P2P-VoIP Application Project

Peer-to-Peer Systems and Security

SS2015
Technical University of Munich

VoIP Module

Interim Report

Group20

Team Members
Sri Vishnu Totakura - 083658427
Anshul Vij – 03645942

4th June 2015

1 Architecture

1.1 Multi-Threaded Architecture

The proposed application architecture consists of a multi-threaded environment. Each thread is responsible for handling a sequence of events and generating new events. Fig. 1 depicts the proposed multi-threaded architecture for VoIP module by designed by us.

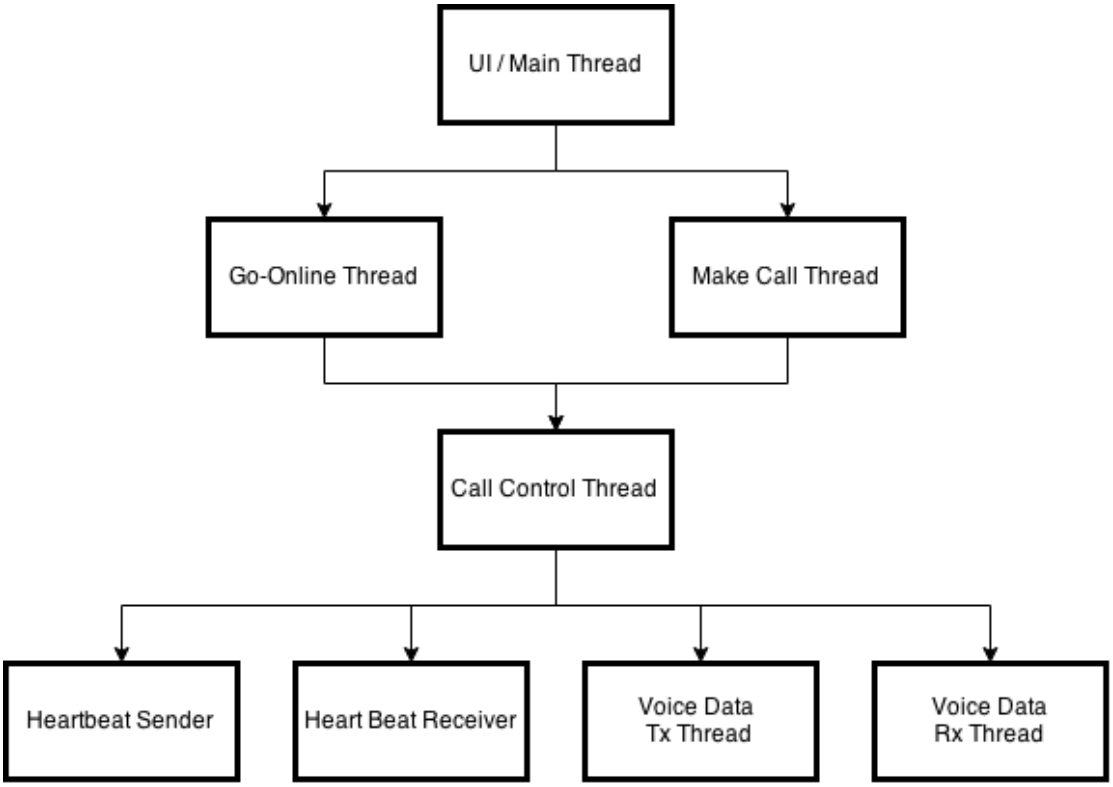

Fig. 1: Multi-threaded architecture

The proposed architecture scheme has following six threads each with different set of responsibilities:

1. **UI / Main Thread:** This is the main application thread controlling the GUI and receiving inputs from the user and providing desired outputs.

2. **Go-Online Thread:** This is the bootstrapping thread of VoIP module and it will have responsibility of finding a random exchange point, announcing this exchange point as incoming interface for this pseudo-identity by putting this information signed by its PKI private key which can be verified using this Pseudo-Identity's PKI public key. The public key will form the key for DHT put request for exchange point information. After this information is put to DHT module this thread will request the KX module to build an incoming onion tunnel

from the exchange point found in previous steps. Once the tunnel is built this thread will listen to any incoming TCP traffic at the TUN interface returned in the previous step. Once this listening is setup it provides asynchronous callback to UI thread to set the status to "Online".

3. **Make Call Thread:** This thread is responsible for triggering an outgoing call to a remote peer. It will be created once the user enters the pseudo-identity of a remote peer and presses the make call button. This thread will search for the exchange point information published by the remote peer via a DHT-GET message. If the information is found then it is verified that it was actually published by the remote peer by validating the signature with its public key. Once validation is successful this thread will ask the KX module to build an outgoing onion tunnel to the exchange point information it received.
   After successful creation of this tunnel this thread will send VoIP module specific message to ask the Go-Online thread of remote peer that it wants to call it. If the remote peer replies with an accepted message, then the make-call module of the host and go-online module of remote peer will exchange a common symmetric key using "Diffie–Hellman" key exchane mechanism.
   Once this key is created the next message is to start the data transmission threads at both the host and remote peer system.

4. **Call Control Thread:** This thread will then invoke two separate threads to receive and transmit the voice data. Also this will invoke two more threads to send and receive heartbeat data to keep the connection alive. If heartbeat is not received within a timeout then call will be dropped.

5. **Heartbeat Sender Thread:** This thread will be responsible for sending heartbeats ping messages. Messages will contain incremental sequence ID to safeguard from replay attacks by any attacker.

6. **Heartbeat Receiver Thread:** As receiving TCP messages is blocking call therefore we use separate thread for receiving a heartbeat. This will wait for a timeout before it declares that the connection is not valid any more and application can stop the transfer of voice data packets through the UDP channel.

7. **Data Tx Thread:** The instances of this thread at both the host and remote peer side will be responsible for accessing the microphone input of the host(caller) and the remote peer(callee) and encoding them into a suitable format and transmit in the form of bytes via UDP packets sent over the onion tunnel interface made by "make call" and "go online" threads.

8. **Data Rx Thread:** The instances of this thread at both ends will listen to the data sent by the "data Tx thread" at the other end. Once it receives the data it will decode it and play it via the speakers at both host(caller) and remote peer(callee) system.

1.1 Component Level Architecture

This section provides the proposed component level architecture of our VoIP module design. The top level components of our application consists of the following:

1. **UI Component:** UI component will display the main screen of application with a list of previously called peers, as well as the option to call a new peer. It will also display the in-coming/out-going call dialog and in-call screen.

2. **Pseudo Identity Manager:** This component will be responsible for generating pseudo identity for the current host.

3. **Session key manager:** This component will be responsible for exchanging the symmetric key using "Diffie-Hellman" mechanism.

4. **DHT wrapper:** This wrapper component will be used to connect to the local DHT module via TCP sockets. It will also provide an easy interface for other components taking care of marshaling and unmarshalling of the DHT messages.

5. **KX wrapper:** This wrapper component will be used to connect to the local KX module via TCP sockets. It will also provide an easy interface for other components taking care of marshaling and unmarshalling of the KX messages.

6. **Encryption Manager:** It will provide interface for encryption and decryption of messages/data via RSA (asymmetric) and AES (symmetric) encryption algorithms.

7. **Protocol Manager:** This component will be responsible for easy creation of JSON protocol messages and also to extract relevant details from incoming message.

8. **Call Controller:** This component will be responsible for establishing and controlling the voice call. It will maintain the call state machine for initialization, exchange of session keys, transmission of control signals, initialization of voice transmission, liveliness and call termination.

9. **Voice Recorder:** This component will continuously record audio from input device such as microphone and encode it into suitable format to be streamed on to the remote peer.

10. **Voice Player:** This component will receive encoded audio message from remote peer and will be responsible for decoding and playback of the received audio stream.

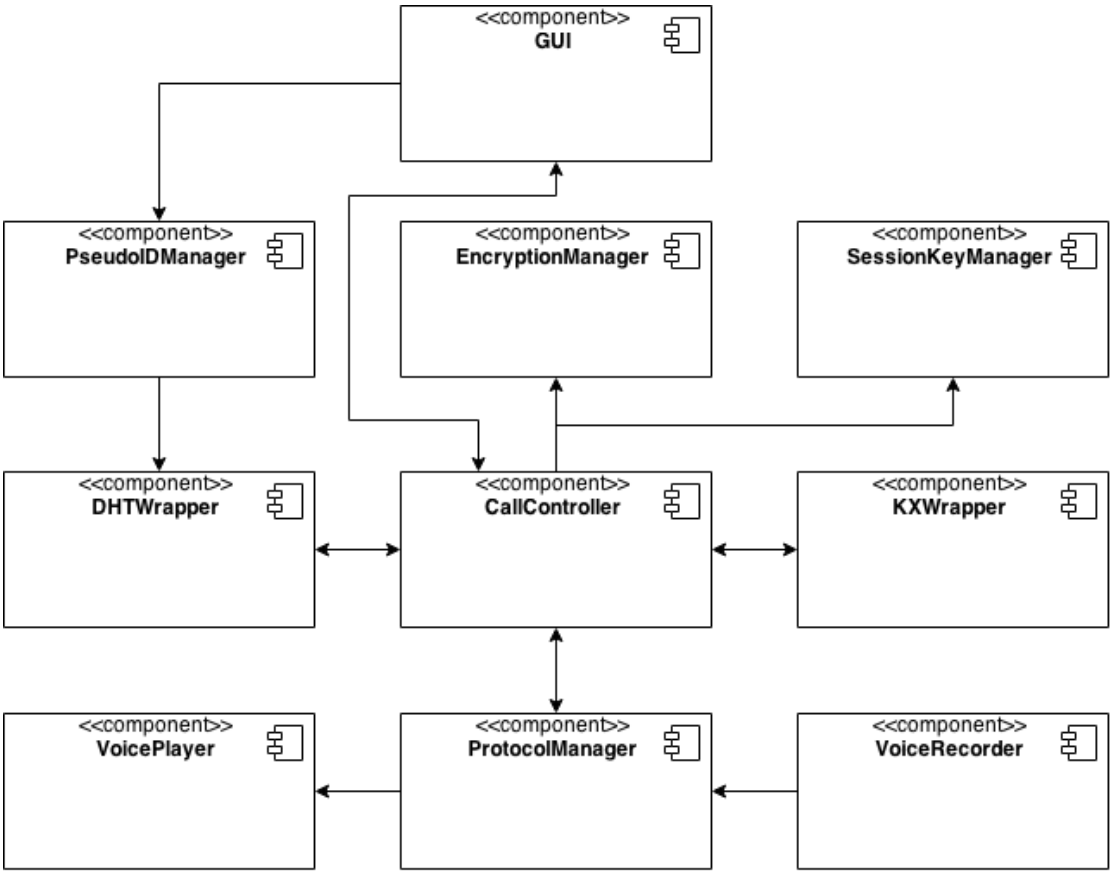

Fig. 2: Component level module architecture

2 Inter-modular Communication

The VoIP module will be built on a protocol that uses JSON structured messages for the inter-modular communications like call initiation, termination etc. Detailed description of the protocol can be found in the following sections of this document.

2.1 Protocol

The inter-modular communication is triggered when a peer wants to call a remote peer. The whole connection flow between the VoIP modules on the two peers follows protocol described in the following section.

2.1.1 Basic Protocol

The basic protocol consists of a sequence of message exchange between the VoIP modules at caller and callee end.

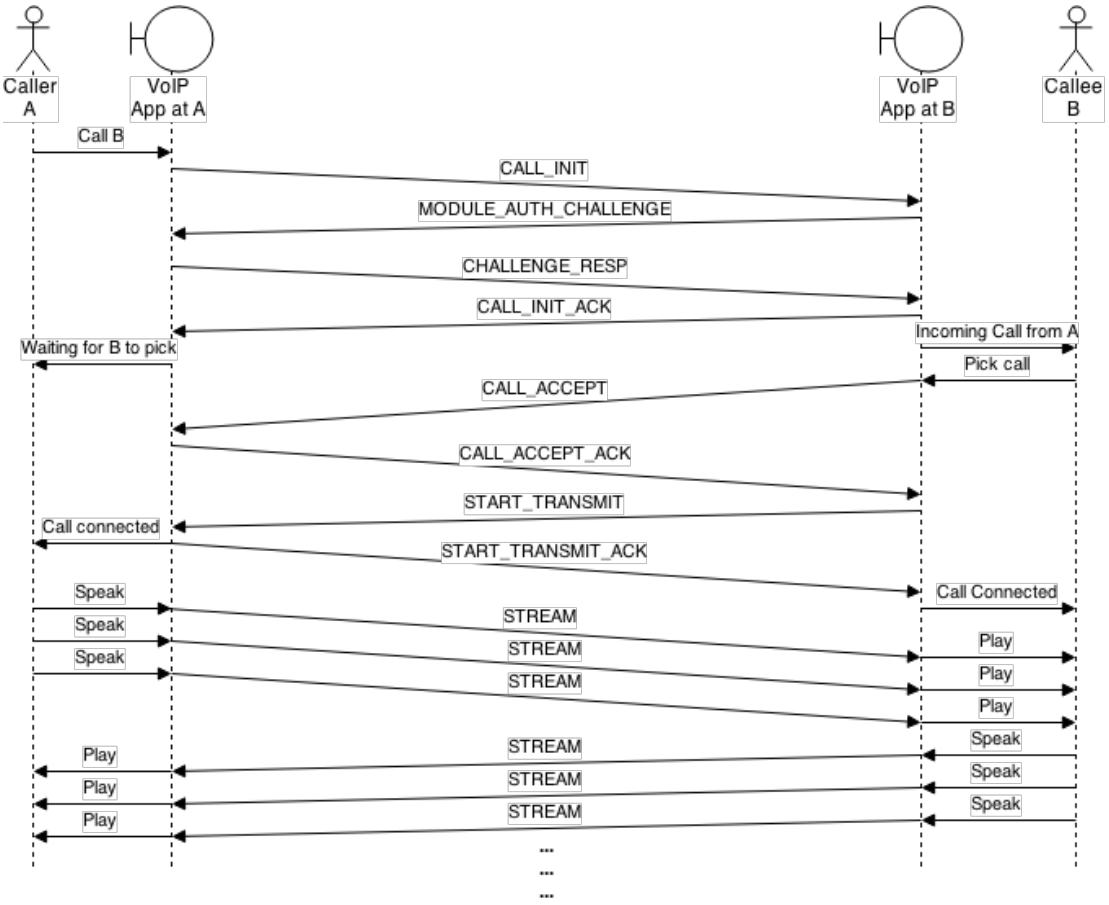

Fig. 3: Basic inter-modular protocol

1. Caller sends "CALL_INIT" message to Callee along with a challenge to authenticate the VoIP module on the callee.

2. Callee would respond with "MODULE_AUTH_CHALLENGE" message with the challenge response and a new challenge for caller to authenticate its VoIP module.

3. Caller responds with "CHALLENGE_RESP" message with the challenge response.

4. After successful verification of the challenge response from caller, callee will acknowledge the "CALL_INIT" with a "CALL_INIT_ACK".

5. Caller then waits for a timeout before the end user at callee accepts the call via UI interaction.

6. Callee accepts call with a “CALL_ACCEPT” along with Diffie-Hellman values for generating a symmetric session key.

7. Caller responds with “CALL_ACCEPT_ACK” and his Diffie-Hellman key values. At this stage, both parties will have a symmetric session key generated.

8. Callee then starts listening for audio stream and sends a “START_TRANSMIT” message.

9. Callers will then also start listening for audio stream and sends a “START_TRANSMIT_ACK”. At this stage, caller can already send audio to callee as it has already started listening.

10. Caller and Callee will continuously send “STREAM” messages containing the voice data that are encrypted with the generated session key.

11. “HEART_BEAT” messages are exchanged at preset intervals, so that the parties know that the other party is active. No “HEART_BEAT” from other party within a preset timeout will cause termination of the call.

2.1.2 Alternate flows:

1. Call Decline: Callee responds with a “CALL_DECLINE” message.

2. Busy : Callee responds with a “BUSY” message.

2.2 Message Formats

The following section describes the message structure for VoIP module inter-modular messages.

2.2.1 CALL_INIT

Sent by the caller to the callee to indicate his will to start communication. Caller also would like to give a challenge to authenticate callee’s VoIP module. A random number is generated and sent for the callee to reply with it ensuring the caller that the reply is not a replay attack. Caller also signs the whole message with the public key for the callee to verify its authenticity.

```
{
    "message": {
        "type": "CALL_INIT",
        "sender": "A's pseudo identity",
        "receiver": "B's pseudo identity",
        "random": (some random number generated by A),
```

```
        "challenge": "same random number",
    },
    "signature": "A's signature of the whole `message` section"
}
```

2.2.2 PEER_BUSY:

Callee would reply with PEER_BUSY if in case it is busy to accept calls from a peer. This could happen which setting up a call for another peer.

```
{
    "message": {
        "type": "PEER_BUSY",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "randomAck": (same random number received from A for CALL_INIT)
    },
    "signature": "B's signature of the whole `message` section"
}
```

2.2.3 MODULE_AUTH_CHALLENGE

Sent by the callee to the caller to send a challenge to authenticate the caller's VoIP module while verifying its authenticity with the challenge response for the caller's challenge form CALL_INIT message.

```
{
    "message": {
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "type": "MODULE_AUTH_CHALLENGE",
        "random": (some random number generated by B),
        "challangeResponse": "Response to the challenge from A",
        "challenge": "same random number generated by B",
    },
    "signature": "B's signature of the whole `message` section"
}
```

2.2.4 CHALLENGE_RESP

Caller responds to callee's MODULE_AUTH_CHALLENGE. It authenticate's caller's VoIP module for the callee.

```
{
    "message": {
        "type": "CHALLENGE_RESP",
        "sender": "A's pseudo identity",
        "receiver": "B's pseudo identity",
        "challengeResponse": "Response to the challenge from B",
    },
    "signature": "A's signature of the whole `message` section"
}
```

Note that the message in this stage won't have a random number because the replay attacks won't be valid as the challengeReponse will be for a fresh challenge which was sent in the previous message.

2.2.5 CALL_INIT_ACK

Informs the caller that the callee has verified and acknowledged the CALL_INIT from caller. At this stage caller can consider that callee's VoIP is in Ringing state. The message is sent with randomAck which is the same random number sent my caller in CALL_INIT to ensure that the message is not a replay attack.

```
{
    "message": {
        "type": "CALL_INIT_ACK",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "randomAck": (random number which is received from A for CALL_INIT)
    },
    "signature": "B's signature of the whole `message` section"
}
```

2.2.6 CALL_DECLINE

Callee would reply with a CALL_DECLINE in case if the call is chosen to be declined.

```
{
    "message": {
        "type": "CALL_DECLINE",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "randomAck": (same random number received from A for CALL_INIT)
    },
    "signature": "B's signature of the whole `message` section"
}
```

2.2.7 CALL_ACCEPT

Callee informs the caller that the call is accepted along with the Diffie-Hellman values for generating a session key. The message is sent with randomAck which is the same random number sent my caller in CALL_INIT to ensure that the message is not a replay attack.

```
{
    "message": {
        "type": "CALL_ACCEPT",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "randomAck": (random number which is received from A for CALL_INIT)
        "DHValues": "Diffie-
Hellman variables from B. includes, p, g and (g^a mod p)"
    },
    "signature": "B's signature of the whole `message` section"
}
```

2.2.8 CALL_ACCEPT_ACK

Caller would respond with a CALL_ACCEPT_ACK along with its generated Diffie-Hellman value. At this stage caller will have generated the symmetric session key. Also, the callee will have the same session key generated upon receipt of this message.

```
{
    "message": {
        "type": "CALL_ACCEPT_ACK",
        "sender": "A's pseudo identity",
        "receiver": "B's pseudo identity",
        "randomAck": (random number which is received from B for MODULE_AUTH
_CHALLENGE)
        "DHValues": "Diffie-Hellman variables from B. (g^b mod p)"
    },
    "signature": "A's signature of the whole `message` section"
}
```

2.2.9 START_TRANSMIT

Sent by the callee when the session key is generated and has started to listen for incoming audio stream. Callee send a random number to assure the caller of a fresh message.

```
{
    "message": {
        "type": "START_TRANSMIT",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
       "randomAck": (random number which is received from A for CALL_INIT)
    },
    "signature": "B's signature for the whole `message` section"
}
```

2.2.10 START_TRANSMIT_ACK

Caller will start listening for incoming audio stream after START_TRANSMIT is received and send the ACK reply to indicate that its is now listening for stream. Caller can already start sending data as the callee has started listening for incoming data. This ACK is replied with the random number from callee for providing replay protection.

```
{
    "message": {
        "type": "START_TRANSMIT_ACK",
        "sender": "A's pseudo identity",
        "receiver": "B's pseudo identity",
        "randomAck": (random number received from B for MODULE_AUTH_CHALLENG
E)
    },
    "signature": "A's signature for the whole `message` section"
}
```

2.2.11 STREAM

Once the both peers know that the other is listening, they start streaming the audio data using within a STREAM. The STREAM message carries a certain chunk of the audio encrypted with the session key. Every STREAM message carries a seqNo which is used to prevent replay attacks.

```
{
    "message": {
        "type": "STREAM",
        "sender": "sender's pseudo identity",
        "receiver": "receiver's pseudo identity",
        "seqNo": "sequence number for stream block"
        "data": CIPHER[A,B](data)
    },
    "signature": "senders's signature for the whole `message` section"
}
```

2.2.12 HEART_BEAT

Both peers need to know that the other peer is active and is listening and receiving the stream messages. This is achieved by a HEART_BEAT message sent by each peer to the other in certain intervals to prove its presence. No HEART_BEAT message from the other peer in a timeout interval would cause the call to disconnect. These messages are replay protected by using a seqNo

```
{
    "message": {
        "type": "HEART_BEAT",
        "sender": "sender's pseudo identity",
        "receiver": "receiver's pseudo identity",
        "seqNo": "sequence number for HEART_BEAT block"
    },
    "signature": "senders's signature for the whole `message` section"
}
```

2.2.13 CALL_DISCONNECT

Either of the peers can choose to disconnect the call by sending a CALL_DISCONNECT

```
{
    "message": {
        "type": "CALL_DISCONNECT",
        "sender": "A's pseudo identity",
        "receiver": "B's pseudo identity",
        "randomAck": (random number which is received from B for MODULE_AUTH
_CHALLENGE)
    },
    "signature": "A's signature for the whole `message` section"
}
```

2.2.14 CALL_DISCONNECT_ACK

CALL_DISCONNECT_ACK will acknowledge a peer's request to disconnect the call.

```
{
    "message": {
        "type": "CALL_DISCONNECT_ACK",
        "sender": "B's pseudo identity",
        "receiver": "A's pseudo identity",
        "randomAck": (random number received from A for CALL_INIT)
    },
    "signature": "B's signature for the whole `message` section"
}
```

2.3 Module authentication

The VoIP module will have a hardcoded value which serves as a salt for generating hash of a challenge. When a challenge is received, the challenged peer responds with the SHA2 hash of the challenge appended to the salt. The challenging peer would calculate the same hash and compares the response to verify the authenticity of the module. Both peers also sign these message exchanges.

2.4 Random numbers

We will use Java's SecureRandom class, which is available from *java.security* package to generate cryptographically strong random numbers.

2.5 Protection from Common Attacks

2.5.1 Replay Protection

Random numbers are used during start of communication which are to be included in the signed replies ensuring that the messages are not replayed.

Also, as the application has different states in it to receive messages, it would discard any messages of kind that are not expected by it at a given moment.

2.5.2 Eavesdropping

Session key will encrypt the voice data so that no malicious peer in the path could eavesdrop on the communication. However, the call control messages like CALL_INIT, CALL_ACCEPT are not encrypted because the communication is anyway expected to happen via a secured onion tunnel. The voice data encryption is just an additional security considering the sensitivity of this data.

2.5.3 Storing messages

Storing messages cannot be prevented by this protocol. However, it provides perfect forward secrecy with the use of freshly generated session keys and makes the storage of messages useless unless a session key for that messages is somehow broken or compromised.

2.5.4 Dropping messages

Protocol cannot prevent malicious peers from dropping the messages between two peers. Since, timeouts will be implemented during communication, a non-responding communication will be dropped and a new connection will be established, through a different tunnel or path.