1. Introduction:

1.1 Byzantine Fault

The Byzantine General's problem is one of many in the field of agreement protocols. In 1982, Leslie Lamport described this problem in a paper written with Marshall Pease and Robert Shostak. Lamport framed the paper around a story problem after observing what he felt was an inordinate amount of attention received by Dijkstra's Dining Philosopher problem. A Byzantine Fault is an incorrect operation (algorithm) that occurs in a distributed system that can be classified as:

- Omission Failure

  This a failure of not being present such as failing to respond to a request or not receiving a request.

- Execution Failure or Lying

  This is the failure due to sending incorrect or inconsistent data, corrupting the local state or responding to a request incorrectly.

The examples of the Byzantine Fault are given below:

a. Round off errors passed from one function to another and then another, etc.

b. Corrupted system databases where the error is not detected

c. Compiler errors

d. An undetected bit flip producing a bad message

The Byzantine General’s Problem is an illustrative example of Byzantine Faults. A number of different armies want to besiege an enemy city. Success requires agreement on a common plan amongst the disbursed army; communication is only by messengers. This is a worse case model since the Byzantine Fault can generate misleading information causing a maximum of confusion.

- Complication

  There may be traitors who send out conflicting messages to sow confusion.

- Challenge

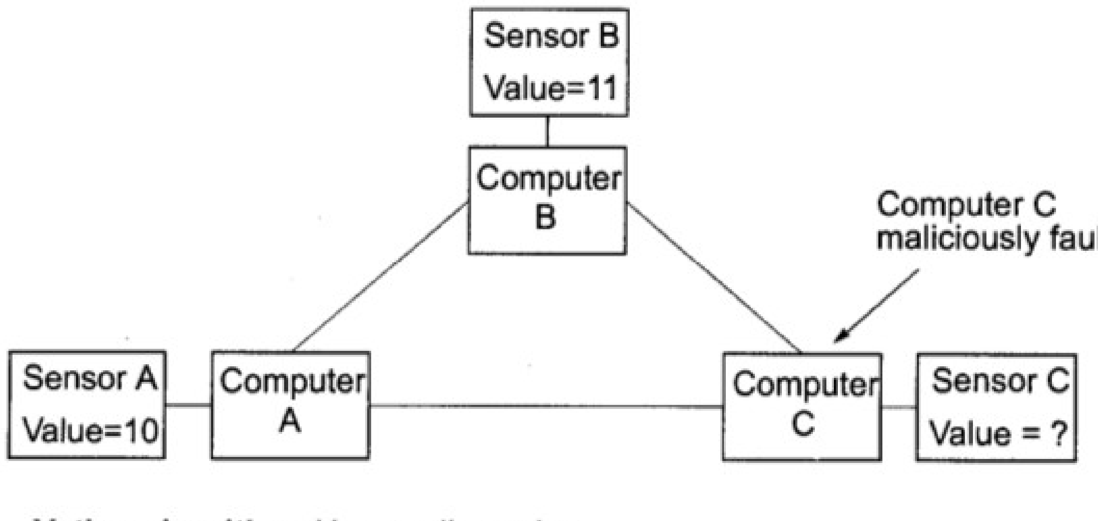
  Find an algorithm that ensures the armies come to an agreement
  and attack at the same time.

1.2 The Real World Relationships:

This problem is built around an imaginary General who makes a decision to attack or retreat, and must communicate the decision to his lieutenants. A given number of these actors are traitors (possibly including the General). Traitors cannot be relied upon to properly communicate orders; worse yet, they may actively alter messages in an attempt to subvert the process. Several armies, each under the command of a general, are camped outside a city which they plan to attack. One of the generals will issue the order attack or retreat. One or several of the generals may be traitors. To win the battle all loyal generals must attack at the same time. The traitors will attempt to fool the loyal generals so that not all of them attack at the same time. To stop the traitor's malicious plan, all generals exchange messages directly with each other. The generals are collectively known as processes. The general who initiates the order is the source process, and the orders sent to the other processes are messages. Traitorous generals and lieutenants are faulty processes, and loyal generals and lieutenants are correct processes. The order to retreat or attack is a message with a 1 or 0.

- Generals are the processors.
- Traitors are faulty processors or faulty system components. It may include software.
- Messengers are processor communications/system data bus.

Impossibility to reach agreement on sensor values



Voting algorithm: Use median value.
Computer *C* sends value = 9 to Computer *A* and *value = 12* to Computer *B*
A uses *median*{9,10,11} = 10, *B* uses *median*{10, 11, 12} = 11
C's malicious behaviour causes A and B to work with inconsistent values.

Another example of the Byzantine Fault is illustrated by the image below:

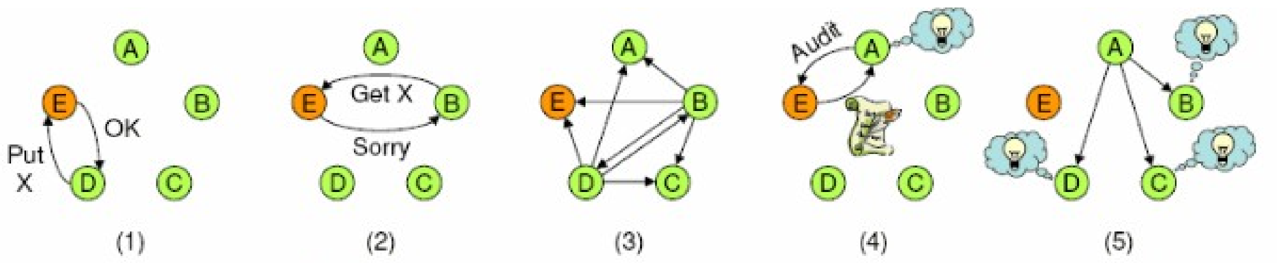


Figure 3: *Node E stores an object for client D (1) and then tries to hide it from client B (2). The two clients broadcast the authenticators they have obtained from E (3). Later, A audits E, discovers the inconsistency, and exposes E (4). Finally, node A broadcasts its evidence against E, so the other nodes can expose E as well (5).*

The generals are collectively known as *processes*. The general who initiates the order is the *source process*, and the orders sent to the other processes are *messages*. Traitorous generals and lieutenants are *faulty processes*, and loyal generals and lieutenants are *correct processes*. The order to retreat or attack is a message with a 1 or 0.

In general, a solution to *agreement problems* must pass three tests: termination, agreement, and validity. As applied to the Byzantine General's problem, these three tests are:

- A solution has to guarantee that all correct processes eventually reach a decision regarding the value of the order they have been given.
- All correct processes have to decide on the same value of the order they have been given.
- If the source process is a correct process, all processes have to decide on the value that was originally given by the source process.

One side effect of this is that if the source process is faulty, all other processes still have to agree on the same value. It doesn't matter what value they agree on, they simply all have to agree. So if the General is subversive, all lieutenants still have to come to a common, unanimous decision.

2. Difficulties:

This agreement problem doesn't lend itself to an easy solution. Imagine, for example, that the source process is the only faulty process. It tells half the processes that the value of their order is 0, and the other half that their value is 1.

After receiving the order from the source process, the remaining processes have to agree on a value that they will all decide on. The processes could quickly poll one another to see what value they received from the source process.

In this scenario, imagine the decision algorithm of a process that receives an initial message of 0 from the source process, but sees that one of the other processes says that the correct value is 1. Given the conflict, the process knows that either the source process is faulty, having given different values to two different peers, or the peer is faulty, and is lying about the value it received from the source process.

It's fine to reach the conclusion that someone is lying, but making a final decision on who is the traitor seems to be an insurmountable problem. And in fact, it can be proven that it is impossible to decide in some cases. The classic example used to show this is when there are only three processes: One source process and two peer processes.

In the configurations in Figures 1 and 2, the peer processes attempt to reach consensus by sending each other their proposed value after receiving it from the source process. In Figure 1, the source process (P_1) is faulty, sending two different values to the peers. In Figure 2, P_3 is faulty, sending an incorrect value to the peer.

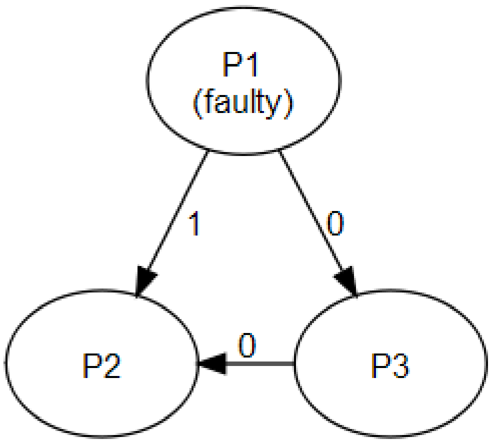


Figure 1: The case in which the source process is faulty.

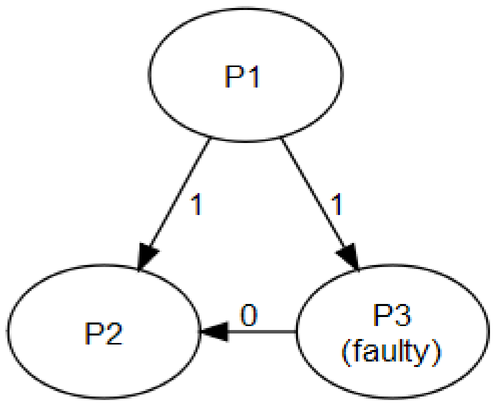


Figure 2: The case in which P_3 is faulty.

You can see the difficulty P_2 faces in this situation. Regardless of which configuration it is in, the incoming data is the same. P_2 has no way to distinguish between the two configurations, and no way to know which of the two other processes to trust.

This situation doesn't necessarily get better just by throwing more nonfaulty processes at the problem. A naïve algorithm (as in Figures 1 and 2) would have each process tell every other process what it received from P_1. A process would decide the correct decision by simple majority.

It's relatively easy to show that, regardless of how many processes are in the system. A subversive source process with one collaborator can cause half the processes to choose to attack, and half the processes to retreat, leading to maximum confusion.

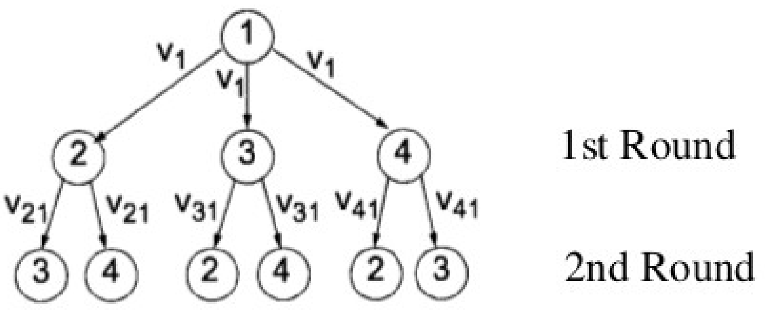
3. Byzantine Fault Tolerance

To reach agreement between the loyal generals in the presence of m faults.

- There must be at least 3m + 1 processors to deal with m faults (traitors)
- Each processor must be connected to each other through at least 2m + 1 communication paths
- m + 1 rounds of messages must be exchanged
- The processors must be synchronized within a known skew of each other

Such a system is called Byzantine Resilient which is a fully fault tolerant system.

Tolerating One Byzantine Fault

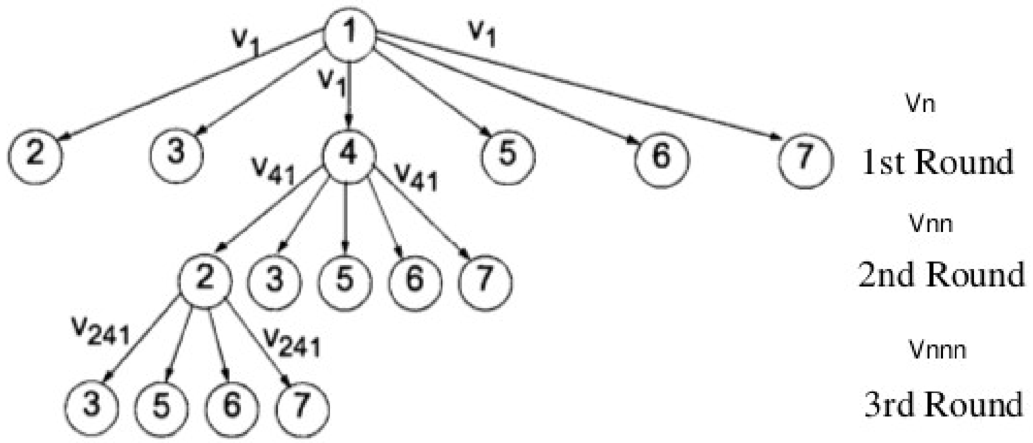


Unit 2 uses $v'_1 = majority\{v_1, v_{31}, v_{41}\}$

Number of messages for agreement on one value is: $3 + 2 \cdot 3 = 9$ messages

A majority decision involving 4 units (e.g. 4 sensor values) requires: $9 \cdot 4 = 36$ messages

Tolerating Two Byzantine Faults



Unit 2 usesv'_1 = *majority*$\{v_1, v'_{31}, v'_{41}, v'_{51}, v'_{61}, v'_{71}\}$

v'_{41} = *majority*$\{v_{41}, v_{341}, v_{541}, v_{641}, v_{741}\}$

Number of messages for agreement on one value is $6 + 6(5 + 5\cdot4) = 156$ messages

4.Algorithm:

Lamport's algorithm is a recursive definition, with a base case for *m=0*, and a recursive step for *m>0*:

- r Algorithm OM(0)
- The general sends his value to every lieutenant.
- Each lieutenant uses the value he receives from the general.
- r Algorithm OM(m), m>0
- The general sends his value to each lieutenant.
- For each *i*, let v_i be the value lieutenant *i* receives from the general. Lieutenant *i* acts as the general in Algorithm OM(m-1) to send the value *vi* to each of the *n-2* other lieutenants.
- For each *i*, and each $j \neq i$, let v_i be the value lieutenant *i* received from lieutenant *j* in step 2 (using Algorithm (m-1)). Lieutenant *i* uses the value *majority*$(v_1, v_2,...v_n)$.

Lamport's algorithm actually works in two stages. In the first step, the processes iterate through *m+1* rounds of messages. In the second stage of the algorithm, each process takes all the information it has been given and uses it to come up with its decision.

4.1 The First Stage:

The first stage is simply one of data gathering. The algorithm defines *m+1* rounds of messaging between all the processes.

In Round 0, the General sends the order to all of its lieutenants. Having completed his work, the General now retires and stands by waiting for the remaining work to complete. Nobody sends any additional messages to the General, and the General won't send any more messages.

In each of the remaining rounds, each lieutenant composes a batch of messages, each of which is a tuple containing a value and a path. The value is simply a 1 or a 0. The path is a string of process IDs, $<ID_1, ID_2,...,ID_n>$. What the path means in this context is that in Round *N*, P_{ID1} is saying that it was told in Round *N-1* that P_{IDN-1} was told by *P* that the command value was *v*. (This is much like the classic party game in which a message is whispered from ear to ear through a chain of players, becoming slightly mangled along the way.) No path can contain a cycle. In other words, if ID_1 is 1, no other ID in the string of process IDs will be a 1.

The message definition is easy in Round 1. Each process broadcasts a message to all the other processes, including itself, but excluding the General, with the value it received from the General and its own process ID.

For example, suppose that in Round 0 that P_1, a faulty general, told P_2, P_3, and P_4 that the command value was 0, and told P_5, P_6, and P_7 that the command value was 1. In Round 1, the messages in Table 1 would be sent.

Sender = P_2		**Sender = P_3**		**Sender = P_4**		**Sender = P_5**		**Sender = P_6**		**Sender = P_7**	
Dest	**Msg**	**Dest**	**Msg**	**Dest**	**Msg**	**Dest**	**Msg**	**Dest**	**Msg**	**Dest**	**Msg**
P_2	{0,12}	P_2	{0,13}	P_2	{0,14}	P_2	{1,15}	P_2	{1,16}	P_2	{1,17}
P_3	{0,12}	P_3	{0,13}	P_3	{0,14}	P_3	{1,15}	P_3	{1,16}	P_3	{1,17}
P_4	{0,12}	P_4	{0,13}	P_4	{0,14}	P_4	{1,15}	P_4	{1,16}	P_4	{1,17}
P_5	{0,12}	P_5	{0,13}	P_5	{0,14}	P_5	{1,15}	P_5	{1,16}	P_5	{1,17}
P_6	{0,12}	P_6	{0,13}	P_6	{0,14}	P_6	{1,15}	P_6	{1,16}	P_6	{1,17}
P_7	{0,12}	P_7	{0,13}	P_7	{0,14}	P_7	{1,15}	P_7	{1,16}	P_7	{1,17}

Table 1: Messages sent by all six lieutenant processes in Round 1.

Things get more complicated in the second round. From the previous rule, we know that each process now has six values that it received in the previous round—one message from each of the three processes—and it needs to send each of those messages to all of the other processes, which might mean each process would send 36 messages out.

In Table 1, I showed the messages being sent to all six processes, which is redundant because the same messages are broadcast to all processes. For Round 2, I just show the set of messages that each process sends to all of its neighbors.

The six messages that P_2 received in Round 1 were {0,12}, {0,13}, {0,14}, {1,15}, {1,16}, and {1,17}. According to the earlier definition, P_2 will append its process ID to the path and forward each resulting message to all other processes. The possible messages it could broadcast in Round 2 are {0,122}, {0,132}, {0,142}, {1,152}, {1,162}, and {1,172}. The first message, {1,122}, contains a cycle in the path value of the tuple, so it is tossed out, leaving five messages to be sent to all processes.

The first message that P_2 is sending in Round 2, {0,132}, is equivalent to saying, "P_2 is telling you that in round 1 P_3 told it that in round 0 that P_1 (the General) told it that the value was 0." The five messages shown in P_2's column in Table 2 are sent to all six lieutenant processes, including itself.

Sender = P_2	**Sender = P_3**	**Sender = P_4**	**Sender = P_5**	**Sender = P_6**	**Sender = P_7**
{0,132}	{0,123}	{0,124}	{0,125}	{0,126}	{0,127}
{0,142}	{0,143}	{0,134}	{0,135}	{0,136}	{0,137}
{1,152}	{1,153}	{1,154}	{0,145}	{0,146}	{0,147}
{1,162}	{1,163}	{1,164}	{1,165}	{1,156}	{1,157}
{1,172}	{1,173}	{1,174}	{1,175}	{1,176}	{1,167}

Table 2: Messages sent by all six processes in Round 2.

It's easy to see that as the number of processes increases, the number of messages being exchanged starts to go up rapidly. If there are *N* processes, each process sends *N-1* messages in Round 1, then *(N-1)*(N-2)* in Round 2, and *(N-1)*(N-2)*(N-3)* in Round 3. That can add up to a lot of messages in a big system.

4.2 The Second Stage:

While sending messages in each round, processes are also accumulating incoming messages. The messages are stored in a tree format, with each round of messages occupying one rank of the tree. Figure 3 shows the layout of the tree for a simple configuration with six processes, one of which can be faulty. Because *m=1*, there are just two rounds of messaging: The first, in which the general sends a value to each lieutenant process, and a second, in which each process broadcasts its value to all the other processes. Two rounds of messaging are equivalent to two ranks in the tree.

In Figure 3, there are six processes, and the General (P_1) is faulty—sending a 1 to the first three lieutenants and 0 to the last two. The subsequent round of messaging results in P_2 having an information tree that looks just like that in Figure 3.

Once a process has completed building its tree, it is ready to decide on a value. It does this by working its way up from the leaves of the tree, calculating the majority value at each rank, and assigning it to the rank above it. The output value at each level is the third item in the data structure attached to each node, and those values are all undefined during the information gathering stage.

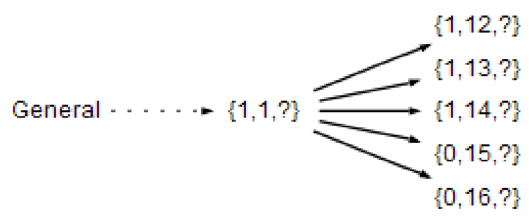


Figure 3: The Tree Layout for 5 processes with 1 faulty process.

Calculating the output values is a three-step process:

- Each leaf node in the tree (all values at rank *m*) copies its input value to the output value.
- Starting at rank *m-1* and working down to 0, the output value of each internal node is set to be the majority of the output values of all its children. In the event of a tie, an arbitrary tie-breaker is used to assign a default value. The same default value must be used by all processes.
- When complete, the process has a decision value in the output of the sole node at rank 0.

In Figure 3, step 1 of the process assigns the initial values to the leaf nodes. In the next step, the majority value of {1,1,1,0,0} is evaluated and returns a value of 1, which is assigned to the output value in rank 0. Because that is the top rank, the process is done, and P_1 decides on a value of 1.

Every lieutenant value in a given exercise will have the same paths for all its nodes, and in this case, because only the General is faulty, we know that all lieutenants will have the same input values on all its leaves. As a result, all processes will agree on the same value, 1, which fulfills the agreement property.