DAT085 Report

Shivneshwar Velayutham

May 2, 2024

1 Introduction

This report details the implementation and testing of a self-stabilizing Byzantine fault-tolerant algorithm for repeated reliable broadcast, as proposed by Duvignau et al. [7]. Our goal is to better understand and explore the practical implications of integrating such an algorithm into real-world systems.

Structured methodically, this report provides an in-depth look into the implementation process, experimental design, findings, and discussions. While the theoretical aspects of the algorithm have been extensively covered in Duvignau et al.'s work [7], our focus here is on bridging theory and real-world application.

The implementation section breaks down the technical aspects of integrating the algorithm, highlighting the addition of key components necessary for communication among distributed processes without shared memory.

Moving forward, we delve into the experiments designed to scrutinize the algorithm's behavior and performance. These experiments range from examining message latency in relation to the number of transmitter processes to exploring the impact of message size.

The results section presents our empirical findings, supported by visual aids to illustrate trends and observations. We also engage in discussions to interpret these findings and their implications.

Overall, this report provides a comprehensive understanding of the implemented broadcast algorithm and its practical implications in distributed systems. It aims to contribute to the knowledge base of researchers and practitioners in this field.

2 Related Work

Self-stabilization is a property in distributed computing where a system, despite starting from any arbitrary state, is able to autonomously converge to a desirable state without external intervention. This property ensures that the system can recover from transient faults or disturbances, maintaining correctness and stability over time. By enabling systems to self-correct and adapt to changing conditions, self-stabilization plays a critical role in the design of robust and fault-tolerant distributed systems. Dolev [5] describes a protocol called stop-and-wait which is a self-stablilizing algorithm that simulates shared memory. This protocol will be used in this project and described more in detail in further section.

Byzantine Reliable Broadcast (BRB) ensures messages are reliably delivered to all correct nodes, even in the presence of faults or malicious behavior. [9] and [8] are some example of BRBs. The term "repeated" signifies the capability for processes to persistently broadcast messages, ensuring continuous communication beyond individual broadcast events.

3 Implementation

The broadcast algorithms, as documented in [7], relied on a shared memory approach where all processes communicated through a shared register. Alongside the implementation of the broadcast algorithm, this project introduces a physical shared memory simulator to facilitate communication among distributed processes lacking a shared memory infrastructure. The implementation consists of three concurrent threads:

1. **Communication Thread**: This thread simulates shared memory functionality.
2. **Broadcast Algorithm Thread**: Responsible for broadcasting and delivering messages.
3. **User Thread**: Invoked to perform broadcast and message delivery operations.

3.1 Communication Thread

The communication thread operates using the bounded stop-and-wait protocol, a self-stabilizing algorithm from [5] suitable for simulating shared memory. This protocol employs token passing, wherein a token containing the message to be shared, along with a message counter, is transmitted from a sending process to a receiving process. Each process in the broadcasting group functions as both a sender and a receiver, sharing its current state with peers and learning the latest state from them. Below is the description of the bounded stop-and-wait protocol:

Listing 1: Stop and Wait Protocol

```
Sender:
upon timeout
    send(counter)
upon message arrival
begin
    receive(MsgCounter)
    if MsgCounter >= counter then
    begin
        counter := MsgCounter + 1
        send(counter)
    end
    else send(counter)
end

Receiver:
upon message arrival
begin
    receive(MsgCounter)
    if MsgCounter != counter then token received
        counter := MsgCounter
    send(counter)
end
```

Upon receiving a token, each process replies with a message containing the token and the contents of the shared register. Write operations are executed locally in memory, and the communication thread disseminates memory contents to all peers. Read operations involve several steps:

1. Receive a token message from a peer.
2. Receive the same token from the peer again.
3. Upon receiving the second token, update the peer's state by writing the contents of the received message to memory.

The protocol is termed "bounded" because the message counter is limited, typically represented by an unsigned 64-bit integer. Messages are transmitted as User Datagram Protocol (UDP) [1] messages in JSON format [4], known for its human-readable text and ease of data interchange.

An essential aspect of the stop-and-wait protocol is calculating the timeout duration for message reception. This project adopts TCP's [2] retransmission and timeout estimation mechanism [10], which employs an Exponentially Weighted Moving Average (EWMA) of past round-trip times. To ensure reasonable waiting times, a maximum and minimum timeout threshold is enforced.

The communication thread's implementation is outlined in Listing 2.

Listing 2: Communication Thread

```
while true do
    for each process p in process list do
        if timeout reached for p then
            retransmit message to p
            update timeout time for p
    next_timeout = get_earliest_timeout_from_list_of_processes(process_list)
    connection.setReadDeadline(next_timeout)
    message, err = connection.BlockingRead()
    if err == timeout then
        continue
    if message from process in process list then
        unserialize message
        process message based on stop and wait algorithm
        update timeout time for process if necessary
```

3.2 Broadcast Algorithm Thread

Listing 3: Broadcast Algorithm Thread

```
while true do
    execute a single iteration of the algorithm's do forever loop
    substitute message sending with local writing to buffer
    update the value of my register for the communication thread to pick up
    for each process p in process list do
        if process hasn't been recycled recently then
            if all the messages in the channel have been flushed then
                retrieve the latest message for process
        else
            retrieve the latest message per process
        update the latest state of other processes for the next iteration
```

The broadcast algorithm thread's implementation is outlined Listing 3.

3.3 User Thread

The user thread carries the dual responsibility of both broadcasting and delivering messages. Its functionality is tailored to suit the specific use case, which, in our scenario, involves conducting experiments using the system. During our experiments, where each process assumes the role of a receiver, the user thread persistently endeavors to deliver all incoming messages.

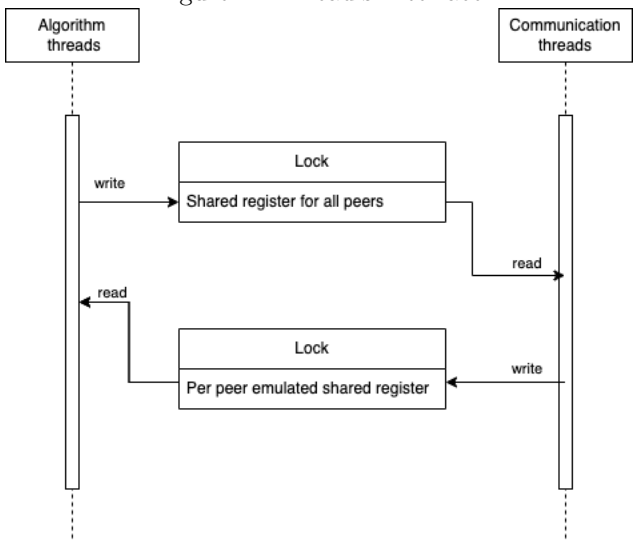
In cases where the process also functions as a transmitter, the user thread follows a cyclical pattern: it broadcasts a message and patiently awaits its delivery. Once the message has been successfully delivered,

the thread proceeds to broadcast a new message. This iterative process continues until the process is terminated.

3.4 Thread Interaction and Lock Mechanisms

Figure 1 describes on how the threads interact with each other using locks. Shared locks are used across all the threads to prevent concurrent writes on shared objects in memory.

Figure 1: Thread's Interface



4 Evaluation

4.1 Research Question

The primary research question aims to validate the correctness of the algorithm and ensure it functions as intended. Following this validation, the next step involves establishing metrics to evaluate the algorithm's performance.

4.1.1 Evaluation Criteria

In the context of broadcasting, one crucial metric is message latency, which denotes the average time taken for a broadcasted message to be delivered. In the algorithm [7], message latency is defined as the average time elapsed from a process invoking the brbBroadcast(message) function until all other processes successfully invoke the brbDeliver(sender process id) function.

Additionally, the impact of message size on message latency is evaluated. Considering the limitations of message size in UDP we only take into account the extremes. The analysis focuses on the smallest and largest possible message sizes. By examining these extremes, we gain insights into how message size influences latency in the system.

4.2 Evaluation Environment

Experiments are conducted on OpenStack servers provisioned by the researcher, all of which are Linux [3] machines capable of intercommunication. In total, there are 10 processes spread across the servers, with

each server hosting a subset of these processes. Each process functions as a receiver, and the experiments aim to investigate the impact of message latency as the number of transmitter processes varies.

At the onset, each process is aware of its peers designated as transmitters or receivers. As receivers, processes attempt to deliver broadcasted messages from transmitter processes. Transmitter processes continually broadcast messages, waiting for successful delivery before initiating the broadcast of a new message. This process iterates until manual intervention halts the processes.

Each broadcasted message includes a timestamp denoting the transmission time. Upon successful delivery, the latency of the message is calculated as the difference between the delivery time and the transmission time. Message latency represents the average time taken for a broadcasted message to be delivered.

4.3 Experiments

4.3.1 Experiment 1

The experiments involve varying the number of transmitter processes from 1 to 10 to observe the corresponding impact on message latency. It is expected that increasing the number of transmitters will lead to higher latency. It is hypothesized that impact on latency might linearly increase as the number of the transmitters increase.

4.3.2 Experiment 2

Additionally, an experiment is conducted to evaluate the effect of message size on latency. Given the maximum size limit of UDP messages, two scenarios are explored: one with small messages and another with the largest possible message sizes. It is hypothesized that while the impact on latency may be minimal for small messages, larger messages may experience a slight increase in latency compared to small messages, albeit not significantly noticeable.

5 Results

The results from the experiments are visualized in Figure 2 and Table 1.

Upon analysis, we noted a consistent trend: a nearly linear increase in message latency as the number of transmitter processes increased, regardless of message size. Additionally, we noted a slight uptick in message latency for experiments involving large messages compared to those with small messages. We also notice that the uptick is considerable lower when the number of transmitters are low and the uptick seems to increase as the number of transmitter increase.

The nearly linear increase in message latency can be attributed to the number of messages being broadcasted simultaneously, which consequently leads to longer delivery times. The uptick in message latency with large messages can be explained by the increase in the amount of time to process the messages. The uptick being considerable smaller when the number of transmitters is low, may be attributed to the algorithm's behavior where the message size limit is reached only when all processes have echoed the messages and are about to deliver. Once a new broadcast starts, the message transmitted becomes small again, reducing the amount of time the message size remains large.

Overall, these findings matches our hypothesis and sheds light on the nuanced interplay between message size, number of transmitter processes, and message latency in our system.

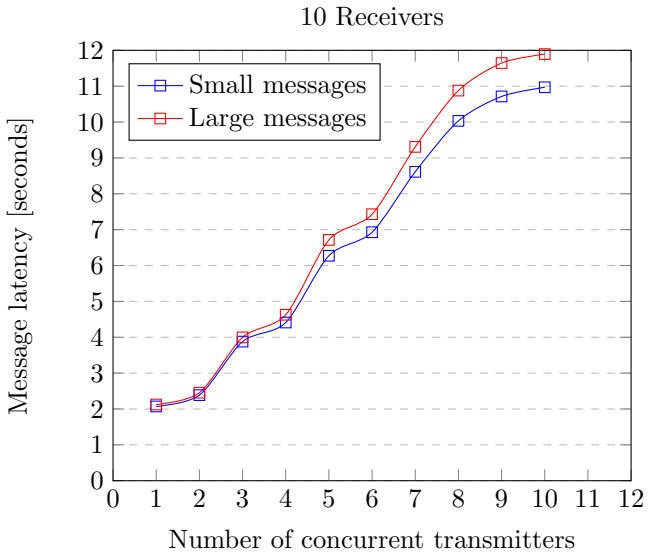


Figure 2: Messages latency vs number of concurrent transmitters

Number of concurrent transmitters	**Message latency for smallest messages**	**Message latency for largest messages**
1	2.07 seconds	2.13 seconds
2	2.39 seconds	2.45 seconds
3	3.88 seconds	3.99 seconds
4	4.41 seconds	4.63 seconds
5	6.27 seconds	6.71 seconds
6	6.93 seconds	7.43 seconds
7	8.61 seconds	9.31 seconds
8	10.03 seconds	10.88 seconds
9	10.71 seconds	11.64 seconds
10	10.97 seconds	11.9 seconds

Table 1: Impact of message size on message latency with 10 receivers

6 Discussion

In this project, we've effectively implemented a self-stabilizing Byzantine fault-tolerant algorithm for repeated reliable broadcast, proposed by Duvignau et al. [7], using the Go programming language [6]. To facilitate the implementation of the algorithm, a low-level communication primitive was developed from scratch.

Prior analyses of the algorithm were primarily theoretical, relying on hand-written proofs. Thus, our project's significance lies in its practical validation and evaluation of the algorithm in real systems for the first time.

7 Conclusion

In conclusion, this project has successfully implemented and evaluated a self-stabilizing Byzantine fault-tolerant algorithm for repeated reliable broadcast, as proposed by Duvignau et al. [7]. By validating the algorithm's correctness and assessing its performance through experiments, this project contributes valuable insights to self-stablizing byzantine fault-tolerant repeated reliable broadcast.

The practical implementation and evaluation of the algorithm provide a deeper understanding of its behavior and performance in real-world scenarios. This project underscores the importance of bridging theoretical concepts with practical implementation, paving the way for the development of more robust and reliable distributed systems. Moving forward, the findings and methodologies presented in this report can serve as a foundation for further research and development in this field.

References

[1] User Datagram Protocol. RFC 768, Aug. 1980.

[2] Transmission Control Protocol. RFC 793, Sept. 1981.

[3] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[4] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, Dec. 2017.

[5] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[6] A. A. Donovan and B. W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015.

[7] R. Duvignau, M. Raynal, and E. M. Schiller. Self-stabilizing byzantine fault-tolerant repeated reliable broadcast. *Theor. Comput. Sci.*, 972:114070, 2023.

[8] R. Guerraoui, J. Komatovic, P. Kuznetsov, Y. Pignolet, D. Seredinschi, and A. Tonkikh. Dynamic byzantine reliable broadcast. In Q. Bramas, R. Oshman, and P. Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[9] A. Mostéfaoui and M. Raynal. Intrusion-tolerant broadcast and agreement abstractions in the presence of byzantine processes. *IEEE Trans. Parallel Distributed Syst.*, 27(4):1085–1098, 2016.

[10] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.