CIS 505 Software Systems
Spring 2016: Final Report
Due: Sunday, April 24, 11:59pm

JAM Team Members:
Bipeen Acharya (acharyab)
Hung Nguyen (hungng)
Krzysztof Jordan (jordankr)

Overview

Project code structure

/code
- /bin: executable files
- /build: temporary CMake cache
- /include: project header files
- /src: project sources files
- /test: project test files

Requirements

The project requires BOOST library with minimum version of 1.50 and (optional) OpenSSL library.

Compile and Run instructions

To compile JAM, execute the following commands (executables are built in /bin folder):

```
cd code/build
cmake ..
make speclab
make speclab-secure
```

In the submission, we already have cmake exported files in /build, so you can omit running cmake again. However, running cmake to rebuild the cache is recommended.

Running parameters are followed the project description. Executables include:
- dchat: normal executable
- dchatd: normal executable with printed debug data
- dchats: secure executable
- dchatds: secure executable with printed debug data

Configurable parameters

```
#define DEFAULT_INTERFACE           "em1"   // Default UDP interface
#define DEFAULT_PORT                "9346"  // Default UDP datagram port
#define MAX_UDP_BIND_RETRIES        10      // # of retries for different port to bind
#define MAX_MESSAGE_LENGTH          256     // Maximum message length per payload
#define MAX_USER_NAME_LENGTH        20      // Maximum displayed user name length
#define MAX_BUFFER_LENGTH           512     // Maximum UDP socket buffer length
#define MAX_HOLDBACK_QUEUE_LENGTH   100     // Maximum length of hold back and history queue
#define UDP_RECEIVER_QUEUE_SIZE     100     // # of payload kept track to prevent duplicate
#define NUM_UDP_RETRIES             2       // Default number of resend before notify crash
#define UDP_TIMEOUT                 2000    // Timeout before trying resend UDP payload in ms
#define ACK_MONITOR_INTERVAL        1       // Ack check interval in seconds
#define JOIN_TIMEOUT                10000   // Timeout to join chat group in milliseconds
```

Internal/Network communication payload

Payload is the single type of object being transported within all the modules of JAM. There are 5 kinds of payloads: message payload, status payload, election payload, recover payload and acknowledge payload - which is set by corresponding flag.
Payload is encoded and decoded into byte stream at UdpWrapper layer; and this byte stream is the content to be transferred over the network. The encoded format for each type is as follows:

- CHAT_MSG:

| 1 byte message type | 4 bytes UID | 4 bytes order | 4 bytes username length *u* | 4 bytes message length *m* | *u* bytes username | *m* bytes message |

- STATUS_MSG:

| 1 byte message type | 4 bytes UID | 4 bytes order | 1 byte status code |

- ELECTION_MSG:

| 1 byte message type | 4 bytes UID | 4 bytes order | 1 byte election code |

- RECOVER_MSG:

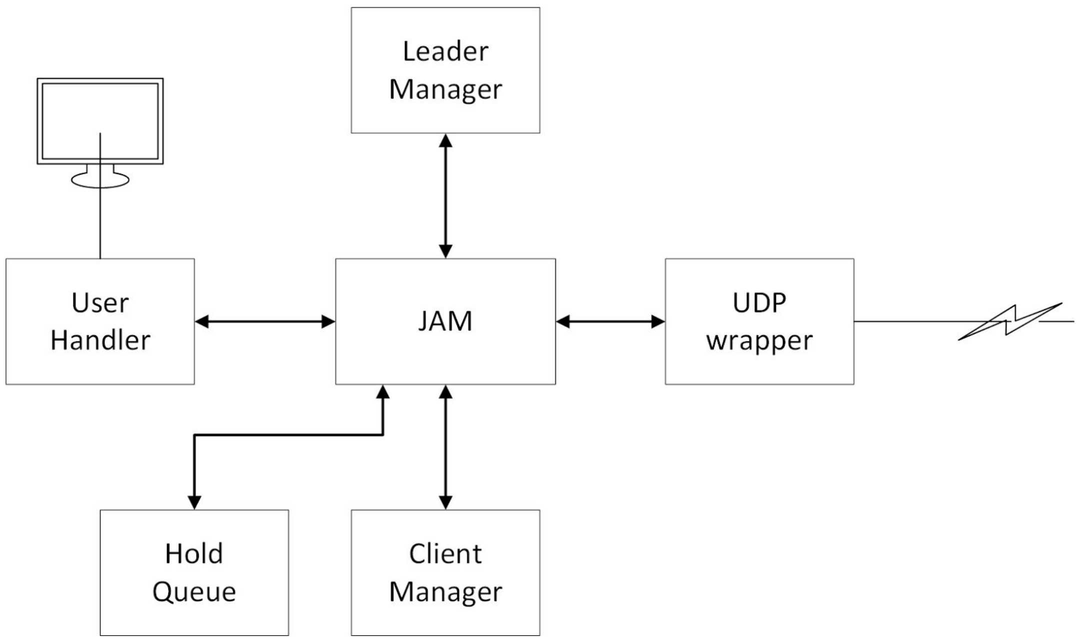
| 1 byte message type | 4 bytes UID | 4 bytes order | 1 byte recover code |

- ACK_MSG:

| 1 byte message type | 4 bytes UID | 1 byte ack code |

PROJECT DETAILS

As mentioned in the First Milestone report, we have divided our distributed chat server into a number of modules (as illustrated in Figure 1) that communicate with one another to facilitate the chat server.



Here, we describe the final modules we have implemented:

1. **Client Info**

The ClientInfo module stores each client as a Client Info object (which stores a sockaddress, username and whether the client is a leader). Client Info was mainly implemented to abstract functionality of a client so that Client Manager could maintain a list of Client Info objects.

```
API:
get_sock_address() -returns sock address of a client
get_username() - gets the username of a client
is_leader() - whether the client is a leader
set_leader() - set the client to be leader
EncodeClientInBuffer - Encode a client to a buffer (this is used by ClientManager to
encode the entire client list)
```

2. **Client Manager**

The Client Manager is responsible for maintaining a list of clients and handling functionality related with adding or removing clients to the list. It also handles functionality like encoding the client list into a buffer to send to other modules (JAM), decoding a buffer of information into client list (used when client list is updated) and getting higher order clients to send to Leader Manager for elections. We outline some of the important functionality of the Client Manager:

```
AddClient(ClientInfo cli ) - Adds a ClientInfo object to the client list
AddClient(sockaddr_in client, std::string username, bool isLeader) - Adds a client
based on its sock address, username, and whether or not it is a leader.

RemoveClient(ClientInfo client, std::string *username) - Removes a ClientInfo client
RemoveClient(sockaddr_in client, std::string *username) - Removes a client from its
sockaddress.

EncodeClientList() - Encodes the client list into a string to send as a buffer
DecodeClientList() - Decodes a buffer into a client list

GetHigherOrderClients() - Used for leader election
GetClientCount() - Returns the number of clients in the list
```

3. **Leader Manager**

Leader Manager handles functionality that deals with the leader and holding elections. The leader also deals with starting and stopping heartbeat signals to clients. Some of the functionality of the Leader Manager are:

```
StartLeaderHeartbeat() - Start a heartbeat signals to all clients.
StopLeaderHeartbeat() -
HandleElectionMessage(Payload msg) - Switches on different election command messages to
hold elections and declare leader.
```

4. **Serializer Helper**

This is just a helper class that consists of method definitions to pack and unpack data to help with encoding and decoding different kinds of payloads, and client information.

5. **Hold Queue**

The Hold Queue class implements a hold back queue that allows the chat server to send chat messages to clients in sequential order. If a payload seems to have been lost due to the network, it queries the program and recovers the lost messages. Similarly, duplicate messages are prevented from being sent as well.

```
API:
AddMessageToQueue: Adds the message to delivery queue and calls ProcessPayloads.
ProcessPayloads: Processes and sends messages to UserHandler if they are in order.
GetPayloadInHistory: Returns payloads that have been sent from a history queue
```

6. **User Handler:**

The User Handler module processes input from the user in the form of chat messages, passes them to JAM, and sends acknowledgement. It listens for incoming messages and messages arrive, it sends it to the Central Queue for delivery. It also listens for messages that arrived over the network and need to be displayed to the client.

```
API:
Start() - Starts a boost thread to handle input
HandleInput() - Gets input from user and handles it as necessary
```

7. **UDP Wrapper**

UDP Wrapper module acts as a low-level I/O handler to send/receive message over UDP datagram reliably. UdpWrapper makes sure that sent messages are delivered with proper acknowledgement or crash notification will be distributed. It also makes sure that messages that couldn’t be delivered due to leader crash will be re-attempted for delivery through a new leader. It has 3 threads running simultaneously as described follows:

- UdpReader: is responsible for handling all the incoming messages from UDP socket - including verifying the correctness (and duplicate), sending acknowledgement, decoding message to payload object and pushing payload to internal queue for JAM.
- UdpWriter: is responsible for sending outgoing payload from its queue to UDP socket - including distributing to multiple targets/single target and uid-stamping each payload.
- UdpMonitor: is responsible for monitoring all the payload sent out by UdpWriter - including retry if not receiving acknowledgement after timeout and notifying client crash after a predefined number of retries by writing to internal communication queue.

```
API:
Start() - Initialize UDP socket and start listening
Stop() - Terminate working threads and exit
SendPayloadSingle() - Put payload to a single receiver to queue
SendPayloadList() - Put payload to list of clients to queue
GetAddressFromInfo() - Convert IP address and port to sockaddr_in type
LeaderRecover() - Check the leader_failed_queue and process
```

8. **Central Queues**

Central Queues is a collection of multiple thread-safe queues used for inter-communication between modules in JAM. It has a single condition variable to wake up JAM module to handle of there is any new object in any of the queues. The various queues it handles are:

- user_out_queue: user messages from User Handler to JAM
- udp_in_queue: incoming message from UDP socket to JAM
- udp_out_queue: outgoing payload from JAM to UDP socket
- udp_crash_queue: crash notification from Udp Wrapper to JAM
- leader_out_queue: commands from Leader Manager to JAM
- history_request_queue: payload history request from Hold Queue to Jam

9. **JAM**

JAM is the main module that integrates all the modules to work together. It provides two public API functions: `StartAsClient()` and `StartAsLeader()` that either try to join a group server or start a new group chat as a leader.

JAM functionalities can be split to 3 main parts:

a. Initialization: during initialization, JAM attempts to start all the internal modules to ready state. It first starts the UDPWrapper and detect network interface address. If starting as leader, JAM adds current interface address as a leader to the ClientManager. If starting as client, JAM requests UDPWrapper to contact server for the current client list. Upon receiving the list, JAM will request UDPWrapper to send join notification to all the clients. Finally, JAM starts LeaderManager heartbeat and moves to Running state. Any error occurred during this state or time-out, JAM will stop and exit.
b. Running: This is an infinite loop that handles/monitors central communication. JAM will monitor the Central Queues and handle any data pushed to the queue in order. This is where all kinds of different payloads are handled. If JAM detect terminate signal is sent from UserHandler, it moves to Exit state.
c. Exit: Upon detecting terminate flag, JAM distributes terminate signal to all internal threads. After all threads terminated or time-out, JAM will stop itself.

Extra Credits

Encryption

In the encryption mode, payload is modified to enable message encryption. We use OpenSSL library with shared encryption key between all clients. The encryption is done by payload object itself by using AES-256-CBC encryption algorithm whenever a message content is set to the object. The decryption is done during the decoding process.
Changes are made in payload_secure.cpp mainly at line 230 and 353.