Distributed Systems (Spring 2022)

Mini-project 2: The Byzantine General's problem with Consensus

Practical information:

Due date: Monday, May 9, midnight (FIRM DEADLINE) – Zero points after deadline

- This task can be performed individually or in a team of max 2 persons.
- Total amount of pts that that can be collected = 100 pts = 100 pts.
- Programming languages that can be used: **Java/Python**

Submit your solution to zhigang.yin@ut.ee, mohan.liyanage@ut.ee, (CC) huber.flores@ut.ee

Instructions: Please respond to the following questions. **Be clear and concise in your answers.** Provide enough explanation to support your arguments. Ambiguous answers to fill up space are considered wrong and no points are granted. For questions that involve implementation, be sure to include everything needed to execute your program **(Re-submissions are not allowed)**.

1. **Consensus in faulty systems**

Byzantine failures: In a distributed system, a node exhibits a byzantine failure when the output result produced by the node is arbitrary from time to time. A byzantine failure is difficult to detect as the node depicts normal behavior most of the time, but it is subject to arbitrary response fluctuations.

Consensus: It is a fundamental solution used in distributed systems, in consensus, one or more nodes propose a value, and then any one of the proposed values is agreed/analyzed as valid by the whole system. Moreover, in consensus, crashed nodes can be tolerated as long as a quorum is in place. Consensus can be used in a wide range of situations ranging from replication and consistency monitoring.

The generals' problem: (aka the two general's problem) is an essential coordination problem of distributed systems. In this problem, two generals are coordinating an attack to capture a city. The main caveat here is that the attack can be successful iff (if and only if), both generals attack at the same time, otherwise, the plan is doomed to fail. The two general's problem can model different failure situations in a distributed system, e.g., network communication. The two general's problem can also be extended to *N* generals to study byzantine failure detection in distributed nodes (aka the byzantine general's problem). Here, the main assumption is that among the generals that are coordinating the attack, there is a general that is a traitor (malicious) and wants to be hamper the attack plan, such that it is unsuccessful. These two problems have been discussed during the lecture in great detail ***(please refer to Lecture 3 – System models)***

Consensus for byzantine detection: Consensus can be used to detect abnormal behaviors in a distributed system. By making a collective decision based on the output of the nodes in the system, it is possible to take the right course of action despite the abnormal behaviors.

Task implementation: In this mini-project, the main idea is to implement byzantine general's problem using consensus to detect abnormal behaviors in generals. The program is required to provide a command line interface, and must implement the following functionality.

###Starting the program
$ **Generals_Byzantine_program.sh** N (75 pts)

- The program receives N as a parameter for starting its execution. Here, N is the number of processes (each process is a thread and depicts a general) that are created concurrently. N cannot be zero (N>0). All of these processes (generals) will forward a propose order (attack or retreat) when planning the attack.
- Each general has a unique identifier in the range of (1 to N)
- Each general has a state value S, which can be set either to F (Faulty) or NF (Non-faulty).
- A general is elected as primary of the system, while others are secondary nodes. Notice that if the primary is killed/crashed, it should be replaced by a secondary general (promoted to primary).

After the program has started, it has also to support the following commands via input terminal:

$ Actual-order: This command proposes to the primary an order (either “attack” or “retreat”). Here, the primary sends the order to the secondary nodes, and secondary nodes exchange messages to verify the proposed value. For each general, if the state is NF (Non-faulty), then generals work as expected just forwarding the proposed order between them. In contrast, if a general has a state F (Faulty), then each message that is exchanged with other generals is subject to (malicious) fluctuation between the proposed value and its counterpart. For instance, if the proposed value is “attack”, then when exchanging messages, the value oscillates between “attack” and “retreat” – (we can assume that the probability of selecting one choice over the other is equal 50/50)

After that, each general decides a final value based on the collected data, and a final quorum is performed to enforce the possible execution of the order. For instance, (assuming 4 generals)

##

$ **Generals_Byzantine_program.sh** 4

##
$ actual-order attack
G1, primary, majority=attack, state=NF
G2, secondary, majority=attack, state=NF
G3, secondary, majority=attack, state=NF
G4, secondary, majority=attack, state=NF
Execute order: attack! Non-faulty nodes in the system – 3 out of 4 quorum suggest attack

$ actual-order retreat

G1, primary, majority=retreat, state=NF
G2, secondary, majority=retreat, state=NF
G3, secondary, majority=retreat, state=NF
G4, secondary, majority=retreat, state=NF
Execute order: retreat! Non-faulty nodes in the system – 3 out of 4 quorum suggest retreat

##
$ g-state ID “State”, where ID is the general ID and state is either “Faulty” or “Non-faulty”. For instance,

$ g-state 3 faulty

G1, state=NF
G2, state=NF
G3, state=F
G4, state=NF

If no parameter is passed to g-state, then it is just show the list of generals and along with their respective state and role

$ g-state

G1, primary, state=NF
G2, secondary, state=NF
G3, secondary, state=F
G4, secondary, state=NF

##
$ g-kill ID, this commands remove a general based on its ID. For instance

$ g-kill 1

G2, state=NF
G3, state=F
G4, state=NF

(in case primary is deleted) New primary selected automatically

$ g-state

G2, primary
G3, secondary
G4, secondary

Notice also that a minimal number of nodes need to be in place to apply Consensus (see appendix). If the minimum amount is not in place, no decision can be reached. For instance,

$ actual-order attack

G2, majority=attack, state=NF
G3, majority=undefined, state=F
G4, majority=undefined, state=NF
Execute order: cannot be determined – not enough generals in the system! 1 faulty node in the system - 2 out of 3 quorum not consistent

##
$ g-add K, where K is the number of new generals. By default, generals have a non-faulty state once created

$ g-add 2

G2, primary
G3, secondary
G4, secondary
G5, secondary
G6, secondary

$ g-state

G2, primary, state=NF
G3, secondary, state=F
G4, secondary, state=NF
G5, secondary, state=NF
G6, secondary, state=NF

$ actual-order attack

G2, primary, majority=attack, state=NF
G3, secondary, majority=attack, state=F
G4, secondary, majority=attack, state=NF
G5, secondary, majority=attack, state=NF
G6, secondary, majority=attack, state=NF
Execute order: attack! 1 faulty node in the system – 3 out of 5 quorum suggest attack

Constraints: The algorithm should be implemented using explicit communication primitives (RPC).

Syntax: By default, all the commands are expected to be written solely using a lower-case format.

Deliverables for your Generals_Byzantine implementation:

1- A short 1 min video showing your program in execution (showing all the commands)
2- Source code and everything needed for executing your program (binaries and dependencies (if any)) – in GitHub. Please also commit regularly, such that it is possible to track the contributions of each team member (if more than one).
3- We will evaluate your work with multiple configurations (different N parameters). Thus, do not hard code your solution just to work with the described example.

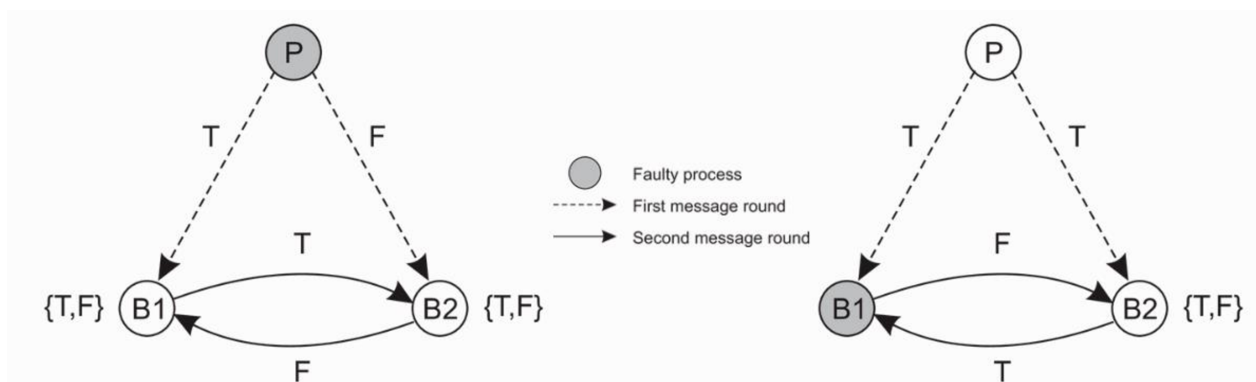
Appendix

Requirements for Byzantine Agreement.

- Reaching consensus in a fault-tolerant process group in which k members can fail assuming arbitrary failures.
- We need at least 3k + 1 members to reach consensus under these failure assumptions.

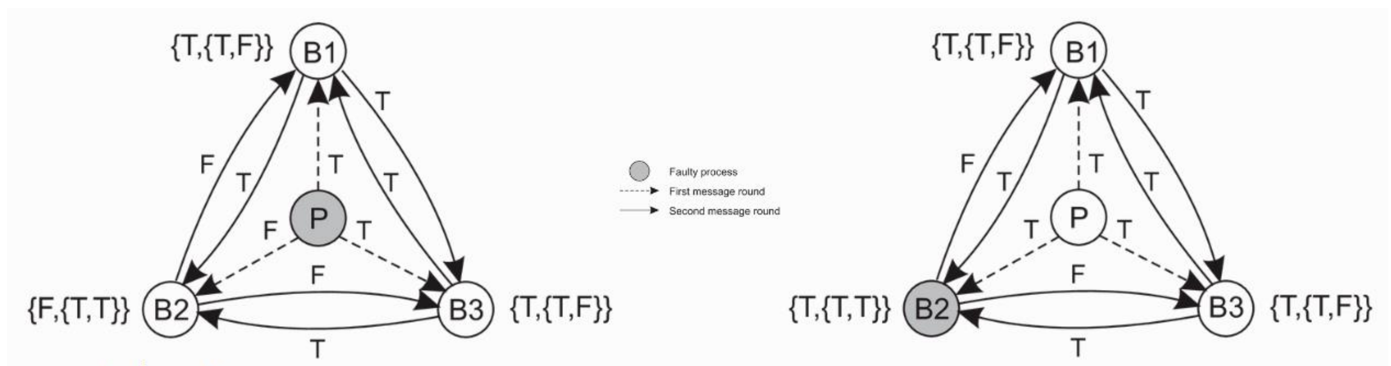
Definitions: Consider a process group consisting of **n** members, of which one has been designated to be the **primary, P**, and the remaining **n - 1** to be the **backups B1, . . . ,Bn -1**.

- **Assumptions:**
  - A client sends a value v ϵ {T, F} to the primary, where v stands for either **True** or **False**.
  - Messages may be lost, but this can be detected.
  - Messages cannot be corrupted without that being detected (and thus subsequently ignored).
  - A receiver of a message can reliably detect its sender.
- **Requirements for Byzantine Agreement:**
  - **BA1:** Every nonfaulty backup process stores the same value.
  - **BA2:** If the primary is non faulty then every nonfaulty backup process stores exactly what the primary had sent.
  - **Note:** If the primary is faulty, BA1 tells us that the backups may store the same, but different (and thus wrong) value than the one initially sent by the client. Furthermore, it should be clear that if the primary is not faulty, satisfying BA2 implies that BA1 is also satisfied.
- **Why having 3k processes is not enough:** Consider the following example where **k=1**:



In above Figure:

- In first figure, we see that the **faulty primary P** is sending two different values to the **backups B1 and B2**, respectively. In order to reach consensus, both backup processes forward the received value to the other, leading to a second round of message exchanges. At that point, B1 and B2 each have received the set of values {T, F}, from which it is impossible to draw a conclusion.
- Likewise, we cannot reach consensus when wrong values are forwarded. In second Figure, the **primary P** and **backup B2** operate correctly, but B1 is not. Instead of forwarding the value T to process B2, it sends the incorrect value F. The result is that B2 will now have seen the set of values {T, F} from which it cannot draw any conclusions. In other words, P and B2 cannot reach consensus. More specifically, B2 is not able to decide what to store, so that we cannot satisfy requirement BA2.
- **Why having 3k + 1 processes is enough:** Consider the figure where **k=1**:



In above Figure:

- In first Figure, the **primary P** is faulty and is providing inconsistent information to its backups:
    - ***Solution:*** The processes will forward what they receive to the others. During the first round, P sends T to B1, F to B2, and T to B3, respectively. Each of the backups then sends what they have to the others. With only the primary failing, this means that after two rounds, each of the backups will have received the set of values {T,T, F}, meaning that they can reach consensus on the value T.
- In Second part of the figure, we consider the case that one of the backups fails.
    - ***Solution:*** Assume that the (nonfaulty) primary sends T to all the backups, yet B2 is faulty. Where B1 and B3 will send out T to the other backups in a second round, the worst that B2 may do is send out F, as shown in the figure. Despite this failure, B1 and B3 will come to the same conclusion, namely that P had sent out T, thereby meeting our requirement BA2 as stated before.