Tribhuvan University

Institute of Engineering

Central Campus, Pulchowk

DS Lab 8

Implementing algorithm for Byzantine generals' problem

Submitted By:

Himanshu Pradhan(077BCT030)

Janam Shrestha(077BCT032)

Mejan Lamichhane(077BCT047)

Submitted To:

Department of Electronics and Computer Engineering

Pulchowk Campus, Lalitpur

Date: 2024/08/12

Theory

Byzantine Generals' Problem

The Byzantine Generals' Problem is a fundamental issue in distributed computing that underscores the difficulties in achieving reliable security and consensus across a network of independent systems. Imagine a scenario involving several military generals, each commanding a separate division of an army, who need to coordinate on a unified battle strategy. However, the problem is complicated by the possibility that some of these generals may be treacherous and intentionally attempt to disrupt the consensus by sending conflicting or misleading information. The central challenge is to devise an algorithm that ensures the loyal generals can agree on a single plan of action despite the presence of these deceitful elements.

Steps Involved

1. Each node in the network starts by proposing a value that it believes should be decided upon. This proposed value is then communicated to all other nodes in the system.
2. During this phase, each node broadcasts its proposed value to every other node. This step involves gathering the proposed values from all nodes, thus creating a pool of potential values.
3. Each node aggregates the values it has received from other nodes into a list or vector. This collection of values is crucial for further processing and analysis.
4. In this round, nodes share their aggregated lists of values with all other nodes. This allows each node to compile comprehensive data from the entire network.
5. Nodes then use a majority voting mechanism on the collected data to determine which value appears most frequently. If no single value garners a majority, a default or fallback value is chosen as a compromise.
6. After applying the majority voting process to the aggregated values, each node reaches a final decision. Ideally, all non-faulty nodes will converge on the same value, thereby achieving consensus despite the potential presence of faulty or malicious nodes.

Implementation

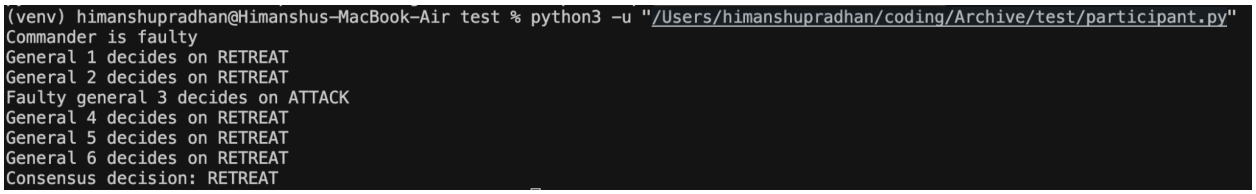
```
import random
import argparse

ATTACK = "ATTACK"
RETREAT = "RETREAT"

class Node:
   def __init__(self, input_value, id):
       self.input_value = input_value
       self.output_value = ""
       self.process_ids = {}
       self.children = []
```

```
        self.id = id

    def send_message(self, id, faulty_generals):
        if id in self.process_ids:
            return None

        # Check if the current node is faulty
        is_faulty = self.id in faulty_generals

        # Determine the message to send based on whether the node is faulty
        order = self.opposite_order(self.input_value) if is_faulty else
self.input_value

        # Update the process ID record
        updated_process_ids = self.process_ids.copy()
        updated_process_ids[id] = 1

        return Node(order, id)

    def opposite_order(self, order):
        return RETREAT if order == ATTACK else ATTACK

    def decide(self):
        # If this node has no children, its decision is based on its input value
        if not self.children:
            self.output_value = self.input_value
            return self.output_value

        decision_counts = {}
        for child in self.children:
            decision = child.decide()
            decision_counts[decision] = decision_counts.get(decision, 0) + 1

        # Determine the majority decision
        self.output_value = self.majority_decision(decision_counts)
        return self.output_value

    def majority_decision(self, decision_counts):
        return ATTACK if decision_counts.get(ATTACK, 0) >
decision_counts.get(RETREAT, 0) else RETREAT
```

```
def generate_faulty_indexes(num_faulty_generals, num_generals):
    # Randomly choose indices for faulty generals
    return set(random.sample(range(num_generals), num_faulty_generals))

def byzantine_generals(num_generals, num_faulty_generals, order):
    faulty_indexes = generate_faulty_indexes(num_faulty_generals, num_generals)

    commander = Node(order, 0)
    if 0 in faulty_indexes:
        print("Commander is faulty")

    queue = [commander]
    current_depth = 0
    nodes_at_current_depth = 1
    nodes_at_next_depth = 0

    # Perform a breadth-first search to distribute messages and create nodes
    while queue:
        node = queue.pop(0)
        nodes_at_next_depth += num_generals - current_depth
        nodes_at_current_depth -= 1
        if nodes_at_current_depth == 0:
            current_depth += 1
            if current_depth >= num_faulty_generals:
                break
            nodes_at_current_depth = nodes_at_next_depth
            nodes_at_next_depth = 0

        for i in range(1, num_generals):
            child_node = node.send_message(i, faulty_indexes)
            if child_node is None:
                continue
            node.children.append(child_node)
            queue.append(child_node)

    # Determine the final consensus decision
    commander.output_value = commander.decide()

    for i, general in enumerate(commander.children):
        if i + 1 in faulty_indexes:
            print(f"Faulty general {general.id} decides on {general.output_value}")
```

```
        else:
            print(f"General {general.id} decides on {general.output_value}")

    print(f"Consensus decision: {commander.output_value}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simulation of the Byzantine
Generals Problem")
    parser.add_argument("-G", type=int, default=7, help="Total number of generals
including the commander")
    parser.add_argument("-M", type=int, default=2, help="Number of faulty generals")
    parser.add_argument("-O", type=str, default=ATTACK, choices=[ATTACK, RETREAT],
help="Initial order (ATTACK or RETREAT)")

    args = parser.parse_args()
    num_generals = args.G
    num_faulty_generals = args.M
    order = args.O

    # Ensure the number of faulty generals is not excessive
    if 3 * num_faulty_generals > num_generals:
        print("Too many faulty generals; cannot have more than a third of the total
generals faulty")
    else:
        byzantine_generals(num_generals, num_faulty_generals, order)
```

Output



The output from the Byzantine Generals Problem simulation reveals an important aspect of the algorithm's behavior in the presence of faulty components. The message "Commander is faulty" indicates that the commander (general 0) is faulty, meaning it does not adhere to the correct protocol and might send misleading information. Despite this, the decision-making process among the remaining generals proceeded as follows: Generals 1, 2, 4, 5, and 6, all of whom are non-faulty, decided to RETREAT. On the other hand, General 3, who is faulty, decided to ATTACK, reflecting its deviation from the expected behavior. The final consensus decision, therefore, was RETREAT, which emerged as the majority choice despite the presence of a faulty general. This outcome demonstrates the effectiveness of the consensus algorithm in achieving agreement among the majority, even when some components of the system fail or act maliciously.

Conclusion

The simulation demonstrates that despite a faulty commander and some generals deviating from the expected behavior, the Byzantine Generals Algorithm successfully reached a consensus. The majority decision of RETREAT prevailed, highlighting the algorithm's robustness in achieving agreement even in the presence of faulty components.