|  |  |  |
| --- | --- | --- |
|  | **Distributed Algorithms - Exercise**  **Winter term 2017/18** | **Danh Le Phuoc**  **Qian Liu**  **ODS** |
| *Exercise sheet 4* | | |

**Exercise 4.1: Distributed Consensus**

**Questions**

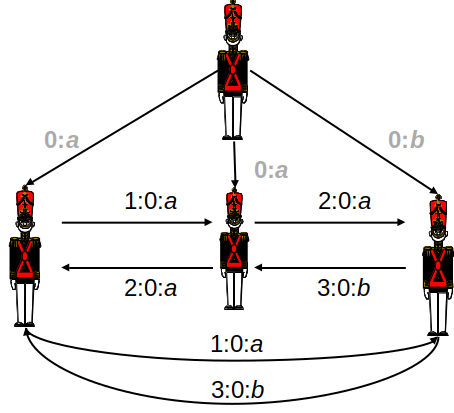
**Describe the agreement of the byzantine generals and explain the recursive algorithm for oral messages.**

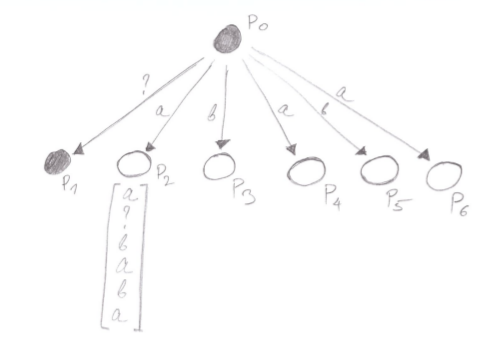
Byzantine generals problem is a common problem in achieving consensus among processes, who may be faulty. One distinguishable process proposes a single value, and all other processes must agree on that value. Problem is that some of the processes are faulty. Even the proposing process can be faulty, so byzantine problem can be defined as proposing a value by a single process and:

* All correct processes should agree on the single value
* If a proposing process is not faulty, the value, agreed upon, should correspond to proposed value

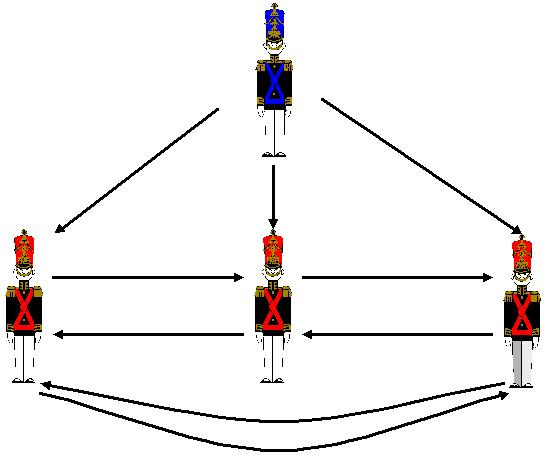
let n be the number of generals and m the number of faulty generals. To reach an agreement, the following must stand : or

When a general receives the command from the commander, it is not sure if the commander is faulty. Therefore, each general sends its message to other generals. Each general decides on the most likely value, shown on Figure 1. In some scenarios, there can be draws (Figure 2). To handle these situations, every general starts an instance of an algorithm with n-1 processes with the value which it received from its commander.

  
Figure 1: Decision is made by each general regardless of the fact that commander is faulty. Each general adopts most likely value

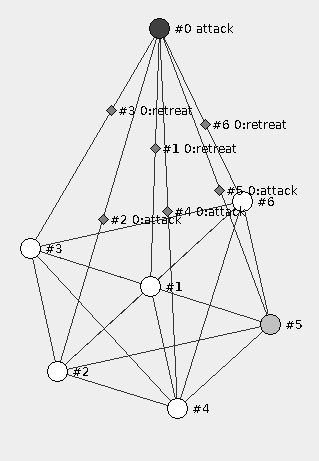
  
Figure 2: Ilustration in which a faulty process can introduce a draw problem among other processes. In order to prevent this scenario, additional recursions are required

**Implement the agreement of the byzantine generals for m faulty generals using pseudo-codes. Do your implementation in a way that it is possible to distinguish between faulty and non-faulty generals as well as between the commanding general and the lieutenant generals. Explain your implementations with graphs and texts, Messages shall visualize their message path and the actual value, as well as integrity (message has been corrupted). Demonstrate your implementation given the following scenarios:**



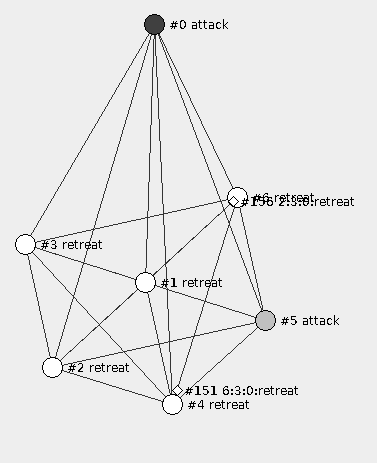
* 1. **n = 4 (4 generals), m = 1 (one randomly chosen general is faulty).**
  2. **n = 7 (7 generals), m = 2 (two randomly chosen generals are faulty).**
  3. **n = 10 (10 generals), m = 3 (three randomly chosen generals are faulty).**

See the attached implementation in java. It is possible to configure the number of generals (n) in the myconfig.txt file, the number of traitors (m) in ByzantineGenerals.java file as well as the initial command, either ‘attack’ or ‘retreat’.



In this figure, the node with id 0 is the commander, all other nodes are lieutenants. White depicts honest generals and messages. Gray depicts faulty generals and messages. So in the figure, the commander with id 0 and the lieutenant with id 5 are traitors. Also the general’s messages are faulty. It sends both, commands to attack and commands to retreat in the initialization phase.

Afterwards the remaining lieutenants interchange the commands. Node 5, the traitor, changes commands, which are ‘attack’ to ‘retreat’ commands and the other way around in order to confuse the other lieutenants.



The next figure shows an example for a message. The message consists of the command and the receiver chain. The receiver chain is a string recording the previous receivers. For example message 151 originated from the commander 0, then lieutenant 3 forwarded the command, then lieutenant 6 and finally lieutenant 4 is about to receive the message and save it in the message tree.

#4

+0:attack

|+2:0:attack

||+3:2:0:attack

||+5:2:0:retreat

||+1:2:0:attack

||+6:2:0:attack

|+6:0:retreat

||+2:6:0:retreat

||+5:6:0:attack

||+3:6:0:retreat

||+1:6:0:retreat

|+1:0:retreat

||+5:1:0:attack

||+3:1:0:retreat

||+6:1:0:retreat

||+2:1:0:retreat

|+5:0:retreat

||+2:5:0:retreat

||+1:5:0:retreat

||+6:5:0:retreat

||+3:5:0:retreat

|+3:0:retreat

||+5:3:0:attack

||+1:3:0:retreat

||+2:3:0:retreat

||+6:3:0:retreat

The above shows the message tree of node 4. To calculate node 4’s conclusion to either attack or retreat, it utilizes its message tree. At first, the algorithm calculates the command majority for each of the subtrees. The resulting tree is as follows:

#4

+0:attack

|+attack

|+retreat

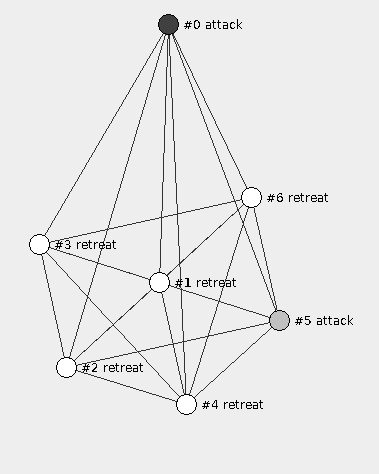
|+retreat

|+retreat

|+retreat

Then, the algorithm calculates the majority for the remaining tree that results in retreat. MessageTree#result() implements the calculation recursively and is generic to all m’s and n’s, which fulfill the preconsumption of the byzantine generals problem.

The last figure depicts the result of the execution.

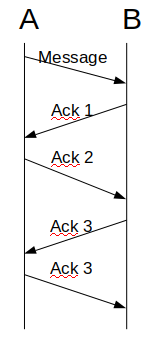


The honest lieutenant conclude to follow the command ‘retreat’.

**Two loyal generals are planning to coordinate their actions for conquering a strategic town. To conquer the town, they need to attack at the same time; otherwise, if only one of them attacks and the other does not attack at the same time, then the generals are likely to be defeated. To plan the attack, they send messages back and forth via trusted messengers. The communication is asynchronous. However, the messengers can be killed or captured – so the communication is unreliable.**

**Argue why it is impossible for the two generals to coordinate their actions of attacking at the same time.**

**(Hint: Unlike the byzantine generals problem, here all generals are loyal, but the communication is unreliable. If general A sends a message attack at 2 a.m. to general B, then he will want an acknowledgment from B; otherwise, he won’t attack in the fear of moving alone. But B also will ask for an acknowledgment of the acknowledgment. Now you see the rest of the story.)**

  
Figure 3: caption here \*\*\*\*\*

As can be seen from Figure 5, general A sends a proposal to general B. To be certain in the reception of the message, he awaits mack 1. When B responds, he remains uncertain if the acknowledgement was delivered. Upon receiving an acknowledgement, process A acknowledges(Ack 2) that he received an acknowledgement(Ack 1). At that point, he is uncertain that B received Ack 2, and waits for further acknowledgements.

This is repetitive and at each iteration, one of the processes is uncertain in the acknowledgement reception on the other side, and will therefore remain inactive.

**Exercise 4.2: Distributed Transactions**

**Questions**

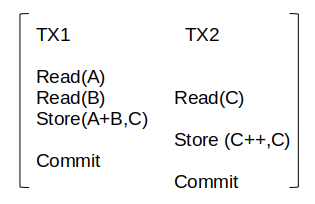
**Two-phase locking works for distributed transactions. As an alternative, consider this: There are n (n > 1) servers; each server manages m objects (m > 1). Each server will allow transaction sequential access to the locks on the objects managed by it. Is this sufficient for serializability? Is this sufficient to avoid deadlocks? Explain.**

Here, we can distinguish two scenarios:

* Sequential access refers to transactions and each transaction has standalone access to m>1 objects.
* Sequential access refers to the objects, where, one transaction can at each moment hold one object, while, multiple transactions can hold multiple different object simultaneously.

First case will result in serializable transactions with no deadlocks. This is a direct result of transaction independence, where, at most one transaction has access to m>1 objects, and with no other transactions to interfere, deadlocks are prevented.

Second scenario will prevent deadlocks because each object is locked by only one transaction for certain time interval, after which the lock is free, and another transaction takes the lock.

Serializability is limited by data consistency. Consider the following scenario:

Here, two transactions require objects A,B,C. At each moment one transaction holds one object, but the resulting data is inconsistent for TX1 because it commits on incorrect value stored in C.

**Specify two execution histories H1 and H2 over a set of transactions, so that (a) H1 is permissible under 2PL, but not under basic time stamp ordering, and (b) H2 is permissible under basic time stamp ordering, but not under 2PL.**

Time stamp ordering corresponds to appending each transaction with a time stamp. Scheduler that rearranges the execution of possible transactions based on the Lamport time stamp. A drawback of this approach can be loop within time stamp labels. Let H be a set of transactions:

* T1(a)
* T2(b)
* T3(c)

Where a,b,c are the Lamport clock stamps. The problem arises if a,b and c are temporally dependent as a loop. Lamport clocks satisfy the following : if a < b and b < c then a < c. In distributed computing, As a result of delays in channels and channel overtaking, there can be a situation in which c is performed before a if scheduler is unaware of T1(a).

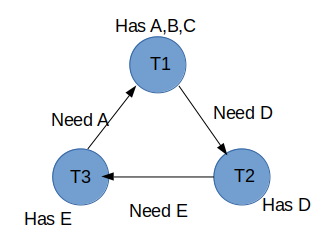
Another problem for time stamp ordering is having multiple transactions with same time stamps.

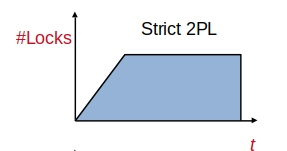
For the set of transactions to be non-permissible under 2PL, simple deadlock is needed, assuming that strict 2PL is used (Figure 6).

Let history H be a set of following transactions:

* T1(A,B,C,D)[t1]
* T2(D,E)[t2]
* T3(A,E)[t3]

Assume T1 starts first, and locks A,B,C. Afterwards, T2 locks D and T3 locks E. Now, T1 is waiting for T2 to free D, T2 is waiting for T3 to free E and T1 is waiting for T1 to free A. Situation, shown on Figure 7, can be solved by assigning time stamps and gaining access with respect to the lower value of Lamport time stamps.

  
Figure 4: Deadlock, as a result of strict 2PL

  
Figure 5: Strict 2 Phase Locking