Byzantine General Problem

Kushal Lakhotia

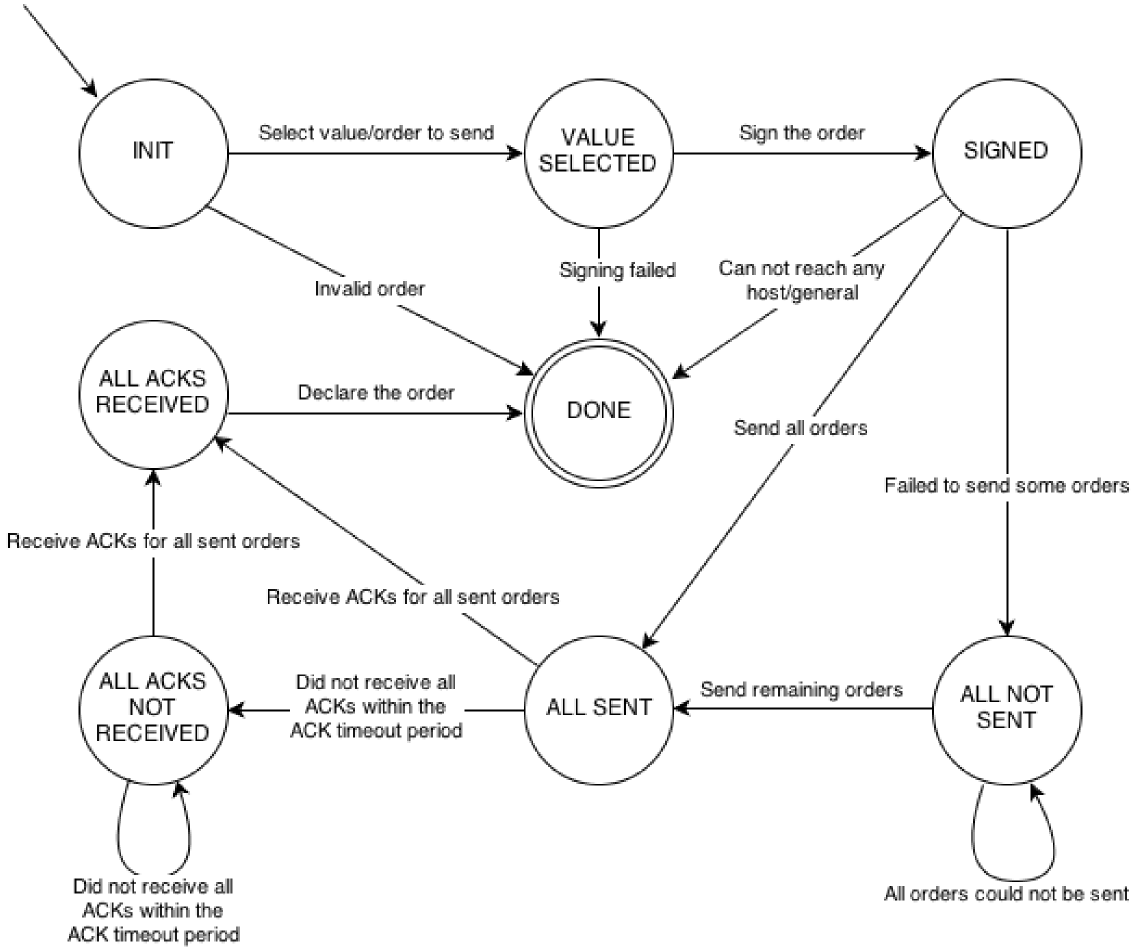
February 14, 2013

Contents

1 State Diagram

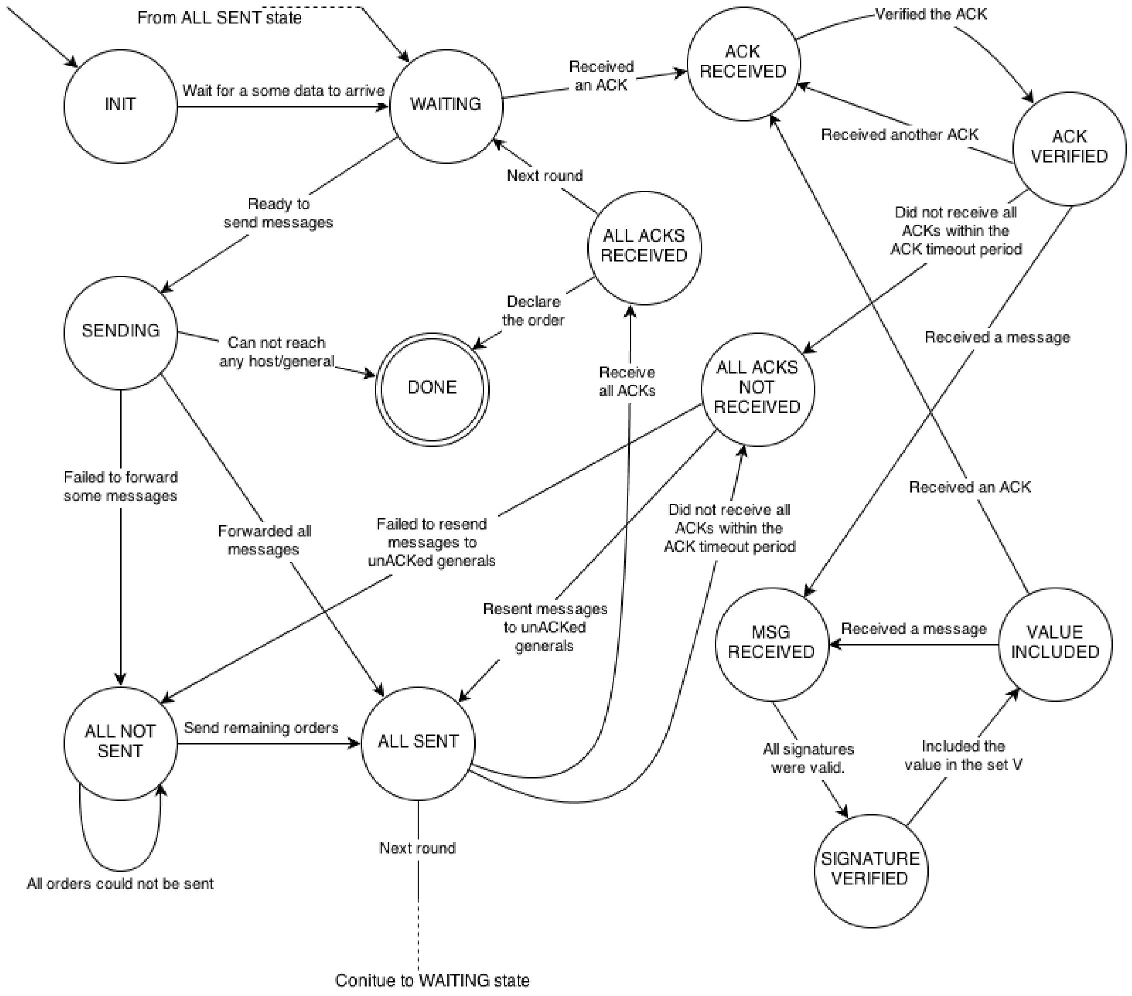
The state diagrams of the Commander and the Lieutenants are given below.

1.1 Commander


INIT
Select value/order to send
VALUE SELECTED
Sign the order
SIGNED
Invalid order
Signing failed
Can not reach any host/general
ALL ACKS RECEIVED
Declare the order
DONE
Send all orders
Failed to send some orders
Receive ACKs for all sent orders
Receive ACKs for all sent orders
ALL ACKS NOT RECEIVED
Did not receive all ACKs within the ACK timeout period
ALL SENT
Send remaining orders
ALL NOT SENT
Did not receive all ACKs within the ACK timeout period
All orders could not be sent

State Diagram of the Commander

1.2 Lieutenant



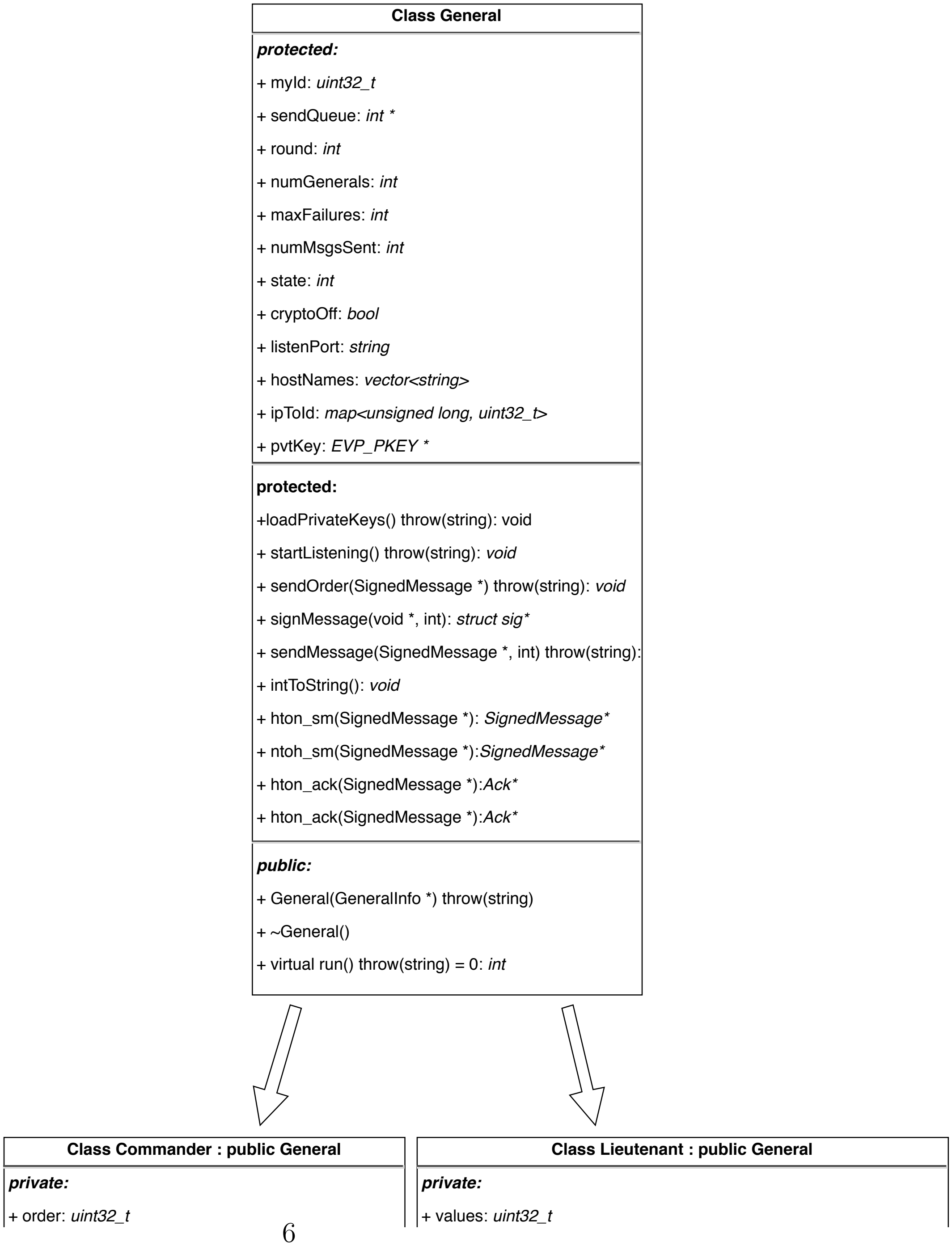
State Diagram of the Lieutenant

2 System Architecture and Design Decisions

The system has been implemented in C++ by the use of three classes and a main function. The base class is `General` which has variables and methods that are common to the actions of both commander and lieutenant. `Commander` and `Lieutenant` classes derive from `General` and contain variables and methods specific to the actions of commander and lieutenant respectively. `General` has one public pure virtual function `run()` that that is overridden by the child classes and is the method that `main()` calls for starting the system.

2.1 Class Diagram

The class diagram shows the various member variables and methods of the classes and the relationship between them. The arrow indicate the *derived from* relationship.



Class General

protected:

+ myId: *uint32_t*

+ sendQueue: *int **

+ round: *int*

+ numGenerals: *int*

+ maxFailures: *int*

+ numMsgsSent: *int*

+ state: *int*

+ cryptoOff: *bool*

+ listenPort: *string*

+ hostNames: *vector<string>*

+ ipToId: *map<unsigned long, uint32_t>*

+ pvtKey: *EVP_PKEY **

protected:

+loadPrivateKeys() throw(string): void

+ startListening() throw(string): *void*

+ sendOrder(SignedMessage *) throw(string): *void*

+ signMessage(void *, int): *struct sig**

+ sendMessage(SignedMessage *, int) throw(string):

+ intToString(): *void*

+ hton_sm(SignedMessage *): *SignedMessage**

+ ntoh_sm(SignedMessage *):*SignedMessage**

+ hton_ack(SignedMessage *):*Ack**

+ hton_ack(SignedMessage *):*Ack**

public:

+ General(GeneralInfo *) throw(string)

+ ~General()

+ virtual run() throw(string) = 0: *int*

Class Commander : public General

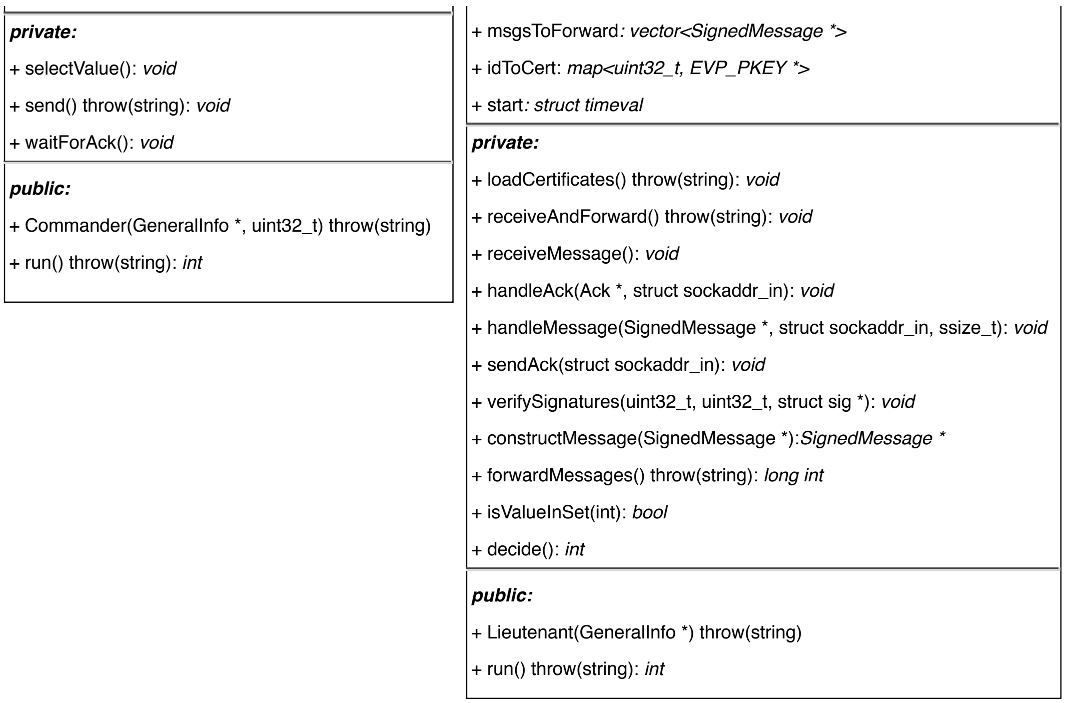
private:

+ order: *uint32_t*

Class Lieutenant : public General

private:

+ values: *uint32_t*



private:

+ selectValue(): void

+ send() throw(string): void

+ waitForAck(): void

public:

+ Commander(GeneralInfo *, uint32_t) throw(string)

+ run() throw(string): int

+ msgsToForward: vector<SignedMessage *>

+ idToCert: map<uint32_t, EVP_PKEY *>

+ start: struct timeval

private:

+ loadCertificates() throw(string): void

+ receiveAndForward() throw(string): void

+ receiveMessage(): void

+ handleAck(Ack *, struct sockaddr_in): void

+ handleMessage(SignedMessage *, struct sockaddr_in, ssize_t): void

+ sendAck(struct sockaddr_in): void

+ verifySignatures(uint32_t, uint32_t, struct sig *): void

+ constructMessage(SignedMessage *):SignedMessage *

+ forwardMessages() throw(string): long int

+ isValueInSet(int): bool

+ decide(): int

public:

+ Lieutenant(GeneralInfo *) throw(string)

+ run() throw(string): int

2.2 Class General

This class implements common actions like opening a port for listening, signing messages, sending messages and some helper functions as shown in the class diagram in **2.1**. The pure virtual function makes sure `Commander` and `Lieutenant` classes **MUST** implement their own ways to starting the run. It also makes sure that `main()` can call the method `run()` on a reference of `General`. The instance on which `run()` is called on is instantiated in the function `bootstrap()` in *main.cpp*.

2.3 Class Commander

This class implements specific actions like selecting a value as order, sending order as a message and waiting for ACKs. The `run()` method basically calls `selectValue()` and then `send()`. After sending all messages to the lieutenants, `send()` waits for ACKs by calling `waitForAck()` and resends the messages if it does not receive an ACK from a lieutenant within the timeout period. After the round timeout period, it exits and declares its decision.

2.4 Class Lieutenant

This class implements specific actions like waiting for a message from the commander, forwarding order as a message, sending ACKs for messages that have been received, waiting for any kind of data (ACK or message) and taking appropriate action, verifying the signatures on a message received, and take a decision based on the final values in its set after $f + 1$ rounds. The `run()` method calls `receiveAndForward()` which loops till $f+1$ rounds are over. In the loop it calls `forwardMessage()` and `receiveMessage()` for forwarding and receiving messages. Whenever a message is received, the appropriate handler is called depending on whether it's an ACK or a message which takes necessary actions. Message handler sends an ACK and buffers the message to be forwarded in the next round. Whenever ACK timeout occurs and if there are pending ACKs to be received, messages are resent again. After the round timeout period, it proceeds to the next round.

2.5 Synchronous Behavior and ACK Timeouts

1. To achieve the property of synchronous communication, a round timeout of 500 milliseconds has been implemented after which every lieutenant goes to the next round.

2. To achieve reliable communication, an ACK mechanism has been implemented which has a timeout of 200 milliseconds after which a general resends the message. Resend is tried till the round lasts after which the next round starts and the lieutenants listen for new messages again.

3 Implementation Issues

The `recvfrom()` call for the lieutenants is a blocking one for the first round to account for any delay in commander coming up in the system. So, in the situation when Turret drops all packets from the commander (the commander is disloyal and stays silent), the lieutenant programs will all block and will then be killed by Turret. In this case the output file for lieutenants will not contain the decision since the lieutenant processes get killed before getting to decide.