Programming Assignment 1: Byzantine Agreement

DUE Saturday Feb. 18, 10 PM.

1 Overview

The Byzantine Generals' Problem [2] is one of classical problems in distributed systems. The goal of this problem is to defend against Byzantine failures of components of the system by guaranteeing that the correct components continue their services even if some components (less than majority) exhibit Byzantine failure. The objective of this lab is for students to get first hand experience in implementing a Byzantine agreement protocol.

2 Lab Tasks

In this lab, you need to implement the *Byzantine Agreement Algorithm* [2] to achieve consensus in the presence of Byzantine failures.

2.1 Implementation

2.1.1 Byzantine Agreement Algorithm

According to this algorithm, `Commander`[1] sends an order to all of his lieutenants in the first round. Each lieutenant forwards any message he received to all the other lieutenants (except the sender of the message) until every message has been received by $f+1$ generals at the presence of at most f Byzantine failures. If no more message to send in a round, each lieutenant decides using the `choice` method to have everyone agree on the same order.

Formally, we can restate the Byzantine Agreement algorithm with messages as follows. There are two possible orders that the `Commander` can send to a lieutenant either `attack` or `retreat`. Let V be the set of orders received by a lieutenant during the execution of this algorithm. At the end, every loyal lieutenant (i.e., process [2])uses a deterministic $choice(V)$ to decide on a value. $choice(V)$ is defined as follows:

$$choice(V) = \begin{cases} v & \text{if } V = \{v\} \\ RETREAT & \text{if } V = \varnothing \text{ or } \mid V \mid \geq 2 \end{cases}$$

[1] `Commander` commands the *lieutenant*s. We use *general*s to refer to both in general.

[2] We interchangeably use *process* and *lieutenant* in this document to refer to the program running on a host.

Algorithm: Byzantine agreement

Commander::

Selects a value v
Send v_0 to every lieutenant

Lieutenant P_i::

Initially $V_i \leftarrow \varnothing$
Upon receiving v_0 in round 0:
  if $V = \varnothing$ **then**
    $V \leftarrow \{v\}$
    Send v to all other processes except P_0 in the next round
  end if

Upon receiving v from P_{j_k} in round k:
  if $v \notin V$ **then**
    $V \leftarrow V \cup \{v\}$
    if k is less than f processes **then**
      Send v to processes other than P_0, $P_{j_1}, \ldots, P_{j_k}$ in the next round (where $P_{j_1}, \ldots, P_{j_k}$ are the senders of this message)
    end if
  end if

Upon finishing $(f+1)^{th}$ round: $\triangleright$ f is the total Byzantine processes in the system
  $decide = choice(V)$
  return $decide$

2.1.2 Task 1: Design and implementation of the algorithm

State diagrams [1] are used to present the high-level overview of the behavior of a system. So you need to draw state diagrams for `Commander` and lieutenants. **Make sure you include these state diagrams in your report**. You are advised to have the state diagrams ready before you start coding because these diagrams will come in handy during the implementation. You need to implement this algorithm in `C/C++` that allow the user to configure the execution of the process. Therefore your program (named as **general**) **must accept** the following command line arguments.

```
Usage: general -p port -h hostfile -f faulty -C commander_id [-o order]

  -p port
    The port identifies on which port the process
    will be listening on for incoming messages.
    It can take any integer from 1024 to 65535.

  -h hostfile
    The hostfile is the path to a file that contains
    the list of hostnames that the processes are
    running on. It assumes that each host is running
    only one instance of the process. It should be
    in the following format.
```

```
    xinu01.cs.purdue.edu
    xinu02.cs.purdue.edu
    ...

    All the processes will listen on the same port.
    The line number indicates the identifier of the process.

  -f faulty
    The "faulty" specifies the total number of Byzantine processes
    in the system. The value of faulty is non-negative. It
    also indicates after which round a process should terminate.
    Whenever a process finishes the (faulty + 1)th round or reaches
    a round greater than the (faulty + 1)th round, the process can
    safely decide and terminate. Note that the total number of
    processes must be no less than (faulty + 2).

  -C commander_id
    The identifier of the commander.

  -o order
    The order can be either "attack" or "retreat".
    If specified, the process will be the Commander and
    will send the specified order. Otherwise, the process will
    be a lieutenant.
    Please note that ONLY one process can have this option.
```

***Note*:** You **MUST** write `C/C++` code that compiles under the GCC (GNU Compiler Collection) environment. You have to make sure your code will **compile** and **run correclty** on the machines in WVH 102 (These machines run Linux).

Traitor mode of operation. A *traitor* general can perform sophisticated malicious act on every message. The traitors can perhaps make the loyal generals to take longer time to decide. To demonstrate the impact of a traitor or multiple traitors, your implementation will be tested with the traitor `Commander` and/or the traitor lieutenant.

A traitor `Commander` can send the valid order (*e.g.*, `attack`) to a random set of lieutenants and the invalid order (*e.g.*, `retreat`) to the remaining lieutenants. Even a traitor lieutenant can act similar to the traitor `Commander` except that the valid order means the order received in the message and the invalid order is created by flipping the received order. In addition, a traitor lieutenant can perform the following malicious behaviors while sending a message to another lieutenant.

- Remain silent.
- Delay the sending of a message.
- Send a message to a random set of lieutenants.
- Flip the order of a message.
- A random meaningful combination of the above malicious acts.

Synchronizing process. As you know that the aforementioned algorithm to solve the Byzantine Generals Problem implicitly assumes the processes to be synchronous at every round. You can utilize **timer** to achieve this synchrony among processes. For this lab, you can assume that

processes do not crash.

***Note*:** You must take care of some corner cases. For instance, all the processes may not start execution at the same time. If not properly handled, this situation may have processes agree on wrong values or may crash some of processes.

Communication protocol. The algorithm (see § 2.1.1) requires *reliable message delivery.* Therefore the generals must utilize a reliable communication protocol. For this lab, you **MUST** implement a simple reliable **UDP** as follows:

- After sending each packet, the sender starts a timer.
- The recipient sends an ACK for each received packet.
- Upon the reception of the ACK, the sender assumes successful delivery and resets the timer. But if the timer expires, the sender resends the packet and restarts the timer.

Tips: Instead of having an individual timer for each message in a round, a sender can keep one timer for the reliable UDP. The sender can send off the message of a round to all the receivers at once and then start the timer. Upon the reception of an ACK, the sender can remember the recipient. Later when the timer expires, the sender can decide whether it needs to resend the message, and if so, to which receiver(s).

Message format. Generals exchange messages to communicate with others. You **MUST adhere** to the following message format in your implementation even if you are using `C++`.

```
typedef struct {
    uint32_t type;  // Must be equal to 1
    uint32_t size; // size of message in bytes
    uint32_t round; // round number
    uint32_t order; // the order (retreat = 0 and attack = 1)
    uint32_t ids[]; // id's of the senders of this message
} ByzantineMessage;


typedef struct {
    uint32_t type;  // Must be equal to 2
    uint32_t size;  // size of message in bytes
    uint32_t round; // round number
} Ack;
```

***Note*:** You **MUST** follow the standard socket programming. For example, use *network byte order* for communication.

Output. Once consensus is reached each process (including the `Commander`) **must print to the standard output** the agreed value as follows and terminate. If the agreed value is `attack`, each loyal process and the loyal commander must print

```
ID of the process: Agreed on attack
```

and if the agreed value is `retreat`, each loyal process and the loyal commander must print

```
ID of the process: Agreed on retreat
```

3 Submission Instructions

A separate instruction with where to submit and how to submit will be posted on piazza. Your submission must include the following files:

1. Source and header files (no object files or binary)
2. Makefile to compile and to clean your project
3. A README file containing your name, instructions to run your code and anything you would like us to know about your program (like errors, special conditions, etc.)
4. A **REPORT** describing the system architectures, state diagrams, design decisions, and implementation issues

***Note*:** Please use the discussion group on **piazza** for general questions about the project (by selecting *project1* as the category when posting questions).

Late submission
The late submission policy in described in class syllabus. If you plan to use any of the late days for this project please send the instructor an email **before the project deadline** indicating how many late days you wish to use.

4 Additional resources

You may find the following resources helpful

- Socket programming: `http://beej.us/guide/bgnet/`
- Unix programming links: `http://www.cse.buffalo.edu/~milun/unix.programming.html`
- C/C++ programming link: `http://www.cplusplus.com/`

References

[1] State diagram. `http://en.wikipedia.org/wiki/State_diagram`.

[2] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.