Byzantine Generals’ Problem:

Decrypting Leslie Lamports’ Solution

This blog explains Leslie Lamport’s solution to the Byzantine Generals’ problem that uses unsigned messages, providing a step-by-step walkthrough with visual examples.

Motivation

Distributed systems are everywhere in our modern digital infrastructure—from cloud computing to blockchain networks. However, a fundamental challenge in these systems is achieving consensus when some components might be faulty or malicious.

The Byzantine Generals’ Problem, introduced by Leslie Lamport, Robert Shostak, and Marshall Pease in 1982, elegantly captures this challenge through a military metaphor. Understanding this problem and its solution is crucial for anyone working with distributed systems, blockchain technology, or fault-tolerant computing.

This blog focuses specifically on the solution using unsigned messages—the most challenging scenario where messages carry no authentication. We’ll examine a non-trivial instance with the most adversarial case: where the commander is a traitor and strategically sends conflicting commands to create maximum confusion.

The unsigned message solution is particularly fascinating because it operates under the loosest guarantees. Unlike solutions that rely on cryptographic signatures or other forms of authentication, this approach must achieve consensus through pure logic and majority reasoning, making it both intellectually stimulating and practically relevant.

Introduction

Imagine a scenario where several Byzantine generals have surrounded an enemy city. They must decide collectively whether to attack or retreat. To succeed, all loyal generals must agree on the same plan of action—either all attack or all retreat. A split decision would be disastrous.

The generals can only communicate through messengers, and some generals might be traitors (including potentially the commanding general) who seek to prevent the loyal generals from reaching agreement. These traitors can:

- Send different messages to different generals
- Collude with other traitors
- Lie about messages they claim to have received

Our challenge is to design an algorithm that ensures all loyal generals reach the same decision despite these traitors. This is the essence of the Byzantine Generals' Problem.

The solution we'll explore doesn't rely on message signatures or authentication. Instead, it uses a recursive message-passing approach called Oral Message (OM) algorithm that can withstand up to t traitors in a system with at least $3t + 1$ generals.

Terminology

Before diving into the solution, let's establish some terminology:

- **Generals**: The participants in the system, denoted as G_i where i is the general's identifier.
- **Commander**: The general who initiates the decision process (usually G_0).
- **Lieutenants**: All other generals who must follow the commander's decision.
- **Loyal ($\mathcal{L}$)**: Generals who follow the algorithm correctly.
- **Traitors ($\mathcal{T}$)**: Generals who may behave arbitrarily and maliciously.
- **Commands**: The possible decisions, typically Attack (**A**) or Retreat (**R**).
- **OM(m)**: The Oral Message algorithm with recursion depth m.

The Byzantine Generals' Problem has two key requirements:

1. **Agreement**: All loyal generals must agree on the same decision.
2. **Validity**: If the commander is loyal, all loyal generals must follow the commander's order.

Lamport proved that with oral (unsigned) messages, we need at least $3t + 1$ generals to tolerate t traitors, and the algorithm requires $t + 1$ rounds of message exchanges.

The Algorithm

The Oral Message algorithm (OM) works recursively:

OM(0): The commander sends a value to all lieutenants, and each lieutenant uses the value received.

OM(m), for $m > 0$:

1. The commander sends a value to each lieutenant.
2. Each lieutenant acts as a commander in the OM($m - 1$) algorithm to share the value they received with all other lieutenants.
3. Each lieutenant determines the majority value reported for each general and uses that as their final decision.

The algorithm guarantees correct operation if $n \geq 3t + 1$ where n is the total number of generals and t is the maximum number of traitors.

The recursive nature of this algorithm creates a tree of message exchanges that grows exponentially with m. For a system with n generals, the message complexity is $O(n^{m+1})$, which makes it practical only for small values of t and thus m.

The Problem

Let's examine a specific instance of the Byzantine Generals' Problem:

- **Total generals**: $N = 7$
- **Traitors**: $t = 2$
- **Loyal generals**: $N - t = 5$

In our scenario:

- Commander G_0 is a traitor
- Lieutenant G_1 is also a traitor
- Lieutenants G_2 through G_6 are loyal

This is a particularly challenging case because the commander, who initiates the algorithm, is malicious. The commander's strategy is to create maximum confusion by sending conflicting commands:

- Commander G_0 sends Attack (A) to generals G_2, G_4, and G_6
- Commander G_0 sends Retreat (R) to generals G_1, G_3, and G_5

Additionally, the second traitor G_1 will also attempt to confuse the loyal generals by inconsistently relaying messages during the second round.

With $t = 2$ traitors, we need at least $3t + 1 = 7$ generals, which is exactly what we have. According to the theory, OM(2) should be sufficient to achieve consensus among loyal generals. Let's see how this works in practice.

The Solution

We'll implement the OM(2) algorithm for our scenario with 7 generals and 2 traitors. Since $m = 2$, we'll need two rounds of message exchange.

Round 1

In the first round, the commander G_0 (who is a traitor) sends commands to all lieutenants. Since the commander is a traitor, they can send different commands to different lieutenants.

Figure 1 illustrates what happens in Round 1:

G_0 sends its chosen messages to the lieutenants.

As shown in the diagram, the traitor commander G_0 sends:

- Retreat (R) to generals G_1 (traitor), G_3, and G_5
- Attack (A) to generals G_2, G_4, and G_6

At this point, each lieutenant only knows what the commander told them directly. The table in Figure 1 represents this initial knowledge state, with each lieutenant recording their received command on the diagonal. Notice how the equal distribution of Attack and Retreat commands creates a deadlock if lieutenants were to make decisions based solely on this first round.

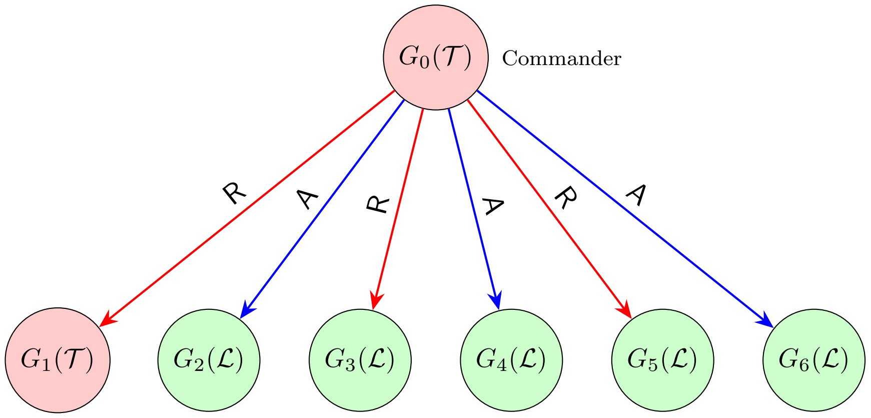

Figure 1: Round 1: Traitor Commander G_0 sends messages.

$G_1(\mathcal{T})$	$G_2(\mathcal{L})$	$G_3(\mathcal{L})$	$G_4(\mathcal{L})$	$G_5(\mathcal{L})$	$G_6(\mathcal{L})$
R					
	A				
		R			
			A		
				R	
					A

The grid above shows the initial values v_i that each lieutenant G_i (represented by row and column headers) receives from Commander G_0 and stores. This value is placed on the diagonal cell (G_i, G_i). Off-diagonal cells are empty, to be filled by messages exchanged between lieutenants in later algorithm phases (e.g., OM(m-1) rounds).

Round 2

In the second round, each lieutenant acts as a sub-commander and tells all other lieutenants what command they received from the original commander in Round 1.

Figure 2 illustrates the message flow in Round 2:

In the second round, each lieutenant shares with all other lieutenants what they received from the commander in Round 1.

Receiver

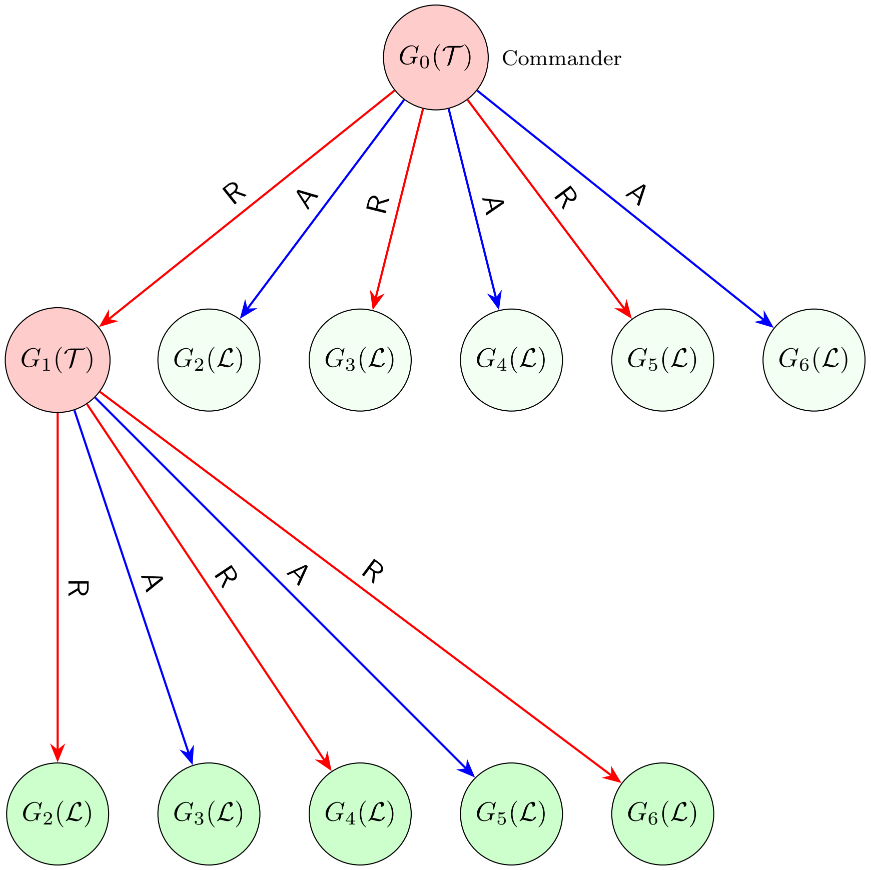

Figure 2: Round 2: Lieutenants exchange what they received from the commander. Traitor G_1 sends conflicting messages.

		G_2	G_3	G_4	G_5	G_6
Sender	G_2	**R**	R	R	R	R
	G_3	A	**A**	A	A	A
	G_4	R	R	**R**	R	R
	G_5	A	A	A	**A**	A
	G_6	R	R	R	R	**R**
	Majority	**R**	**R**	**R**	**R**	**R**

Legend: Rows (left) are senders, columns (top) are receivers. Each cell shows the message the sender tells the receiver that it received from G_1. Yellow diagonal: sender's own message from G_1. Blue = Attack (A), Red = Retreat (R).

$G_1(\mathcal{T})$	$G_2(\mathcal{L})$	$G_3(\mathcal{L})$	$G_4(\mathcal{L})$	$G_5(\mathcal{L})$	$G_6(\mathcal{L})$
R	R	R	R	R	R
	A				
		R			
			A		
				R	
					A

After Round 2, each loyal lieutenant can determine the consensus by taking a majority vote of all reported values. Despite the conflicting information from the traitors, all loyal lieutenants reach the same conclusion: Retreat (R).

The second traitor G_1 (who received "Retreat" from G_0) continues the deception by:

- Telling some lieutenants that it received "Attack" from the commander
- Telling other lieutenants that it received "Retreat" from the commander

Meanwhile, all loyal lieutenants honestly relay the commands they received. After this round, each lieutenant has:

- Their original command from the commander (from Round 1)
- Reports from all other lieutenants about what they claim to have received

The table in Figure 2 shows what each lieutenant knows after Round 2. Each row represents a sender (a lieutenant sharing what they received from the commander), and each column represents a receiver. The cell values show what message was conveyed.

Now, each loyal lieutenant can determine what command to follow by taking a majority vote of all reported values. As shown in the "Majority" row of the table, the consensus among all loyal lieutenants is to "Retreat" (R).

This demonstrates the power of the OM(2) algorithm: despite having a traitor commander and a traitor lieutenant, all loyal lieutenants reach the same conclusion. The Byzantine Generals' Problem is solved for this instance!

Conclusion

The Byzantine Generals' Problem represents one of the fundamental challenges in distributed computing. Through our examination of the Oral Message algorithm (OM), we've seen how it's possible to achieve consensus even in the presence of malicious actors.

Key insights from our exploration:

- With n generals and at most t traitors, consensus requires $n \geq 3t + 1$ and $m \geq t$ rounds of communication.
- The recursive nature of the algorithm allows loyal generals to filter out inconsistent information.
- Even with a traitor commander sending contradictory commands, loyal generals can still reach agreement.
- The solution works without requiring any form of message authentication or signatures.

This algorithm laid the foundation for many fault-tolerant distributed systems we rely on today, from cloud databases to blockchain networks. Modern consensus protocols like Practical Byzantine Fault Tolerance (PBFT) and those used in blockchain systems are direct descendants of Lamport's original solution.

Understanding these principles not only helps in designing robust distributed systems but also provides insight into the fundamental limits of what's achievable in the presence of Byzantine failures.

What other distributed consensus challenges are you curious about? Share your thoughts and questions in the comments below!