY message format

The IP address of the TUN/TAP interface is chosen by the local KX module if the built tunnel is an incoming tunnel. For outgoing tunnels, the ip address should be negotiated by the local KX module with the KX module on the tunnel's end point.

3.2.4 TUNNEL DESTROY

This message is used to instruct the KX module that a tunnel it created is no longer in use and it (and the associated TUN device) can now be destroyed. The message is identified by the message type `MSG_KX_TN_DESTROY`. The pseudo identity should be a value for which a tunnel exists, *i.e.,* the KX was called with a TUNNEL BUILD message (see Section 3.2.1 and 3.2.2) and TUNNEL DESTROY was not already called with the same pseudo identity.

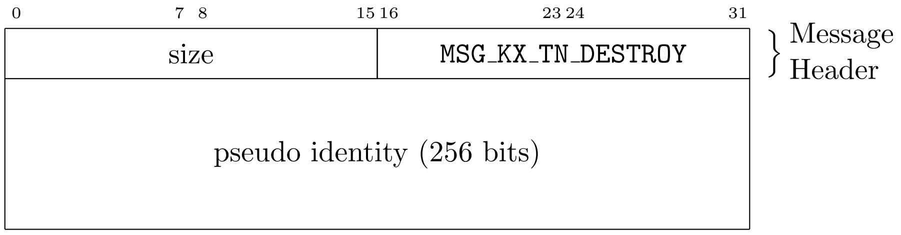

Figure 9: TUNNEL DESTROY message format

3.3 VoIP

The expected functionality of the VoIP module is that it provides an interface for the user for initiating and receiving calls. This could either be a graphical or command line interface. The VoIP module should also integrate with the underlying operating system's audio interface, *e.g.*, PulseAudio [2], to play the received audio stream from the other party and to record the local microphone input stream to send it to the other party.

Call management involves two connections: *control* and *data* connections. A control connection is used for signalling call initiation, busy, waiting, termination, etc., and also for establishing a session key with the call receiver. The control connection must have reliable transmission semantics and could be achieved by using a TCP over the TUN/TAP device created by the KX module. Data connection is the connection through which caller and callee audio streams are exchanged. This connection does not require reliable transmission semantics as some amount of packet loss could be tolerated for voice streams; thus this connection should use UDP.

The session keys for encrypting control and data connections are to be derived from the pseudo-identities of the caller and callee. For this to work, the caller is expected to give as an input the pseudo-identity of the callee. The VoIP module should then search for the pseudo-identity in the DHT; alternatively it may cache the pseudo-identity from an earlier DHT GET result. If a response is not found or if the response does not contain the address information of the callee's exchange point, the VoIP must signal an

[2]https://wiki.freedesktop.org/www/Software/PulseAudio/

error to the user after some timeout. In the other case, VoIP must retrieve the address information of the exchange point and ask the KX to build a tunnel to the exchange point.

For the caller to find the pseudo-identity of the callee, the callee's VoIP must choose a random exchange point using DHT TRACE and then do a DHT PUT with the pseudo-identity, the address information of the exchange point and the port number it is going to listen for incoming connections. Afterwards, the callee's VoIP must ask its KX to build an incoming tunnel from the exchange point. When the tunnel is built, the VoIP must listen on the tunnel IP address given by the KX and the port number it advertised in the DHT.

Generation of pseudo-identities is also delegated to the VoIP module. Pseudo-identities must be keys supporting derivation of session keys. Recommended are Diffie-Hellmann (DH) keys. The pseudo-identities are assumed to be 256 bits long. If you decide to use longer keys or is necessitated by the weaknesses in the underlying crypto, SHA256 sums of those keys could be used as their pseudo-identities.

There exists some protocols for streaming audio and video streams in real time. RTP and SRTP are some examples. If you are familiar with those or have found a supporting library compatible with your development environment, it may be worth looking into them.

To mitigate profiling attacks of an adversary snooping on your Internet connection. The VoIP should be able to make fake calls by pumping some cover traffic in the network such that it appears to the adversary that you are communicating with someone at times when you actually do not communicate. This could be achieved by self-calling, *i.e*, by finding a random exchange point, associating a newly generated pseudo-identity with it, publishing it in the DHT, building an incoming tunnel from the exchange point, doing a DHT GET on it after sometime, building an outgoing tunnel to the exchange point, and calling the same pseudo-identity.

The VoIP module does not have a API interface as no further modules are envisioned to use it. This module just needs to interface with the KX and DHT modules using their respective APIs.

3.4 Testing

The aim of this module is to test and evaluate the system. It involves writing testcases to test the compliance of above modules to the given protocol specifications, devising attack scenarios *e.g.*, attacker or misbehaving peers. This is achieved by building testing stubs and proxies for the module APIs. Stubs are used for blackbox testing module APIs by sending test messages according to the specification and then checking how the module responds. The proxies are used to alter, drop and inject new messages in the module protocols.

Furthermore, this module should develop tools or scripts to evaluate the modules under one of the following environments:

- testing based on mininet.

- testing on some laptops/computers in the university network.
- testing on planetlab.

Yet to be covered: Include details about mininet and planetlab testing environments.

The evaluations should evaluate various characteristics under different test configurations applicable in the given testing environment. An example for one such evaluation is: what is the impact of link latency on the performance of VoIP data?

4 ChangeLog

draft-0.0: First released draft version.
draft-1.0:

- Address information fields added to DHT TRACE and KX TUNNEL BUILD messages.
- Refined KX and VoIP description.

References

[1] George Danezis et al. “Drac: An Architecture for Anonymous Low-Volume Communications”. In: *Privacy Enhancing Technologies*. Springer. 2010, pp. 202–219.

[2] R Dingledine, N Mathewson, and P Syverson. *TOR: the onion router. Tor Project/EFF*. `https://www.torproject.org/`.

[3] Vincent Scarlata, Brian Neil Levine, and Clay Shields. “Responder anonymity and anonymous peer-to-peer file sharing”. In: *Network Protocols, 2001. Ninth International Conference on*. IEEE. 2001, pp. 272–280.