CS 505 Project 1

Short Overview:

The project is about implementation of the *signed message algorithm* to achieve consensus in presence of Byzantine failures. The algorithm consists of an initiator commander who sends order to all or some lieutenants (in case of malicious). Later the lieutenants exchanges their respective orders received among themselves to reach a consensus at the end of the algorithm. This algorithm has some assumptions like the communication is synchronous, there cannot be less than (faulty + 2) processes in the system. Each process (commander or lieutenant) signs the message they received and forwards the signed message to processes who has not received this message before. After the end of the (faulty+1) rounds all the process decides upon a common decision. The decision should be among the following:

a) All loyal lieutenants obey the same order.
b) If the commander or initiator is loyal, then loyal lieutenants obey the order of commander.

System Architecture:

The system consists of multiple process in distributed environment. There is a commander process which will initiate the algorithm and apart from single commander all the processes are treated as lieutenants. There is a file named ***"hostfile.txt"*** which is shared among all the processes to know about each other. The hostfile contains the hostname of all the machines participating in the algorithm. The communication is duplex for each process and they have their own sender side and receiver side. The algorithm proceeds for (faulty + 1) rounds before making any decision where faulty is the number of the malicious processes. We are assuming the first host mentioned in the hostfile as the commander process. The message exchanged between the processes are signed and verified by each process with their 2048 bit generated private/CA certified public rsa keys. We are using UDP as the transport layer protocol and have implemented the ACK mechanism to make it reliable.

Commander:

Initially in round 0 of the algorithm the commander process will send a signed ordered message to all the lieutenants. To keep the process in synchronisation we are using the timer. Here each round is about 500ms long and the acknowledgement message timer is around 100ms. When the commander sends the message it waits for ACK to be received by the lieutenants. If it receives all the valid acknowledgements from the lieutenants then it doesn't send any more messages and decides on an order. If the commander doesn't receive any ACK or some ACK before the ACK timer expires it re-sends the message to those lieutenants only from which no ACK is received. After round 0 the commander will not send or receive any message from any of the lieutenants and will exit after deciding upon an order.

Lieutenants:

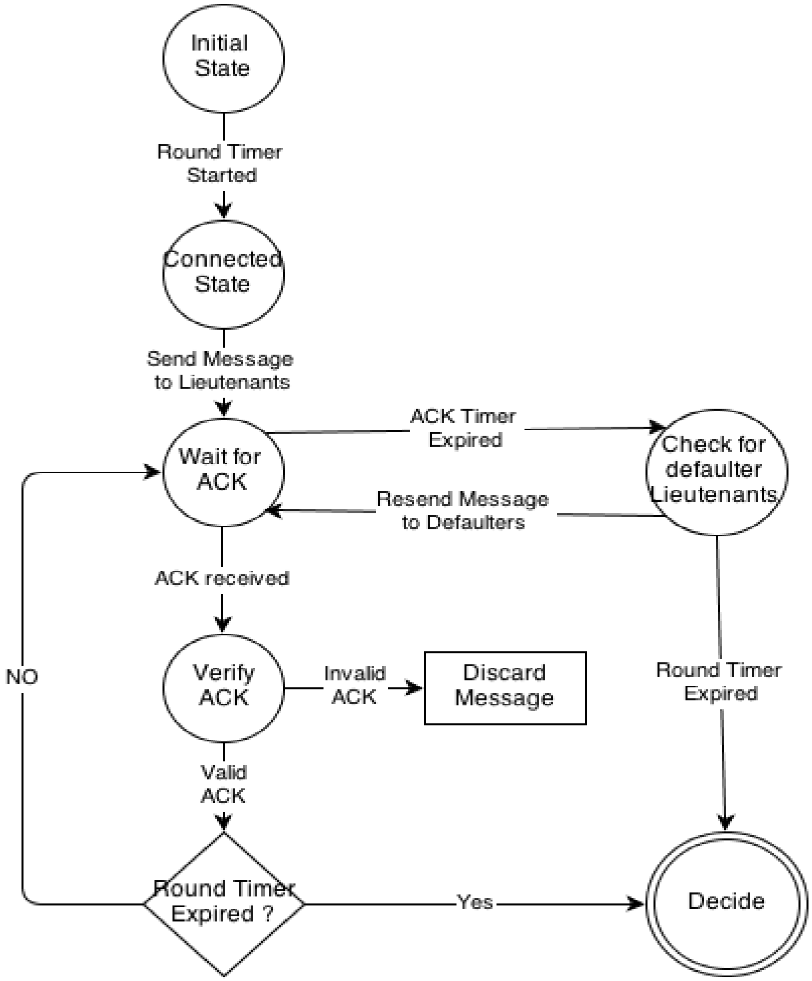
All the lieutenants will wait in round 0 until they receive first message from commander or any other lieutenant. When it receives any signed message from the commander then it validates the received message. During validation it verifies all the signatures inside the message and puts the order received in its order set if the message is valid. If the message is invalid because of any reason such as signature verification failure, past round messages received or invalid order received then it simply discard the message. For each valid or invalid signed message received it sends back an acknowledgement to the sender. It stores the received message if the number of signature in it doesn't exceeds (faulty + 1) to send it to the lieutenants who has not received it yet in the next round. If a lieutenant doesn't receive any message from the commander in round 0 it will remain waiting for a message. Later on receiving the message from future round by any lieutenant it validates and moves itself to that round. If the message is valid it stores it to send it in the next round.

In all the other rounds the lieutenant process will retrieve the signed message stored in the last round and will send it to the other processes (who has not received it before) after signing it with its private key. After sending the message it will start the ACK-timer and will wait for the signed messages from other lieutenants and expected ACK messages. Upon receiving an ACK or signed message it validates both the types of message. If the signed message is valid then it stores it to send in next round if number of signature and value in message condition is satisfied. If the messages is invalid it will simply discard all the message. On ACK-timer expiry the lieutenant resend the signed message to the defaulter lieutenants (who has not send the ACK). It retransmit and receives the message until a round timer expires.

State Diagram:

The state diagram for the algorithm is as below:

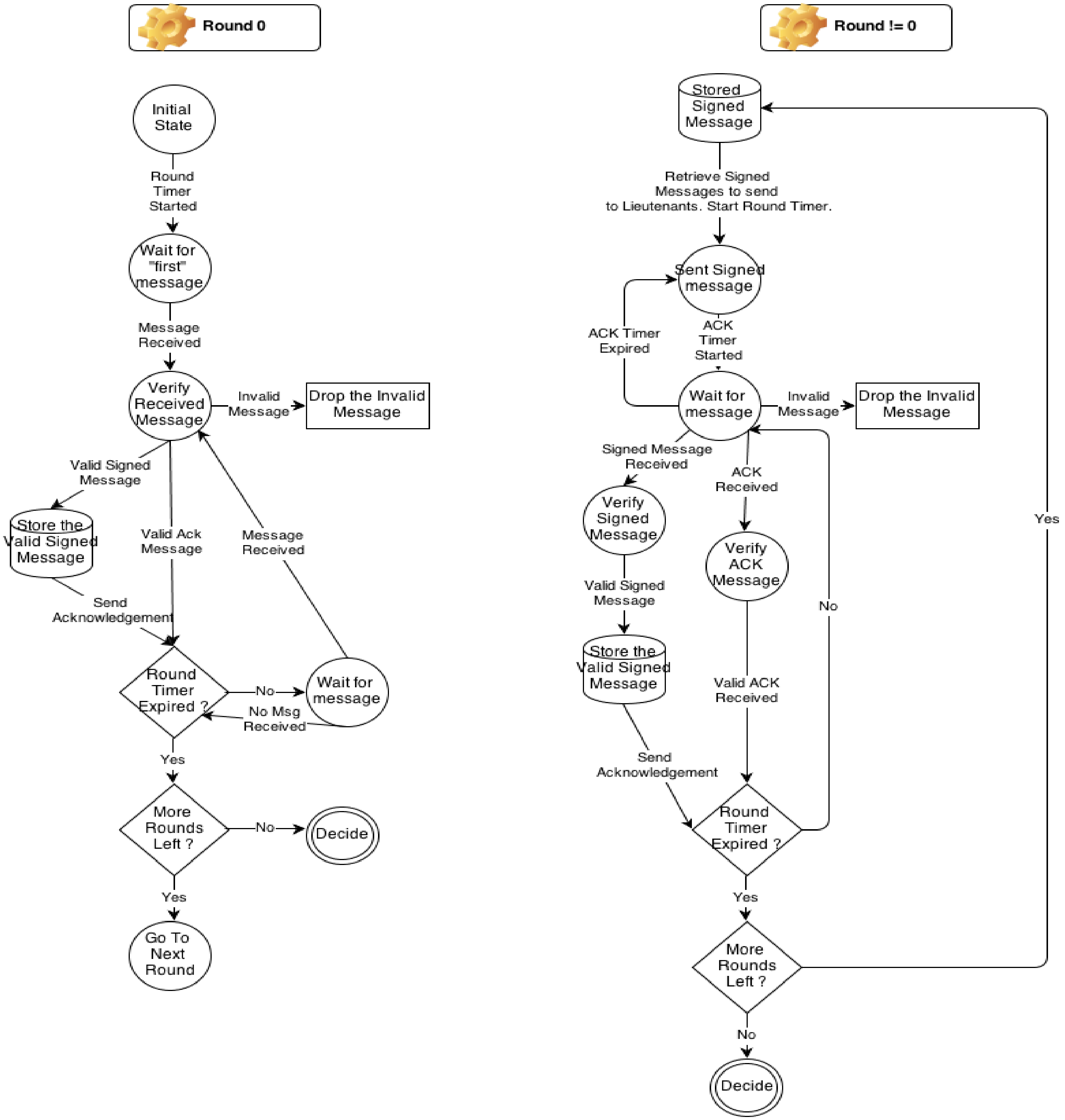
Commander State Diagram



Defaulter Lieutenants are those who have not send the ACK's

Lieutenant State Diagram:

Lieutenants State Diagram



Design Decisions:

The program accepts command line arguments to configure the listening port specified by “-p” option, “-h” to specify the location of the hostfile, “-f” to specify the number of faulty processes, “-c” to switch off the verification of the signed message “-o” to specify the order. The process for which “-o” option is specified will be considered as the commander. In general the first process in the hostfile will be deemed as commander.

Since its a duplex communication there is receiver and sender side of each process. I have made 3 structures such as: ***generalSystemInfo, senderInfo, receiverInfo.***

In structure ***generalSystemInfo*** I am storing the general info which will be used by both the sender and receiver state of a process. It contains information such as a map for hostname to ip address, ip address to id of the process. It keeps track of all the configurable parameters value, round timer and ack timer. It also stores the information about the process hostname and ID. I am also storing private and public keys of all the process inside a map whose keys are process id and value the key value.

In structure ***senderInfo*** I am storing the information related to the sender side of the process. It keeps the sender socket, set containing the ID’s of the processes from which its expecting ACK in a round. A map which stores the message to be send to the processes in the current round with keys as process ID’s.

In structure ***receiverInfo*** I am storing the information related to the received side of the process. It keeps the receiver socket information, listening port number, orders received from all the process (here order can be either attack(1) or retreat(0)) and a map which stores all the valid received signed messages which needs to be sent in the next round with keys as the process ID’s.

I am using a tool named ***pkiTool.sh*** which generates public and private key pairs for all the processes and a Certification Authority (CA). Later it gets the public keys of the lieutenants signed by the CA. The private key files have name in the format ***host_<hostname>_key.pem*** and signed certificate file has the name format ***host_<hostname>_cert.pem***. This tool generated the 2048 bit public and private key which is later used by the processes to sign and verify the messages.

To keep the code modular I have written few methods whose functionalities are re-used. Functions like ***sendAckMessage***: To send an ack message after receiving a signed message. I am sending the ack message independent of the fact that the signed message received is valid or invalid.

verifyAndStoreSignedMessage: This function will verify the signed message received and if its a valid message will then store the message in the map maintained in ***receiverInfo*** structure to send the message in the next round. During verification it checks for all the fields of the Signed Message individually. That is it checks for type to be 1, total_sigs is equal to (round+1) or not (I am starting round from 0, also based on the total_sigs value in the message it checks if the received message length is same as expected message length or not, it check for valid order value (i.e. 0 or 1), it checks for id to be within [1,totalNumOfProcess]. If any of the above condition fails then it discards the message before verifying it for signature. If all the above condition agrees then it verifies the signature present inside the message if crypto is turned on. If in a round any new order is received which is not their inside the order received set maintained by receiver side then it saves that order after verifying all the signatures. Also the message is only forwarded in the next round if it has less than (faulty + 1) signatures.

verifyAckMessage: This function verifies the ack received by a process. If a valid ack is received it updated

the expected ack list in sender side. If an invalid ack is received then it just discards the message.
getTimeDiff: This function helps to get the difference between start and end time of a round as well as ACK timer. Based on the returned value of this function we judge if the ack timer expire or a round has expired or not.

There are few other utility functions too which helps in verifying the command line parameters entered by the user. And few are used to print the message fields.

Implementation Issues:

I am assuming few things which might cause some issues. The assumption are as follows:

a) I am exiting the program if there is failure in opening a socket, binding a socket, opening a file (hostfile or public/private key file), if any mandatory command line parameter is missing, if gethostbyname function fails and if a memory is not allocated by the malloc library.

b) I have kept the processes apart from commander to be blocked until they received first message. In case when none of the lieutenant will received the message then the algorithm will not move ahead.
If say a single lieutenant (L1) is not receiving any message from the commander and other lieutenants have received the message. Then upon receiving a message from any of the lieutenant, it (L1) will accept the message after validation inspite the message is from future round as per L1 state. L1 then moves to the round from which it received the first message and will start non-blocking execution from there.
However if a message in some other round (>0) then it will discard all the messages from the past and future rounds.

c) I am sending acks for all the received signed messages whether it's valid or invalid. So a processes(L1) if receives a message from another process(L2) it will send ack to L2. In this case if message is invalid still the process L2 will not resend the signed message to L1. So in scenario where it's expected that a process should not send ack for invalid signed message, my algorithm can generate different output.

d) I am checking the total_sigs field value in the message and calculating the expected size of the message from that. If the expected size doesn't matches the received message size then I am discarding the message. I am not verifying all the signatures first and then from that getting the count of signature present inside the message. If we follow the second approach then I believe there is no reason to give total_sigs field inside the message structure. This will vary with implementation of different people.

e) I am assuming that the process will not get messages delayed beyond one round. Only in the first round if a process gets its first message from the future round then it will accept that message. The process will then move its round number according to the message received and will restart its timer from there. Due to this there will be always an overlap between the rounds of the processes and the algorithm will work as in a synchronous environment. In all the other round if a process gets the message from future round then it will discard it.

f) I am printing few statements used for my debugging and the outputs are redirected to the ***stderr.***