Distributed Byzantine Generals – A Fault-Tolerant Solution

1. Introduction

The Byzantine Generals Problem illustrates the challenges of achieving consensus in a distributed system in the presence of faulty (or malicious) nodes. As originally introduced by Lamport, Shostak, and Pease , the problem models a scenario in which a commanding general and several lieutenants must agree on a common plan—either to attack or retreat—despite the possibility that some nodes (generals) might be traitors sending conflicting information.

Our solution implements a distributed version of Lamport's OM algorithm using gRPC in Python. Unlike a centralized approach, our design deploys independent server processes (one per general) to simulate a real-world distributed system. Each server runs on its own port, logs intermediate steps to a dedicated log file, and communicates with its peers to propagate orders. A separate scaling script, `scale_server.py`, automates the process of spawning these independent servers.

2. Byzantine Fault Tolerance & Difficulties

2.1 Byzantine Faults

A Byzantine Fault is any fault presenting arbitrary (possibly malicious) behavior, including:

- **Omission Failures:** Missing messages or no response.
- **Execution Failures (Lying):** Sending incorrect or inconsistent data.

In our scenario, traitors (faulty generals) may deliberately invert orders (e.g., converting "ATTACK" to "RETREAT" for certain recipients) to subvert consensus.

2.2 Difficulties in Consensus

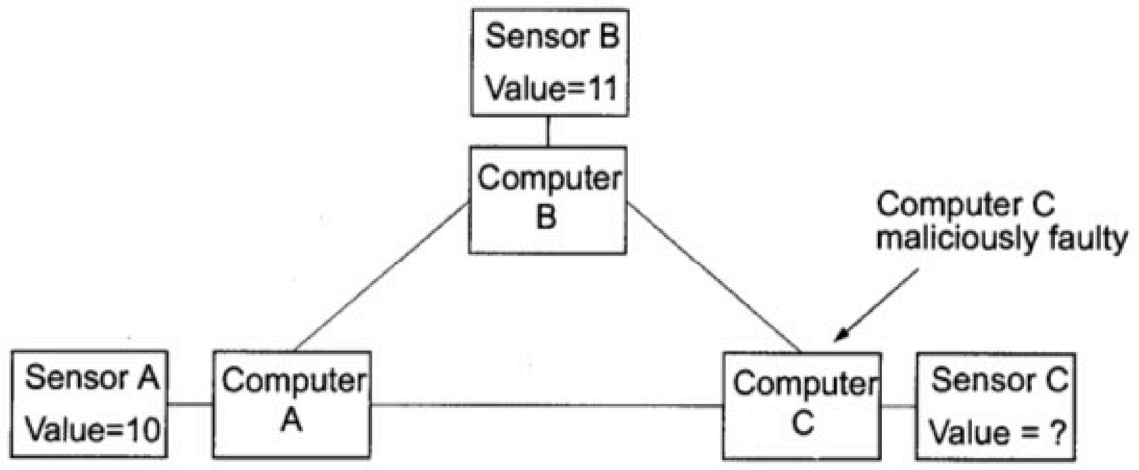
The inherent challenge in the Byzantine Generals Problem is that even a single traitor can inject ambiguity into the system. A loyal process must be able to decide correctly even if it receives conflicting orders. Thus, the algorithm requires at least $3t + 1$ generals to tolerate t faults/traitors and $t+1$ rounds of messages to be exchanged. Each processor must be connected to each other through at least $2t + 1$ communication paths.

3. 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:

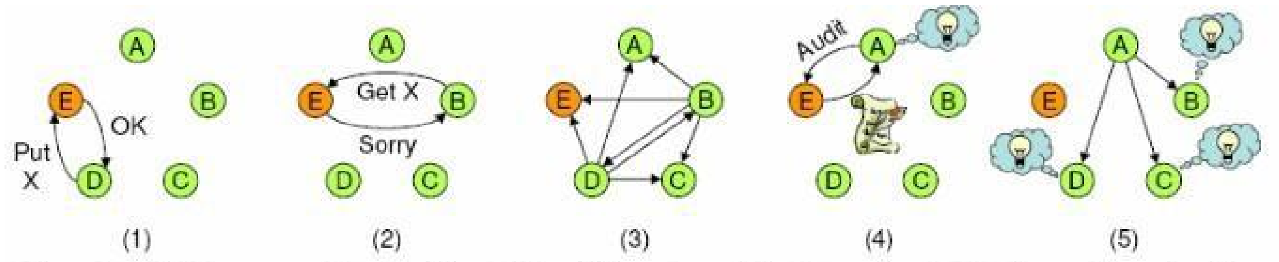


Figure 3: *Node E stores an object for client D (1) and then tries to hide it from client B (2). The two clients broadcast the authenticators they have obtained from E (3). Later, A audits E, discovers the inconsistency, and exposes E (4). Finally, node A broadcasts its evidence against E, so the other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

4. Lamport’s Byzantine Generals Algorithm – In Depth

4.1 Background and Problem Statement

In a distributed system, achieving agreement is challenging when some nodes (generals) may behave arbitrarily (i.e., be faulty or malicious). The Byzantine Generals Problem abstracts this situation into a scenario where:

- A **source process** (the commanding general) issues an order (ATTACK or RETREAT).
- **Lieutenant processes** must agree on the order.
- Up to *m* faulty nodes (traitors) may exist, potentially sending conflicting or incorrect messages.

For the system to achieve consensus despite these faults, the algorithm must satisfy three properties:

- **Termination:** All loyal (non-faulty) generals eventually decide on a value.
- **Agreement:** All loyal generals agree on the same value.
- **Validity:** If the commanding general is loyal, then every loyal lieutenant decides on the value originally sent by the general.

4.2 The Recursive OT(t) Algorithm

Lamport’s algorithm, often referred to as OT(t), is defined recursively:

1. **Base Case (OT(0)):**
   When *t* = 0 (i.e., no traitors), the commander sends its order to every lieutenant. Each lieutenant simply adopts the order it receives.

2. **Recursive Case (OT(t), t > 0):**

   - **Step 1:** The commander sends its order to each lieutenant.
   - **Step 2:** For each lieutenant *i*, it receives an order vi from the commander.
   - **Step 3:** Lieutenant *i* then acts as a commander in its own instance of OT(t-1) and sends the value vi to every other lieutenant.
   - **Step 4:** Each lieutenant collects the orders received from all other lieutenants and uses a majority function to decide on a final value.

This recursive structure allows the loyal generals to “vote” on the received orders, ensuring that even if some lieutenants are faulty and send conflicting orders, the loyal ones can achieve consensus by using a majority rule. The algorithm requires at least 3m+1 generals to tolerate *m* faulty nodes.

4.3 Challenges and Detailed Explanation

- **Message Propagation:**
  In the first round (OT(0)), the general sends its order to every lieutenant. For OT(t), each lieutenant further relays the order they received (possibly modified by traitors) to all others. With every additional level of recursion, the number of messages grows rapidly, ensuring that every lieutenant eventually gathers enough information to compute a majority value.

- **Fault Injection:**
  Faulty generals (traitors) may invert or alter the order when relaying messages. For example, in our implementation, a traitor may flip "ATTACK" to "RETREAT" for certain recipients based on their position in the peers list. This creates conflicting data among the loyal generals. However, because each lieutenant gathers multiple reports (through multiple rounds), they can use the majority of received orders to filter out the erroneous ones.

- **Majority Decision:**
  After gathering all the messages, each lieutenant applies a majority rule to decide on a final value. This decision process guarantees that if the commanding general is loyal, the final decision is the order originally sent. Even if the general is faulty, all loyal lieutenants will agree on a common value, though it might not be the value the commander intended.

- **Rounds of Communication:**
  The algorithm involves $t + 1$ rounds of messaging. In our distributed implementation, these rounds are simulated through recursive gRPC calls. We also map the recursion level to a "round" number in the logs so that the very first communication round (when m is maximum) prints as "round 1" and subsequent rounds are numbered accordingly.

5. Our Distributed Implementation

5.1 Architecture Overview

Our solution consists of three main components:

- **Distributed Server (`distributed_server.py`):**
  Each general is implemented as an independent gRPC server. The server executes the OM algorithm recursively. Intermediate steps (such as receiving an order, flipping an order due to traitorous behavior, and forwarding orders) are logged into a dedicated file (`general_<id>_intermediate.log`). The logs use a mapping (printed_round = 4 - t - 1) to represent the rounds clearly.

- **Scale Server (`scale_server.py`):**
  This script automates the process of starting the distributed servers. It assigns unique ports (starting from a base port) and spawns a server for each general. It also designates traitors based on command-line input. This allows easy scaling to any number of generals.

- **Distributed Client (`distributed_client.py`):**
  The client triggers the OM algorithm by contacting the commander (general 0) and then collects the vote tallies from all generals through a dedicated RPC. Before each run, it resets the log files on all servers (via the ResetLog RPC) so that logs from previous runs do not mix with current results. Finally, it prints the vote tallies for all generals and computes the final consensus among the loyal generals.

5.2 Intermediate Log Files

Each server writes its detailed intermediate steps to a log file named `general_<id>_intermediate.log`. These logs include:

- Order receptions and the corresponding round (using our mapped round values).
- Messages indicating when a traitor flips an order.
- Details of messages sent and received during the recursive execution of the algorithm.

This detailed logging helps trace the execution of the algorithm and verify that the consensus is being reached despite faulty behavior.

5.3 Scaling and Port Assignment

The `scale_server.py` script accepts command-line parameters such as:

- **--num_generals:** Total number of generals (e.g., 10).
- **--traitors:** Comma-separated indices of traitors (e.g., 3,4,7).
- **--base_port:** The starting port number (e.g., 50051).

It then spawns each server on a unique port (e.g., from 50051 to 50060) and passes the full list of peers to every server. This design simulates a fully distributed network where every general can directly communicate with every other general.

---

6. Testing

6.1 Test Setup

To test the distributed solution, you first spawn the server processes and then run the client. For example:

Spawn Servers:
Use the following command to spawn 10 generals with generals 3, 4, and 7 designated as traitors:

```
python scale_server.py --num_generals 10 --traitors 3,4,7 --base_port 50051
```

1. This command automatically assigns ports 50051 to 50060 and creates (or resets) the intermediate log file for each general.

Run the Client:
Trigger the OM algorithm and collect vote tallies by running:

```
python distributed_client.py --order ATTACK --t 3 --num_generals 10 --base_port 50051 --generals l,l,l,t,t,l,l,t,l,l
```

2. The client will first reset all logs, then initiate the algorithm via the commander (general 0), and finally gather and display the vote tallies from all generals.

6.2 Expected Client Output

A sample output might be:

```
Vote tallies from generals:

General 0:

  ATTACK: 314

  RETREAT: 262

General 1:

  ATTACK: 412

  RETREAT: 164

General 2:

  ATTACK: 314

  RETREAT: 262

General 3:

  ATTACK: 412

  RETREAT: 164

General 4:

  ATTACK: 314

  RETREAT: 262

General 5:

  ATTACK: 412

  RETREAT: 164
```

```
General 6:
  ATTACK: 314
  RETREAT: 262
General 7:
  ATTACK: 412
  RETREAT: 164
General 8:
  ATTACK: 314
  RETREAT: 262
General 9:
  ATTACK: 412
  RETREAT: 164
Final consensus reached: ATTACK
```

This output shows the vote tallies of each general and confirms that the final consensus among loyal generals is “ATTACK.”

---

7. Conclusion

Our distributed implementation of the Byzantine Generals Problem showcases a fault-tolerant consensus protocol in a realistic, decentralized system. By using independent gRPC servers, automated scaling with `scale_server.py`, detailed logging, and the recursive OM algorithm, the system successfully reaches consensus even in the presence of traitorous nodes. The in-depth explanation of Lamport’s algorithm underscores the importance of message exchange, fault detection, and majority-based decision-making in achieving Byzantine Fault Tolerance. The provided test case demonstrates that with 10 generals (and 3 traitors), loyal generals converge on the final decision “ATTACK.”This report encapsulates the theoretical foundations of Byzantine Fault Tolerance, a detailed explanation of Lamport’s algorithm, and the practical implementation of our distributed solution.