IVCM15Proto

Official Documentation

Requirements

- List all members in the chat system
- Private message to 1 user
- Message to everyone in the system
- Point-to-point file transfer
- Has to be reliable
- Needs some security
- Neighbors have to be friends and authenticate in the initial connection

Assumptions

- Each user has a RSA keypair that can be used for the UUID and all security related data transfers (initial neighbor authentication, encrypted chat, encrypted files).
- Stop-and-wait
- PGP must be installed on the machine running the software.

Jargon

- Our basic unit is called **message**, which consists of **header** and **payload**.
- Every instance of the application, when connected to the network, is called a **node.**
- In the documentation we refer to the user's machine from the user perspective as **node A**, or just **A**.

UUID

The UUID is based on the V4 Key ID from the OpenPGP Message Format RFC. The RFC describes the Key ID as follows:

"A V4 fingerprint is the 160-bit SHA-1 hash of the octet 0x99, followed by the two-octet packet length, followed by the entire Public-Key packet starting with the version field. The Key ID is the low-order 64 bits of the fingerprint." [1]

Implementations, such as GnuPG, differentiate between a short and long version of the Key ID of which the short version is usually shown in hexadecimal form. The difference between these versions is that the long version is the full 64 bits and the short version is the low-order 32 bits of the long version. In this protocol 32 bits is more than enough for the UUID and it is a bit easier to retrieve the short version of the Key ID. Therefore the UUID equals the short version of the Key ID.

Ubuntu: https://help.ubuntu.com/community/GnuPrivacyGuardHowto
MAC: http://notes.jerzygangi.com/the-best-pgp-tutorial-for-mac-os-x-ever/

Message

version	**source**	**destination**	**type**	**flag**	**hop count**	**length**	**payload**
1	4	4	1	1	1	1	87
100 Bytes							

Doesn't matter how you interpret bytes in your program, but the communication has to be a fixed bytestream.

Version	The protocol version. - (Integer in a single byte)
Source	Source UUID - (ASCII String)
Destination	Destination UUID - (ASCII String)
Type	Distinguishes between types of message - (Integer in a single byte)
Flags	Indicate message characteristics - (Integer in a single byte)
Hop count	Remaining amount of hops - (Integer in a single byte)
Length	Length of the payload - (Integer in a single byte)

Payload	The content of the message - (ASCII String)

Message - General info

The protocol handles ASCII encoding. Converting a string to bytes needs to be done with ASCII encoding.
Version, type, hop count and flag should be represented as integer in a single byte.

Example of integer in a single byte:
This is 123 -> 01111011
This isn't 123 -> 00000001 000000010 00000011

Payload, source, destination are strings (string is already bytes)
String = 1 Byte
Integer = 4 bits (½ bytes)
Example: Version field value set to (int) 1 = binary 0001 but it must be padded with 0000 0001 so it will be an **entire** byte. (*01 in hex*)

Message in Data exchange

Our protocol message is wrapped by a UDP datagram.

The maximum length of the UDP datagram **payload** itself can be 100 bytes.

Packets (our message: header+payload) must be sent without empty **trailing** data. (Without padding 0x00 in the payload)
Payload starts from the **14-th** of byte (included).

Example: Message header 13 + Message payload 70 = 83 -> UDP datagram total payload size itself has to be 83.

Version

Indicates the version of the protocol and is set to 0x01 for the test run.

Source

The UUID from which the message is generated.

Destination

The UUID to which the message is sent.

Type

Indicates the type of message:

- hex(0x01): **Data** message (0000 0001) (8th bit)
- hex(0x02): **Control** message (0000 0010) (7th bit)
- hex(0x04): **Authentication** message (0000 0100) (6th bit)

Control message

Utilised for management:

- Routing updates
- Flow control

Data message

Utilised for communication:

- Text messages data
- File data
- Authentication data
- Control data/Routing data

Authentication message

Utilised for authentication:

- Add new neighbor

Hop Count

The number of available remaining forwarding steps in routing the packet.
Hop Count fields initialised to **15**.

Length

Length of the payload in bytes.

Flags

Used to model different situations and trigger appropriate decisions.

Data message

3th bit	0x20	Routing updates
4th bit	0x10	Session
5th bit	0x08	File data (file contents)
6th bit	0x04	Text message data (chat messages)
7th bit	0x02	Sequence number. Because we use a Stop and Wait approach there is no need for more than 1 bit.
8th bit	0x01	Last fragment of payload, bit is then set to 1. **In case of non segmented message, this bit is set also to 1.**

Data messages are segmented and its segments are continuous (stop and wait) - we do not need segmentation anywhere else.
Control and Authentication messages will never be segmented **(you always ack 0 them back)**

Control message (no payloads)

2th bit	0x40	Session init
3th bit	0x20	File transfer init
4th bit	0x10	Routing update initialization
5th bit	0x08	Keep alive message
6th bit	0x04	ACK
7th bit	0x02	Sequence number
8th bit	0x01	RST

Sequence bit is always 0 in a control message, unless you send an ACK response for a specific data message with a different Sequence number.

The sequence number in the control message is used to ACK the corresponding data message sequence.

An ACK control type message is needed in response for every message that will be sent.(between two endpoints, see Routing/Forwarding messages paragraph)

If a user logs off gracefully the RST control flag comes in handy - by telling all peers that he is now gone and no extra traffic is needed from other peers to find out that he is gone. (so there won't be any temporary ghost peers)

Authentication message (no payloads)

7th bit	0x02	AuthenticationSuccess (Legacy name: auth_success)
8th bit	0x01	InitiateAuthentication (Legacy name: NoAuth)

No sequence defined. (ACK0 as response)
See example1 below - "authentication" is just a fancy word to describe our "no_auth" - "auth_success" shortcut.

General info - Communication

An ACK control type message is needed in response for every message that will be sent.

If a user logs off gracefully the RST control flag would come in handy - by telling all peers that he is now gone and no extra traffic is needed from other peers to find out that he is gone. (so there won't be any temporary ghost peers)

Example1: >> (sends) == (knows)

A >> Authentication message, **Initiate authentication**, (no sequence defined, so it's always 0)
B >> Control message ACK0 (type control, flags 0000 0100 -> ack for sequence 0)
B >> Authentication message, **Authentication succeeded** (no sequence defined, so it's always 0)
A >> Control message ACK0 (type control, flags 0000 0100 -> ack for sequence 0)

Example2:
ack**0** - type control, flags 0000 01**0**0 -> ack for sequence **0**
ack**1** - type control, flags 0000 01**1**0 -> ack for sequence **1**

A - Control message routing update, Control message seq=none (0)
B - ACK0
A - Data message with flag "control data" (Data messages seq=0)
B - ACK0
A - Data message with flag "control data" (Data messages seq=1)
B - ACK1
A - Data message with flag "control data" (Data messages seq=0)
B - ACK0
A - Data message with flag "control data" (Data messages seq=1)
B - ACK1
……

Routing

The goal of this chapter is to describe the different aspects used in the routing part of the protocol and consists of routing tables, timers and messages.

Every node keeps information on how to reach the other nodes connected to the network, saving the **Next Hop** and **Cost** in his own routing table.

Forwarding messages and Hop Count

Every message received with *hopcount* **=** 0 and *destination* **!=** **A** (oneself) must be dropped.
Every message received with *hopcount* ≠ 0 and *destination* **!= A** (oneself) must be forwarded decreasing *hopcount* by 1.
Hop Count fields initialised to **15**.

All forwarded messages (control, data) are not acked. Only the destination sends the ack, which will be forwarded all the way back.
So, the rule would be: anything **not** destined to you must be forwarded and NOT acked back.

Routing tables

The protocol needs two different routing tables: one for the physical neighbors and one for all the other nodes in the overlay network.

Neighbor table

Each entry in the neighbor table has the UUID and socket information that indicates a connected node via the underlay network. This table is used to determine where to send an overlay packet through the underlay network.

UUID	Socket (ip:port)	Passive Timer

Routing table

The routing table is used to determine the next step of a message via an immediate neighbor towards the destination of the message. The routing table starts with one entry, which is the current node with cost 0, and will eventually contain all the systems in the connected network as a destination via a neighbor.

Destination (UUID)	Via (UUID)	Cost (hops)

- TYPE: CONTROL
- FLAGS: ROUTINGUPDATE
- VIA field, should contain only direct neighbours
- PAYLOAD: (DESTINATION:4 bytes, COST: 1 Byte) = 4B4B4B4B01 => 1 line 5 bytes when sending/receiving
- Cost is represented in binary
- Everybody has himself in the ~~neighbour table and~~ routing table
- Routing table entries are sent in single byte array to ensure that it would be easier to delete lost routes
- Application should handle disappearing nodes correctly whenever a routing update is received.
- Application should be ready to take in routing updates any time
- Application should handle routing update triggers (time, can't reach node, etc)
- Protocol requires timed triggers, but it does not require other triggers
- The neighbour next to you cost will be 1.

Additional info

Routing here works as routing in RIP does.
Header will stay the same, exception made for hop count which will be decreased by 1.
And OF COURSE the source will stay the same.
To forward the message, I will look in the routing table for the destination, get his VIA field, and search that VIA (which is an UUID) in my neighbor table. From there I will get the correct socket I will use for forwarding the packet.

Timers

Timers are used to control the status of the network, particularly which neighbors are available.
Timers are divided in **active** and **passive** timers.

Active timers
Are kept by every node and trigger the send of a particular control message from **A**. They can not be overridden.

Routing table update	Set to 30s -- 0 triggers the routing update

KeepAlive	Set to 5s -- 0 triggers the keepalive

Passive timers

Are kept by every node and trigger the deletion of a node from the **A**'s routing table.
They are reset by 1 event:

- A KeepAlive message is received

Neighbor B timer	Set to 18s -- 0 triggers the deletion of node B from the routing table
Neighbor n	Set to 18s -- 0 triggers the deletion of node n from the routing table

Messages

The protocol uses various messages to authenticate and connect neighbors, update the routing table and to keep connections alive.

Authentication

The authentication messages handle the authentication between neighbors and make sure that both have added each other. The Destination field in the Initiate authentication is set to “FFFFFFFF”.

A wants to connect to B and become B’s neighbour.

A >> Authentication message, **Initiate authentication**, (no sequence defined, so it’s always 0)
B >> Control message ACK0 (type control, flags 0000 0100 -> ack for sequence 0)
B >> Authentication message, **Authentication succeeded** (no sequence defined, so it’s always 0)
A >> Control message ACK0 (type control, flags 0000 0100 -> ack for sequence 0)

B adds A as a neighbour. (and vice versa)

Routing table update

The routing table update is send to all neighbors and the payload consists of all the entries concatenated together with the 4 byte destination and 1 byte cost.

A: TYPE: **CONTROL**, FLAG: **ROUTINGUPDATEINIT**, seq=none (0)
B: ACK0
A: TYPE: **DATA**, FLAG: **ROUTINGUPDATE**, PAYLOAD: **DESTINATION**+**COST**+**DESTINATION**+**COST**+each entry, seq=0
B: ACK0
A: TYPE: **DATA**, FLAG: **ROUTINGUPDATE**, PAYLOAD: **DESTINATION**+**COST**+**DESTINATION**+**COST**+each entry, seq=1
B: ACK1
...

Handling an incoming table update

When a node receives a routing table from a neighbor it should compare the existing routes that go through the neighbor to the ones received from the neighbor and:

- Add new routes;
- Update if the cost is lower;
- Remove the ones that are missing in the update but exist in the current routing table.

Handling a time-out

When a time-out occurs keep trying for 3 tries before giving up.

KeepAlive

KeepAlive control messages are represented with the 5th flag set to 1. Their purpose is to notify the receiver of the available status of the sender. They are sent by every node to every neighbor with a hopcount of 1.

Handling an incoming KeepAlive

KeepAlive messages are only received and processed by updating the passive timer in the neighbor table.

Handling a time-out

KeepAlives are not ACK’ed and therefore a time-out does not matter.

Chatting and file transfer

Chatting and file transfer packets shall have the type 0x01. All type 0x01 packets shall contain symmetrically encrypted payload.

Session initialization

Before chatting or transferring files between nodes the parties must have initialized a mutual session key. Session key is a 128-bit value chosen by the initial sender. At any time can a party re-initialize the session (meaning to generate a new key and send it to the opposing party).

The session key shall be sent using the message type “data” with flag “authentication data” flag bit set. The key shall be PGP-encrypted for the recipient and then PGP-signed by the sender. The result shall be the payload (and be fragmented, if necessary).

They same principle applies for chatting with multiple peers. For every one-to-many message, for every recipient, the procedure has to be completed.

Example

```
import os, gnupg

gpg = gnupg.GPG(homedir="~/.gnupg")
key = os.urandom(16)

header = [
  0x01,                   # version(1)
  0xDE, 0xAD, 0xBE, 0xEF, # source UUID
  0x00, 0x0B, 0x00, 0xB5, # destination UUID
  0x01,                   # packet type (this is a data packet)
  0x10,                   # flags (authentication flag set)
  0x0F                    # hop count (set to 15 initially)
  # payload length will be added later
]

c = gpg.encrypt(key, "DEADBEEF").data
s = gpg.sign(c, default_key = "000B00B5", passphrase = "mypass").data

n = 87 # payload length
payloads = [s[i:i+n] for i in range(0, len(s), n)]

headers = [header[:] for i in range(0, len(payloads))]
for i, header in enumerate(headers):
  if (1 == (i % 2)):
    headers[i][10] |= 0x02 # add alternating sequence number
  if ((i + 1) < len(headers)):
    headers[i][10] |= 0x01 # indicate that there’s more to come
  headers[i] = ''.join([chr(bytes) for bytes in headers[i]])
```

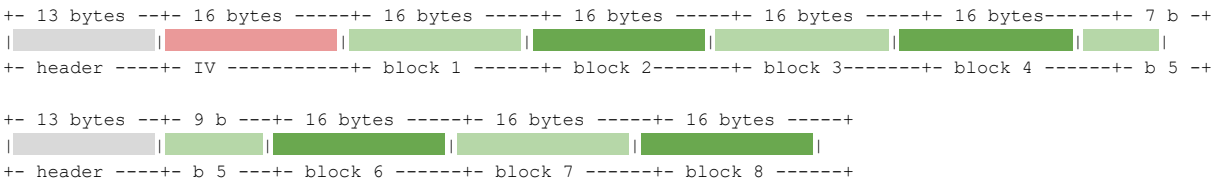
```
packets = [hdr + chr(len(pld)) + payload for hdr, pl in zip(headers, payloads)]

transmit(packets)
```

Encryption

All end user messages and files shall be encrypted using AES in CBC mode with PKCS#7 padding (symmetric encryption). The 128-bit session key shall be used as a . Each separate message shall use its own initialization vector (hereinafter IV).

Example

Here's how a message of 128 bytes would look like (assuming 13 byte header). Notice how the first 16 bytes are used for the IV. Also take notice at block 5 which is split up into 2 packets, first one containing 7 and the second one containing 9 bytes.



Chat

Chat message format is plain ASCII string (encrypted). **No control message for chat init exists, therefore chatting starts with just sending data messages with type text message.** Data messages with type text message are used to deliver the payload.

Example

```
# inspired by http://stackoverflow.com/a/12525165/766120
from Crypto.Cipher import AES
from Crypto import Random

BS = 16
pad = lambda s: s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
unpad = lambda s : s[:-ord(s[len(s)-1:])]

# for session key negotiation, please look at previous chapters
key = getSessionKeyByUUID("000B00B5")
msg = "This is my super secret message!"

iv = Random.new().read(AES.block_size)
cipher = AES.new(key, AES.MODE_CBC, iv)
payload = iv + cipher.encrypt(msg)

header = [
  0x01,                   # version(1)
```

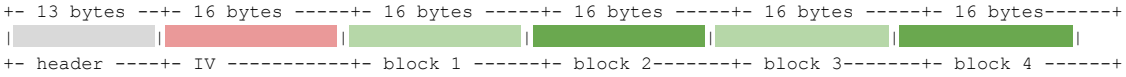
```
  0xDE, 0xAD, 0xBE, 0xEF, # source UUID
  0x00, 0x0B, 0x00, 0xB5, # destination UUID
  0x01,                   # packet type (this is a data packet)
  0x05,                   # flags (text flag 0x04 and last packet flag 0x01)
  0x0F,                   # hop count (set to 15 initially)
  len(payload)            # example payload fits in 1 packet
]

header = ''.join([chr(byte) for byte in header])
packet = header + payload
```

File transfer

File message format is plain data (encrypted). Control message file transfer init is used to start sending the data. Data messages with type file content are used to deliver the payload.

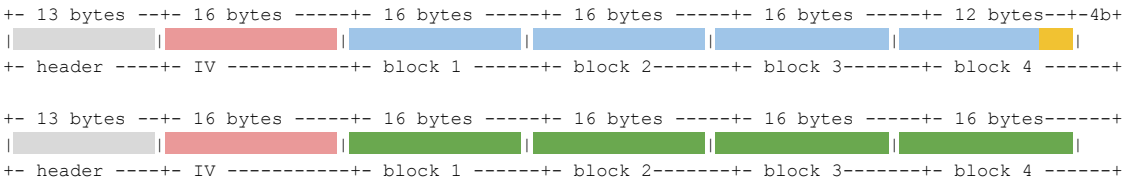
Since we need a bit extra information for file transfer, (name of the file and length basically), and since the major priority is to not change the protocol to much, adding these meta data in the payload together with the file data make sense, so no need of extra control messages or flags.

Using the example from before



The first package will be destinated to meta data, so first 16 bytes is the vector initialization, next 60 bytes for file name and 4 bytes for file length, next package will contain only file data.

The implementation only require to append the filename and length to the payload in the beginning, with the only difficulty to add extra empty bytes if the name require less than 60 bytes, or cut the name in case this one is longer than 60 bytes (keeping the extension of it)



File name
File Length
File data

Encryption procedure

Sending

- Encrypt the full text/file
- Segment the encrypted text/file

- Send segments via stop and wait

Receiving

- Receive all segments
- Reconstruct full payload
- Decrypt payload to get original text/file

Chatting with multiple peers

There is no need for group chats (referencing to the evaluation criteria document). A peer must be able to send one specific message to all peers in its neighbour table. The response (if any) is an unicast to the sender.

Implementation notes

References

[1] https://tools.ietf.org/html/rfc4880#section-12.2