

**A SIMULATIONS-BASED PERFORMANCE EVALUATION OF TOTAL ORDER
PROTOCOLS**



Agbaeze Ejem

**School of Computing
Newcastle University**

**This dissertation is submitted for the degree of
Doctor of Philosophy**

Computing Science

January 2025

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others except as specified in the text and Acknowledgements.

This thesis incorporates work that has been published in conference proceedings. The publications are as follows:

- 1 A. Ejem and P. Ezhilchelvan. Design and Performance Evaluation of a High-Throughput and Low-Latency Total Order Protocol. In: 38th Annual UK Performance Engineering Workshop (UKPEW 2022) [Online].
- 2 A. Ejem and P. Ezhilchelvan. Design and Performance Evaluation of Raft Variations. In: 39th Annual UK Performance Engineering Workshop (UKPEW 2023) [Online].

Agbaeze Ejem

January 2025

Acknowledgement

I would want to offer my sincere gratitude to everyone who has helped and inspired me along this challenging but worthwhile path to finishing my thesis.

Firstly, I sincerely thank my supervisor, Paul Ezhilchelvan, whose direction, knowledge, and constant support have been crucial in forming this research. His insightful advice, helpful criticism, and confidence in my abilities have been a continuous source of motivation for me to pursue greatness in my studies. I want to express my sincere gratitude for allowing me to develop despite my little computer science knowledge and for believing in me. Working with you over the past few years has been an incredible delight and a transformational experience. My profound gratitude goes out to my devoted wife, Chinenye, for her unending tolerance, compassion, and support throughout this study. My pillars of strength have been her unshakable confidence in me and her constant assurance, which have helped me to endure even in the most trying circumstances.

I am also incredibly grateful for my exceptional children, Favour, Samuel and Grace, whose contagious joy and innocent smiles helped me manage my academic goals and family obligations.

I also want to thank my siblings, David (Major), Ucha and Emmanuel, for their words of support, motivation, and encouragement during my academic endeavours. It has meant so much to me that you continue to believe in and support me.

I also want to thank Nkisi-orji Ikechukwu and Jack Wauby for their insightful assistance in writing the simulation codes for this thesis.

I also thank my kind sponsor, the Petroleum Tertiary Development Fund (PTDF). Thanks to your financial assistance, I could concentrate on my studies without being constrained. Your support of the value of education and research is greatly appreciated.

Abstract

Process replication on fail-independent computers is a fundamental requirement for incorporating fault tolerance and thereby guaranteeing service availability in the event of computer failures in distributed systems. A given service request is executed on *all* process replicas so that even if some replicas crash, the request execution at surviving ones will ensure that the client of that request receives the response. For replication to be effective, however, all replicas must process all requests in the *same* order so that they all undergo identical state transitions and produce identical responses for a given request, of which any one can be given to the client. To ensure this identical ordering, replication employs a total order protocol that is distributed and crash-tolerant to obtain an ordering sequence on the requests directed at the replicated system.

Total order protocols can be typically classified as leader-free and leader-based. LCR and Raft order protocols are well-known leaderless and leader-based categories respectively. The LCR protocol arranges replicas on a logical ring with inter-replica communication supported by unidirectional flow of messages. Due to this structure, it has been proven to offer the highest achievable throughput. Raft is a widely used, leader-based, total order protocol that is simpler and easier to understand than other leader-based protocols and provides a foundation for system implementation. In Raft, one process is designated as the leader and others as followers and the latter obey the ordering decided by the leader; this leads to messages flowing to/from the leader as if the processes are arranged in a star topology with the leader at the centre of topology.

Having analysed LCR and Raft protocols [2, 3], we observe that LCR is designed with some potentially performance-limiting assumptions that may not lead to smaller latencies when several processes concurrently forward requests for ordering; these assumptions are: (i) use of a vector clock with one integer for every replica, and (ii), employing a fixed idea of "last" process to order concurrent messages. In addition, in the Raft protocol, all requests are directed at the leader and hence the load at the leader becomes a performance bottleneck as the client request arrival rate at the leader process increases. This bottleneck is exacerbated by the crash

tolerance requirement that the leader requires a quorum of acknowledgements from followers before confirming the order it initially placed on a single client request.

This thesis seeks to eliminate these performance-limiting design assumptions and propose demonstrably efficient alternative designs. To this end, it makes four contributions. First, we design and evaluate daisy chain total order protocol (DCTOP), a new ring-based leaderless total order protocol using Lamport's logical clocks for sequencing messages, and a novel idea of "last" process that is the closest to the sending process in the opposite direction to message flow. This results in a unique last process rather than a globally fixed one for each process. Secondly, we evaluated DCTOP using a *greedy* sending approach. This approach ensures processes transmit all their messages first before receiving messages from other processes. We extended this greedy sending approach to include a *fairness* control approach. fairness is defined as every process P_i has an equal chance of having its sent messages eventually delivered by all processes within the system. To ensure fairness, we modified the fairness control algorithm of the Fixed Sequencer and Ring (FSR) order protocol and applied it to our design. Thus, leading us to implement a fairness-controlled DCTOP. Thirdly, we proposed two variations of Raft, which are all capable of reaching an agreement in fewer communication steps than Raft. We modified the traditional Raft system framework into Chain Raft (RaftCh) and Balanced Fork Raft (RaftBf) variations using a novel idea that restricts a leader to commit a client message sequence number when it receives a single acknowledgement from the last follower from either the chain or fork. The outcome is that the last follower on the chain or fork sending a single acknowledgement for sequence number commitment at the leader process is unique, unlike the quorum of acknowledgements required for the same task in the conventional Raft. The RaftBf offers excellent performance but is only appropriate for cluster environments with five processes or more (ideally, odd numbers of processes). The alternative, RaftCh, on the other hand, is created without this limitation. Finally, a detailed performance evaluation of DCTOP and LCR in a non-fairness-controlled (greedy sending) and a fairness-controlled cluster environment is presented. Additionally, Raft and Raft-variant protocols are evaluated, including the evaluation of Raft-variants and DCTOP. Our experiments show that our new approaches offer significant improvements over the existing state-of-the-art methods.

Glossary

- FIFO** First-In-First-Out is a method for organizing and manipulating a network channel where messages sent from a primary replica to a backup are received and read in the order in which they are sent.
- N** Group of processes $N, N \geq 3$ that are fail-independent and fully connected.
- n** The number of followers, where $n = N - 1$.
- Π** Set of DCTOP, LCR, Raft, and Raft-variant processes, one process in each server.
- m** Message in DCTOP, LCR, Raft and Raft-variants clients can send to DCTOP, LCR, Raft and Raft-variants servers, and it can be either type write or read operations.
- LCR** Logical Clock and A Ring, a ring-based total order protocol.
- VC_i** The vector clock of any process P_i in LCR
- P_i** LCR, or DCTOP process
- P_0** the first process in LCR or DCTOP ring
- P_{N-1}** the last process in the LCR or DCTOP ring
- TO delivery(m)** Total order delivery of message m
- m_dst** Message destination
- aNoMD** Average number of messages delivered by any process P_i during the simulation time
- |t.p|** The absolute value of the number of the target process
- mcast** Message Multicast
- rmcast** Reliable Message Multicast
- amcast** Atomic Message Multicast
- abcast(m)** atomic broadcast of message m
- FD** Failure Detector
- GM** Group Membership
- acks(m)** The acknowledgements sent by nodes in LCR, Raft, and Raft-variants protocols
- Seq(m)** Sequence number assigned to message m, which enforces its FIFO ordering.
- FSR** Fixed Sequencer and A Ring protocol.

m(i, r) Message m with the identity of the sending process as i , and r , which indicates the number of rounds m has made during transmission of m as used in LCR.

N – 1 the maximum number of hops a message m makes before a round is completed in LCR and DCTOP, while N is the ensemble size.

UTO Uniform Total Order

utoDeliver Uniform Total Order Deliver

utoMulticasts Uniform Total Order Multicasts

SUC_i Successor of any process P_i

PRED_i Predecessor of any process P_i

GCQ_i Garbage Collection Queue for any process P_i used for storing delivered messages

TCP Transmission Control Protocol.

RPC Remote Procedure Call

MTBF Mean Time Before Failure

DES Discrete Event Simulation

DCTOP Daisy Chain Total Order Protocol

SC_i Stability Clock for any process P_i in DCTOP

LC_i Lamport Logical Clock for any process P_i in DCTOP

ACN_i Anti-clockwise Neighbour of any process P_i in DCTOP

TS timestamp

VT Vector timestamp

[] Mathematical notations, ceiling brackets, which round numbers to upper integer

[] Mathematical notations, ceiling brackets, which round numbers to lower integer

m_{ts} Timestamp associated with a message.

m_{vt} Vector timestamp associated with a message.

CN_i Clockwise Neighbour of any process P_i in DCTOP

mBuffer_i Message m Buffer for any process P_i in DCTOP

DQ_i Delivery Queue for any process P_i in DCTOP

GCQ_i Garbage collection queue for any process P_i in DCTOP

μ(m) Process acknowledgement by the last process of the message sender in DCTOP

DC-mcast – Daisy Chain multicast

Hops_{i,j} The number of hops a message m makes moving from process P_i to process P_j .

DC_i Delivery Counter for any process P_i in DCTOP which indicates the timestamp upto which it has locally completed TO delivery.

GCT_i Garbage Collection Target timestamp for any process P_i in DCTOP

TI Target Increment, which is a fixed value, used to increment GCT.

Δ .GCT The current target value

Δ Message Delta Message used to exchange information about local TO delivery among Processes.

Δ .Setter The identity of the process that sets Δ .GCT

G represents the group of DCTOP processes executing the protocol

Gprev The old group version that immediately precedes G

Survivor (G) The set of processes of Gprev that remain as members of G.

Joiners(G) The processes that joined G after recovering from crashes.

Last_E The earliest last messages delivered by processes of Survivor(G) when the latter is arranged in total order

Last_L The latest last messages delivered by processes of Survivor(G) when the latter is arranged in total order

P_s Survivor process (that is not P_i) is denoted as P_s

λ Client message arrival rate

μ Client message transmission delay

π Client message processing delay

f degree of crash tolerance. for a DCTOP, it is expressed as $f = \left\lfloor \frac{N-1}{2} \right\rfloor$ same as in Raft, where

N is the number of processes within the cluster.

F Failure Pattern

D(F) Failure detector histories

RaftCh Chain Raft

RaftBf Balanced Fork Raft

SMT Simulation Time

Table of Contents

Glossary	vi
Table of Contents	ix
List of Figures	xiii
List of Tables	xvi
1 Introduction	1
1.1 Problem Statement	11
1.2 Our Approach	13
1.3 Thesis Contributions	16
1.4 Thesis Structure	17
2 Background and Related Works	18
2.1 Distributed System Replications	18
2.2 Replication Schemes	20
2.2.1 Active Replication	20
2.2.2 Passive Replication	22
2.3 Atomic Broadcast and Multicast Protocols	26
2.3.1 Broadcast	26
2.3.2 Multicast	27
2.3.3 Reliable Multicast	27
2.3.4 Atomic Multicast	28
2.4 Mechanisms For Message Ordering	29
2.4.1 Fixed-Sequencer Protocol	29
2.4.2 Moving Sequencer Protocol	31
2.4.3 Communication History	32
2.4.4 Destination Agreement	32
2.4.5 Failure Detection	32
2.5 Leaderless Ring-Based Order Protocols	36
2.5.1 Privilege-Based Protocol	37
2.5.2 Fixed Sequencer and A Ring Topology (FSR) Protocol	43
2.6 Leader-Based Order Protocols	47

2.6.1	Paxos	47
2.6.2	Ring Paxos	48
2.6.3	ChainPaxos	49
2.6.4	Chubby	50
2.6.5	Zookeeper	51
2.7	Background	52
2.7.1	LCR Order Protocol	52
2.7.2	The Raft Protocol	68
2.7.3	Differences among Zookeeper, Paxos and Raft Protocols.....	80
2.8	Discrete Event Simulation (DES)	82
2.9	Summary	83
3	Daisy Chain Total Order Protocol - DCTOP.....	85
3.1	Rationale.....	86
3.2	System Model.....	87
3.2.1	Assumptions.....	90
3.2.2	Ring Structure	91
3.3	Protocol Principle.....	92
3.3.1	Message Sending, Receiving and Forwarding.....	93
3.3.2	Significance of Timestamp Stability.....	95
3.3.3	Crashproofing of Messages.....	95
3.3.4	Data Structures.....	96
3.4	Stepwise Design of DCTOP.....	98
3.4.1	Non-Uniform DC_mcast.....	99
3.4.2	Uniform DC_mcast.....	101
3.4.3	Non-Uniform, Total Ordered multicast	104
3.5	DCTOP: Uniform and Total Order Multicast	108
3.5.1	DCTOP Algorithm Main Points	108
3.5.2	DCTOP Pseudocodes.....	111
3.5.3	Exemplifying DCTOP Pseudocode Implementation	113
3.5.4	Garbage Collection	118
3.5.5	DCTOP Membership Changes	124
3.5.6	DCTOP Proof of Correctness	134
3.6	DCTOP Operations	141
3.6.1	Single Multicast	141
3.6.2	Concurrent Multicast	142

3.6.3	Performance Benefits of DCTOP	144
3.7	Fairness-Controlled Environment	144
3.7.1	Fairness Performance Model	145
3.7.2	Fairness Control Scenarios	141
3.8	Summary	147
4	Comparative Performance Evaluation of Leaderless Protocols	151
4.1	Motivation	151
4.2	Greedy Sending Approach (Without Fairness Primitives).....	152
4.2.1	Simulation	153
4.2.2	Evaluation	156
4.2.3	Summary	159
4.3	With Fairness Primitive.....	159
4.3.1	Evaluation	160
4.3.2	Fairness vs Without Fairness Control Primitives.....	163
4.3.3	Summary	165
5	Tackling Leader Load in Raft Protocol	168
5.1	Motivation - Raft Performance Bottleneck	168
5.2	Design Objectives	173
5.3	Assumptions	174
5.4	Design Approach.....	177
5.4.1	Single Explicit Acknowledgement	177
5.4.2	Commit Messages.....	179
5.4.3	Switch to/from Raft.....	179
5.5	Design Difference between Raft and Raft-variants.....	179
5.6	Raft-variants Protocol Details	181
5.6.1	Protocol 1: RaftCh	182
5.6.2	Protocol 2: RaftBf	183
5.7	Summary	185
6	Performance Evaluation of Raft-variants, and DCTOP	186
6.1	Simulation	187
6.2	Evaluation.....	190
6.2.1	Raft and RaftCh	190
6.2.2	Raft, RaftCh and RaftBf	193
6.2.3	RaftCh and DCTOP	196
6.2.4	RaftBf and RaftCh with Zero Processing Delay	199

6.2.5	RaftBf and RaftCh with Non-Zero Processing Delay.....	201
6.2.6	RaftCh (N = 3, and N = 5)	202
6.2.7	RaftCh, RaftBf and DCTOP	204
6.3	Summary	207
7	Conclusion	212
7.1	Thesis Summary	212
7.2	Recommendations	211
7.3	Limitations	212
7.4	Future Research.....	214
7.4.1	Heterogenous Node Setting	214
7.4.2	Additional Implementation and Experiments	214
7.4.3	Process Crashes Evaluation	214
7.4.4	Request Batching	215
	References.....	218
	Appendix A1 – Simulation Design	231

List of Figures

Figure 1.1 Conflict in Ticket Allocation Due to Lack of Consistent Order	3
Figure 1.2 Out of Order Processing	4
Figure 1.3 Total Order Processing	5
Figure 1.4 Total Order within Concurrent Broadcast	7
Figure 1.5 LCR Ring Structure	10
Figure 1.6 Five Process Cluster in LCR	15
Figure 1.7 Thesis Structure. Topics are given in boxes with square corners. Algorithmic contributions are illustrated with rounded boxes and shaded in orange; the performance evaluation technique used is shaded in blue.	17
Figure 2.1 Active Replication [52]	21
Figure 2.2 Passive Replication [52]	22
Figure 2.3 Fixed-Sequencer Protocol [18]	29
Figure 2.4 Common Variants of Fixed Sequencer Protocol [18]	30
Figure 2.5 Moving Sequencer Protocol [18]	31
Figure 2.6 FSR Protocol Design [1]	44
Figure 2.7 Incoming buffer and forward list of a process initiating a TO-broadcast [1]	45
Figure 2.8 States of Values [35]	48
Figure 2.9 Chubby's Read/Write in Master/Process Scheme [64]	51
Figure 2.10 LCR System Model [2]	53
Figure 2.11 Performance Model [2]	54
Figure 2.12 LCR Communication Design [2]	55
Figure 2.13 Concurrent unicast ordering in LCR [2]	58
Figure 2.14 Complex Scenario	59
Figure 2.15 Total Order Delivery [2]	61
Figure 2.16 Pseudocode of LCR Protocol [2]	65
Figure 2.17 A Replicated Log [3]	69
Figure 2.18 Server Hosting Processes in a Typical Raft Cluster [3]	70

Figure 2.19 Raft Server States [3].....	71
Figure 2.20 Raft Star Structure [3]	72
Figure 2.21 Raft Leader Election [3]	74
Figure 2.22 Timeline for a configuration change [3].....	77
Figure 2.23 Raft Client Interaction [3]	79
Figure 3.1 Traditional Network System.....	87
Figure 3.2 Logical Ring in DCTOP Model	88
Figure 3.3 P_0 Sending m_1 , and m_2 to P_1 and P_2	88
Figure 3.4 Time Steps in Traditional Network System	89
Figure 3.5 Time Steps in Ring Structure	89
Figure 3.6 DCTOP Ring Structure.....	91
Figure 3.7 Message Sending, Receiving, Forwarding	93
Figure 3.8 The Relationship between m and $\mu(m)$	97
Figure 3.9 DCTOP High Level Framework	98
Figure 3.10 Daisy Chain Multicast Design.....	100
Figure 3.11 The role of μm in uniform DC_mcast.....	103
Figure 3.12 Non-uniform Total Order Multicast	107
Figure 3.13 Checking the crashproofness of a message	110
Figure 3.14 Illustrating DCTOP Pseudocodes.....	113
Figure 3.15 Concurrent Multicast from P_1 and P_3	114
Figure 3.16 P_0 and P_2 Forwards Concurrent Messages from P_1 and P_3 to CN_0 and CN_2	115
Figure 3.17 P_1 and P_3 Forward Concurrent Messages to CN_1 and CN_3	116
Figure 3.18 P_0 and P_2 generate and send $\mu(m)$ and $\mu(m)$ Concurrently to CN_0 and CN_2	117
Figure 3.19 P_1 and P_3 Forward $\mu(m)$ and $\mu(m)$ Concurrently to CN_1 and CN_3	117
Figure 3.20 P_0 and P_2 Forward $\mu(m)$ and $\mu(m)$ Concurrently to CN_0 and CN_2	118
Figure 3.21 Initiating Circulation of Δ Message	121
Figure 3.22 Second Round of Δ Circulation.....	121
Figure 3.23 DCTOP Membership Changes Overtime - An Example	124
Figure 3.24 Pseudocode of the DCTOP Protocol	131
Figure 3.25 Continuation of the Pseudocode of the DCTOP Protocol.....	132
Figure 3.26 Pseudocode of the DCTOP Membership Changes.....	133
Figure 3.27 Example Contradicting Lemma 1	137
Figure 3.28 P_j deduces stability of m_{ts} by receiving $\mu(m)$	139

Figure 3.29 Incoming Queue is empty.....	142
Figure 3.30 Incoming Queue is not empty.....	141
Figure 3.31 The Sending Queue is empty at some point.	142
Figure 4.1 Latency Comparison without Fairness Primitive	157
Figure 4.2 Throughput Similarity without Fairness Primitive.....	158
Figure 4.3 Latency Comparison with Fairness Primitive	161
Figure 4.4 Throughput Similarities with Fairness Primitive	162
Figure 5.1 Conventional Logical Message Dissemination of Raft	168
Figure 5.2 Raft Response Time Estimation	169
Figure 5.3 Raft Latency vs. Arrival Rate	171
Figure 5.4 Raft Throughput vs. Arrival Rate	171
Figure 5.5 Leader Election Scenarios	175
Figure 5.6 Explicit and Implicit Acknowledgements	178
Figure 5.7 Chain Raft.....	182
Figure 5.8 Balanced Fork Raft.....	184
Figure 6.1 Latency Comparison between RaftCh and Raft	191
Figure 6.2 Throughput Comparison between Raft and RaftCh	191
Figure 6.3 Latency Comparison for Raft, RaftBf, and RaftCh.....	193
Figure 6.4 Throughput Comparison for Raft, RaftBf, and RaftCh.....	195
Figure 6.5 Latency Comparison between RaftCh and DCTOP.....	197
Figure 6.6 Throughput Comparison between RaftCh and DCTOP	197
Figure 6.7 Latency Comparison for RaftBf and RaftCh (Zero Processing Delay).....	199
Figure 6.8 Throughput Comparison for RaftBf and RaftCh (Zero Processing Delay).....	200
Figure 6.9 Latency Comparison for RaftBf and RaftCh (Non-Zero Processing Delay)	201
Figure 6.10 Throughput Comparison for RaftBf and RaftCh (Non-Zero Processing Delay).....	202
Figure 6.11 Latency Comparison of RaftCh for $N = 3$ and $N = 5$	203
Figure 6.12 Throughput Comparison of RaftCh for $N=3$ and $N=5$	203
Figure 6.13 Latency Comparison for RaftBf, RaftCh, and DCTOP.....	206
Figure 6.14 Throughput Comparison for RaftBf, RaftCh, and DCTOP.....	207

List of Tables

Table 2-1 Fault Type.....	22
Table 2-2 Eight Classes of Failure Detectors [57].....	36
Table 2-3 LCR Operation with Single Multicast.....	66
Table 2-4 LCR Operation with Concurrent Multicast	67
Table 3-1 Single Multicast in DCTOP using $N=3$	141
Table 3-2 DCTOP Operation with Concurrent Unicast.....	142
Table 4-1 Performance improvement of DCTOP over LCR in the absence of fairness control primitive.....	163
Table 4-2 Performance improvement of DCTOP over LCR in the presence of fairness control primitive.....	163
Table 5-1 Difference Between Raft and Raft variants.....	181
Table 6-1 Simulation Differences Between Raft, Raft-variants and DCTOP	188
Table 6-2 Parameters of the Simulation.....	188
Table 6-3 Performance Comparison for RaftCh, $N = 3$	192
Table 6-4 Latency Performance Improvements of Raft-variants	195
Table 6-5 Throughput Performance Improvements of Raft-variants	196
Table 6-6 Performance Improvement of DCTOP (throughput) and RaftCh (latency) against each other.....	198
Table 6-7 Performance Improvement of RaftBf over RaftCh for Zero Processing Delay	200
Table 6-8 Performance Improvement of RaftBf over RaftCh for Non-Zero Processing Delay	202
Table 6-9 Performance Improvement of RaftCh when $N = 3$ over when $N = 5$	204
Table 6-10 Latency Performance Improvement of RaftBf and RaftCh over DCTOP	207
Table 6-11 Throughput Performance Improvement of DCTOP over RaftBf and RaftCh ...	207

Chapter 1

Introduction

Distributed data applications execute and manage data processing across multiple interconnected processes that must communicate to complete task collaboratively [9-12]. Examples include distributed file systems, distributed databases, and data processing frameworks used in distributed computing. A distributed data application is typically organized around *clients* and *services*. A service consists of one or more *process replicas* that execute *operations* accessible to clients through *requests* [4]. The simplest implementation of a service involves a single centralized process, that handles all client requests. However, this single centralized configuration lacks fault tolerance because if the single process fails, the entire service becomes unavailable.

Moreover, distributed systems allow the use of replicated services. A replicated service refers to a system where multiple copies (replicas) of a service are maintained across different processes. These replicas ensure that the service remains available even if some replicas fail. For example, in a replicated database, a copy of the data is maintained at multiple processes, ensuring continued operation during failures. Thus, efficient handling of requests, combined with robust mechanisms for detecting and recovering from failures, is essential for maintaining high performance and reliability. The need for such capabilities has driven the development of crash-tolerant distributed systems [21-24]. A crash-tolerant distributed system is designed to continue functioning even when individual components fail. Failures in distributed systems can arise from hardware malfunctions, software bugs, or network disruptions [25-28]. A crash failure is when a process fails and stops communication and computation with other processes. In a fail-stop failure, a process halts communication and computation without producing erroneous outputs, ensuring that it does not interfere with the operation of other correct process replicas [29-32]. Fail-stop failures simplify fault detection because a failed process becomes silent and does not generate conflicting data. Hence, incorporating fail-stop methodologies into distributed systems enhances fault tolerance by ensuring that the system remains stable and

functional, even in the presence of component failure. There are several approaches to achieving a replicated service, one of which is passive replication, also known as primary-backup model. In this technique, a single process, designated as the primary, executes all clients requests and synchronizes its state with backup processes. In the event of a primary failure, one of the backups is promoted to become the new primary, ensuring continuity of service. However, in this thesis, our focus is on state machine replication (SMR) [4-8], where all processes execute the same sequence of operations, ensuring consistency across the processes. Furthermore, SMR involves replicating the state of a service across multiple process replicas, ensuring that all replicas execute the same sequence of operations. This guarantees consistency, even in the presence of failures. In SMR, a service is modelled as a deterministic state machine, which starts in an initial state and transitions to new states based on a sequence of input commands. The deterministic nature of the state machine ensures that given the same initial state and input sequence, all replicas reach the same final state. To implement SMR, the system must ensure that even if some replicas fail, the remaining ones remain synchronized and coherent. This synchronization guarantees that the system responds reliably to client requests while maintaining consistency. However, achieving both consistency and availability simultaneously may depend on the system's design and absence of network partitions [33-35]. On the other hand, a replication group or simply a group refers to the set of processes that participate in providing the replicated service, where members communicate by multicasting to all group members [13]. These processes work together to maintain the system's state and respond to client requests. A process can join or leave the group dynamically. If a process crashes, it exits the group and halts communications. Upon recovery, the process joins the group and synchronizes with updates that occurred during its downtime. If a process in the group fails, the remaining processes ensure that the service continues to function. The replication group forms the backbone of state machine replication, by collectively managing request ordering, fault detection, and recovery, ensuring consistency and availability even in the presence of failures. Response time, which is the time elapsed between a client sending a request and receiving the corresponding reply, is a key metric for evaluating the performance of a replicated group [20]. Lower response times are essential for providing a responsive and seamless client experience in a distributed application. However, the fundamental operation of a replicated service is to guarantee that all replicas execute client requests in a consistent order. This ordering can be achieved using two primary approaches: (i) Leader-based ordering, where a single process replica, called the leader, has a responsibility for deciding the order of clients

requests. The leader receives clients requests, orders the requests, and propagates the sequence to other processes. Protocols such as Paxos [38, 39], Zookeeper [40], Chubby [41, 42], and Raft [3, 43] are widely known examples of leader-based ordering. (ii) Leaderless Ordering: In leaderless systems, there is no centralized authority to order requests. Instead, processes collaboratively determine the order using decentralized protocols. Protocols such as Privilege-Based Protocols [44, 45], FSR [1], and LCR [2] are well known examples of leaderless ordering. Thus, our primary focus in this thesis is on leaderless ordering as implemented by the LCR protocol, while the secondary focus is on leader-based ordering, specifically as offered by the Raft protocol.

Nevertheless, achieving total order in a replicated group is critical and challenging, especially in environments with extensive inter-process communication [19] because total order guarantees consistent behaviour across all processes, which is essential for the correct functioning of state machine replication. Significant research has been conducted to address this issue, yielding promising solutions [14-17]. Total order protocols ensure that multiple process replicas within a cluster see things in identical order even though they may receive them differently [18]. Without total order, processes may execute requests in different sequences, leading to inconsistent states and incorrect outputs. For example, Figure 1.1 illustrates a collection of three processes called P_1 , P_2 , and P_3 , depicting a distributed cluster environment: P_1 receives m_1 , P_2 receives m_2 , and P_3 receives m_3 containing request₁, request₂, and request₃ from clients for processing, respectively.

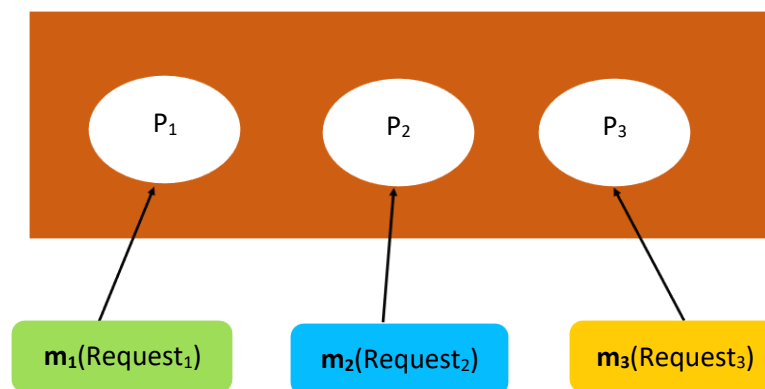


Figure 1.1 Conflict in Ticket Allocation Due to Lack of Consistent Order

Consider a situation where only 1 ticket is available for a London to Lagos flight, but 3 clients are competing for it and the database is replicated on P_1 , P_2 , and P_3 . In this scenario, P_1 agrees to sell the ticket to Client 1, P_2 agrees to sell it to Client 2, and P_3 agrees to sell it to Client 3. Consequently, 3 tickets are sold despite only 1 ticket being available, even when failures are absent, due to the lack of consistent ordering. This situation creates a conflict, underscoring the

critical importance of ensuring a total order in distributed systems to maintain a consistent ordering and prevent such anomalies. Consider another scenario as illustrated in Figure 1.2. Figure 1.2 illustrates a space-time diagram featuring three process replicas denoted as P_1 , P_2 , and P_3 , within a distributed cluster environment. This depiction assumes that all replicas are operational, with no expected crashes. Initially, P_1 , P_2 , and P_3 receive messages m_1 , m_2 , and m_3 containing request₁, request₂, and request₃ from clients 1, 2, and 3 respectively. These processes simply log the messages and then broadcast what they receive from clients to other processes without any agreed-upon order of execution. Consequently, each process replica within the cluster delivers the messages to the high-level application in a different order, as illustrated in Figure 1.2: P_1 delivers m_1 , m_2 , and m_3 , P_2 delivers m_2 , m_3 , and m_1 , and P_3 delivers m_3 , m_1 , and m_2 . This out-of-order processing is harmful to distributed applications. Imagine a scenario where two tickets are available for a flight, and the database is replicated across P_1 , P_2 , and P_3 . Due to inconsistent ordering of client requests, P_1 denies a ticket to Client 3, P_2 denies a ticket to Client 1, P_3 denies a ticket to Client 2. Consequently, no clients receive a ticket, even though two tickets are available. This example underscores the necessity of maintaining total order to ensure consistent outcomes in distributed systems.

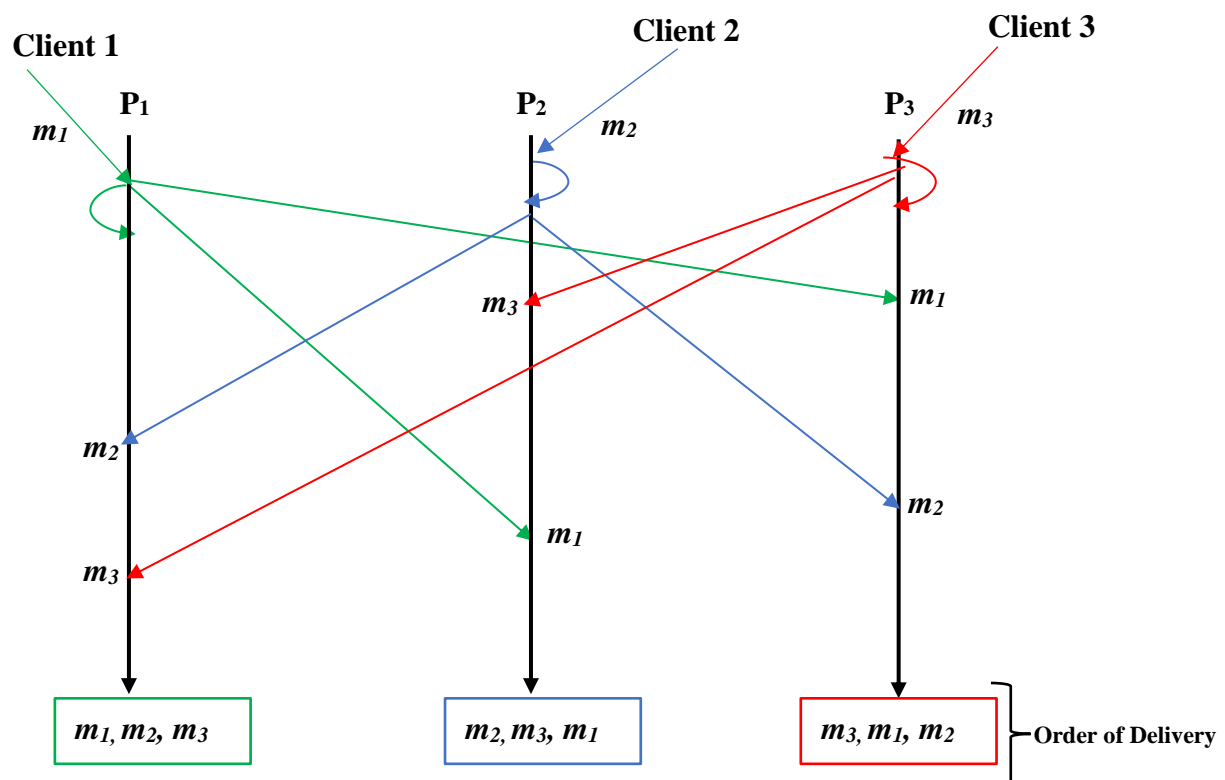


Figure 1.2 Out of Order Processing

Therefore, to ensure consistency, we must enforce a mechanism that compels process replicas to execute client requests in the same order. This approach ensures that all replicas approve tickets for the same two clients while uniformly denying the request for the third client, maintaining uniformity and coherence across the system. Hence, to ensure consistent order processing of the client's request, the replicas must pass the client request executions through a total order protocol. A total order protocol as earlier explained serves as a mechanism used within distributed systems to achieve agreement on the order in which messages are delivered to all process replicas in the group. This protocol ensures that all replicas receive and process the messages in the same order, irrespective of the order in which they were initially received by individual replicas. Establishing total order is critical for maintaining consistency, as it guarantees that all process replicas share a uniform and synchronized view of the system's state. As shown in Figure 1.3, when processes P_1 , P_2 , and P_3 receive messages from clients 1, 2, and 3, instead of broadcasting as they received as noted in Figure 1.2, they broadcast the messages through an order protocol within the group.

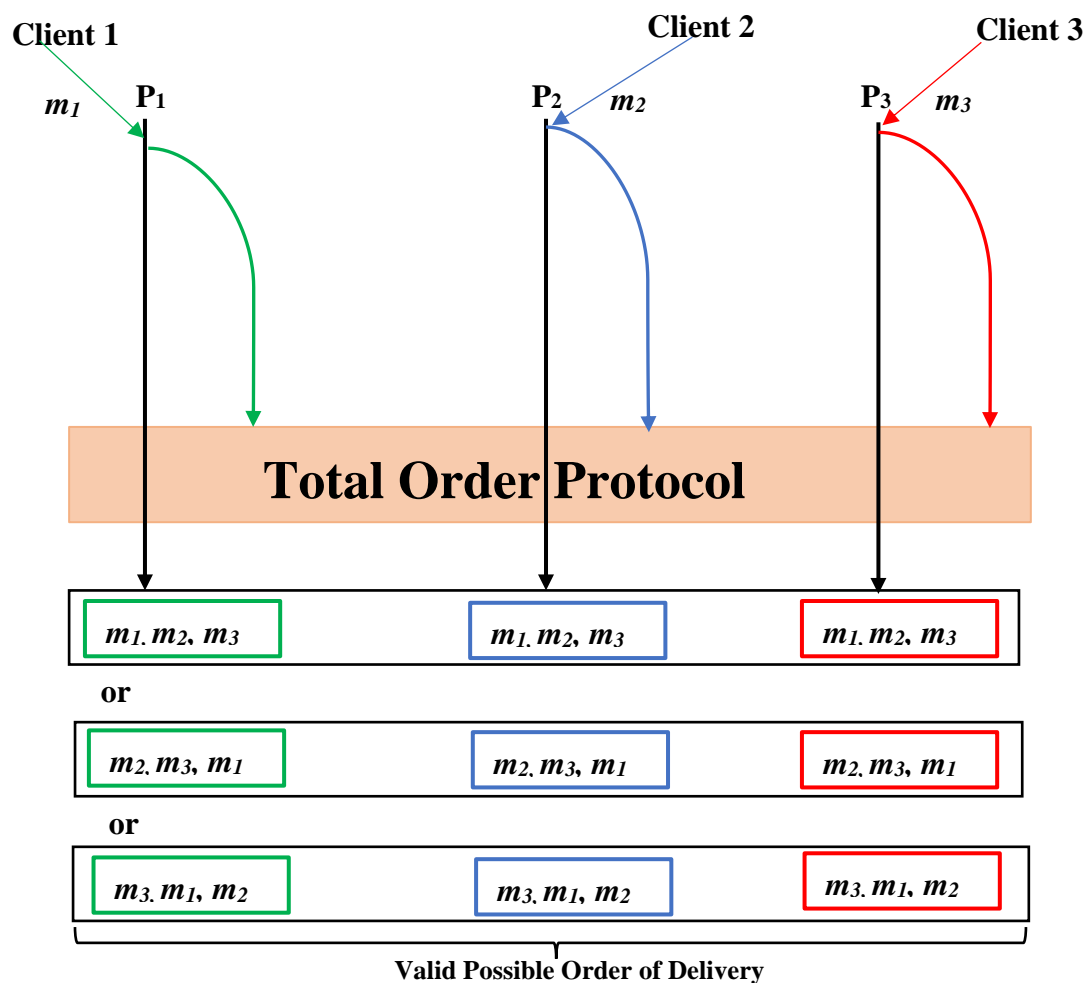


Figure 1.3 Total Order Processing

The use of an order protocol ensures that regardless of the differing client request reception times by P_1 , P_2 , and P_3 , these process replicas must execute and deliver these messages in the same order, and one such order could be: m_1, m_2, m_3 ; m_2, m_3, m_1 or m_3, m_1, m_2 as represented in Figure 1.3. There are various methods for achieving total order in distributed systems; however, this work focuses on utilizing a *logical clock* as the primary mechanism to enforce total order. A logical clock is a mechanism used in distributed systems to assign timestamps to events, enabling processes to establish a consistent order of events occurring. It operates as follows: Each process in the system maintains a logical clock (a counter) initialized to zero. When a process performs a local action (e.g., a send event—sending a message), it increments its logical clock by 1. When a process sends a message, it includes its current logical clock value as a timestamp in the message. When a process receives a message, it compares its own logical clock with the timestamp in the message. It then sets its logical clock to the maximum of its current logical clock and the received timestamp and increments it by 1. In addition, Lamport defined total order using a relation denoted by $' \Rightarrow '$ (total order), as follows: If ' m_1 ' represents an event in process P_1 and ' m_2 ' signifies an event in P_2 . Then $m_1 \Rightarrow m_2$ if and only if either: (i) $\text{Timestamp}_1(m_1) < \text{Timestamp}_2(m_2)$ or (ii) $\text{Timestamp}_1(m_1) = \text{Timestamp}_2(m_2)$ and $P_1 < P_2$ [7]. Consider a scenario where P_2 sends its own message m_2 to P_1 and P_3 after receiving m_1 . However, due to network delay, P_3 sends its own message m_3 before receiving m_2 , as shown in Figure 1.4. It is important to note that the messages sent by P_2 and P_3 are independent of each other, representing concurrent message-sending events. This results in different receive orders for different process replicas in the cluster: $P_1(m_1, m_2, m_3)$, $P_2(m_1, m_2, m_3)$, and $P_3(m_1, m_3, m_2)$. To address this challenge, total order protocol that uses logical clocks is employed to order events. Hence, each process represented in Figure 1.4 timestamps its messages with the current logical clock value at the time of sending. The timestamp associates the message with the specific send or receive event. The protocol then compares these timestamps to establish a consistent total order across all processes, using tie-breaking rules when timestamps are equal. This ensures a reliable and consistently agreed sequence of message order across the processes. According to Figure 1.4, $m_1 \rightarrow m_2$ and $m_1 \rightarrow m_3$ at all P_1 , P_2 , and P_3 . Since m_2 and m_3 are concurrent events, total order must respects happened before relation (\rightarrow) and identical ordering for concurrent events. Hence, the following three possibilities arise when ordering concurrent events, m_2 and m_3 :

1. If $\text{Timestamp of } m_3 (Tm_3) > \text{Timestamp of } m_2 (Tm_2)$: m_2 is ordered before m_3 ($m_2 \rightarrow m_3$)
2. If $\text{Timestamp of } m_3 (Tm_3) < \text{Timestamp of } m_2 (Tm_2)$: m_3 is ordered before m_2 ($m_3 \rightarrow m_2$)

3. If $Tm_3 = Tm_2$: A tie-breaking rule is applied. The message originating from the process with the higher identifier is ordered before the message from the process with the lower identifier. Therefore, m_3 is ordered before m_2 .

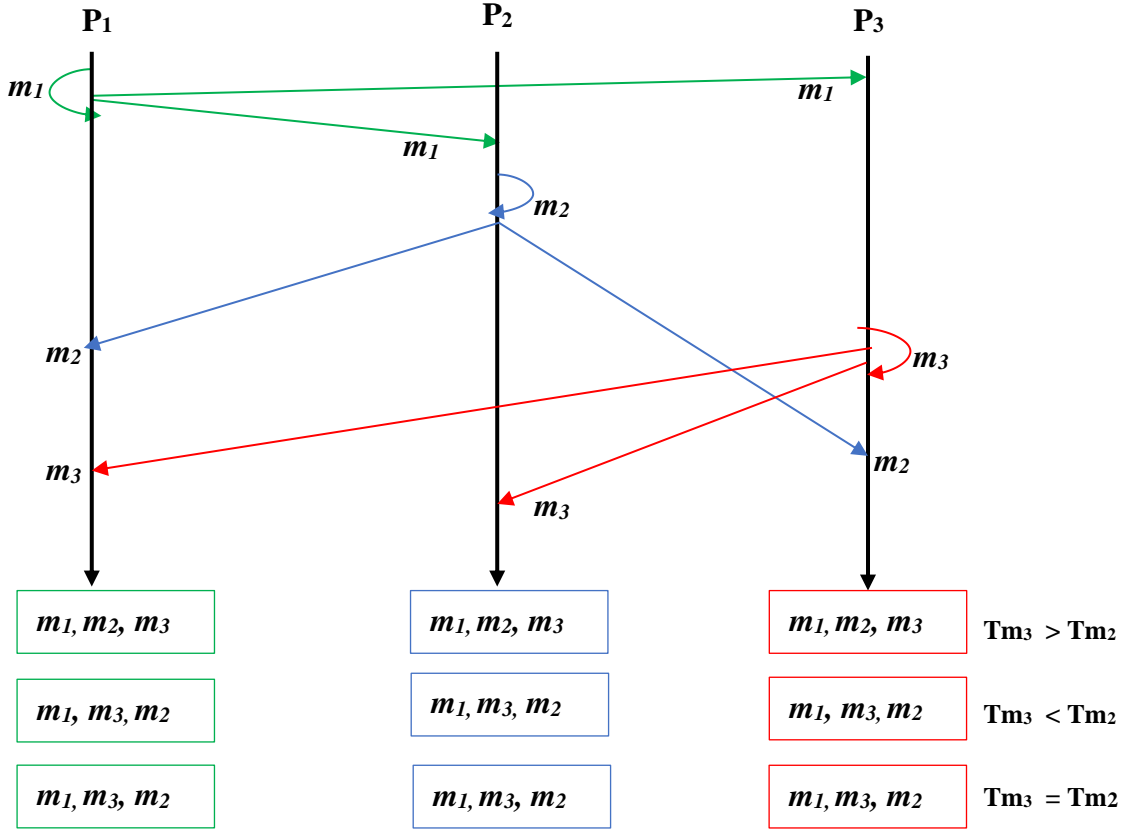


Figure 1.4 Total Order within Concurrent Broadcast

This approach ensures that the consistent state among all process replicas is preserved, maintaining the integrity of the message order despite the network delay-induced disruption [18, 36-37]. Moreover, these examples and many others demonstrate that total order protocols are essential for the reliable operation of distributed applications.

However, the first challenge is how long processes wait to see if another message with $timestamp = Tm_2$ will ever arrive. This uncertainty introduces delays in decision-making and increases the risk of inconsistent message ordering. Consider a scenario in Figure 1.4, where P_3 sends no message m_3 because it receives no client requests. In this case, P_1 and P_2 must decide how long to wait to ensure no message m_3 with $Tm_3 = Tm_2$ from P_3 will ever arrive. Moreover, if P_1 and P_2 wait indefinitely for a potential m_3 with $Tm_3 = Tm_2$, total order latency increases, which in turn elevates the overall system latency. Conversely, deciding prematurely without waiting for m_3 risks violating total order if m_3 eventually arrives. This challenge highlights that a logical clock, while useful for assigning timestamps for event ordering, is not

sufficient for total order because (i) logical clocks provide no guarantee that all processes have acknowledged or received every potential message. (ii) additional mechanisms are required to ensure that processes can determine when it is safe to proceed without waiting indefinitely. Furthermore, the second challenge is that a process can crash. A crash refers to the abrupt failure of a process replica, preventing it from completing its operations. When a process crashes, it ceases communication with other replicas, potentially becoming unreachable within the group. When a process crashes, maintaining total ordering becomes disrupted. The challenge lies in achieving a collective decision in the absence of a crashed process. This disruption can lead to delays in achieving total order because the system may lose the required majority of processes needed to reach agreement within the group. Since at least two processes are necessary to maintain total order, the absence of one or more processes can prevent the system from establishing agreement efficiently. For instance, suppose as illustrated in Figure 1.4, if P_2 and P_3 crash after sending m_2 and m_3 respectively. This situation becomes problematic as P_1 alone cannot determine the total order of the messages, as achieving total order requires agreement with at least one other process. However, if only P_3 crashes after sending m_3 , then P_1 and P_2 after receiving m_3 can still deliver the messages in same order since they form a majority of 2 in the 3-process cluster. While total order protocols play critical role in maintaining consistency and system reliability, achieving crash tolerance requires implementation of additional mechanisms. One such mechanism, as defined in our work, is the crashproofness policy. Given our assumption of $N=2f+1$, as outlined in Section 1.2, the crashproof policy establishes a critical mechanism for ensuring system reliability in the presence of failures. Specifically, this policy dictates that a message is deemed crashproof and safe for delivery once it has successfully reached at least $f+1$ operational processes, where f is the maximum number of processes that can crash in an N group size. The $f+1$ threshold is significant because it represents a strict majority of the system's processes. This majority is essential for preserving consistency and reliability, as it ensures that even if up to f processes crash, there is still one operational process left to communicate the response it to other processes. Therefore, by integrating the total order protocol with a crashproof policy, the system is designed to maintain both functionality and reliability, even in the presence of process failures. This combination of the total order protocol and crashproof policy provides robustness by guaranteeing that message ordering and delivery are unaffected by crashes. Thus, managing crashes effectively is fundamental to designing a reliable and crash-tolerant distributed system. Crashes can interfere with communication, cause delays in reaching agreements, and result in

inconsistencies, posing significant challenges to the system's overall functionality and reliability. Therefore, robust failure management mechanisms are critical to ensuring that distributed systems can detect failures, replicate system state, and recover from faults without compromising the system's functionality and reliability. For example, if a distributed database uses SMR, a failure in one replica does not prevent the remaining replicas from serving client requests if they maintain a consistent state. Failure detection is another critical aspect of crash-tolerant systems. Efficient failure detectors can quickly identify unresponsive or faulty processes, enabling the system to isolate these processes and prevent further disruptions. Fault recovery mechanisms, such as checkpointing and log-based recovery, allow crashed processes to rejoin the system after they have been restored. These mechanisms ensure that a recovering process can synchronize its state with the rest of the group members, minimizing the impact of downtime and maintaining the overall integrity of the application. This resilience is critical for maintaining the reliability and consistency of distributed applications, especially in scenarios that demand high availability, such as financial systems, cloud services, and real-time communication platforms.

Meanwhile, a replicated service latency is the total latency it takes for a distributed replicated service to process and respond to a client request. Replicated service latency can be mathematically expressed as:

$$\text{Replicated Service Latency} = \text{Service latency} + \text{Total order latency}.$$

Service latency is the time taken by a single replica to process a request locally, and total order latency is the time required for all process replicas to agree on the sequence of operations to maintain consistency across the system. In asynchronous systems, measuring latency is critical because it helps understand how delays in communication, coordination, and processing affect the overall performance of the system. Even though processing and communication delays can be arbitrarily large, it is still worthwhile to measure latencies when interested in comparing protocol performances. While service latency is determined by the replica's internal processing efficiency, total order latency depends on the communication overhead and synchronization required by the total order protocol. In this thesis, our focus is on minimizing total order latency, as it is often the more significant contributor to total latency. Reducing total order latency can significantly enhance the responsiveness and scalability of replicated services, ensuring high performance without compromising consistency.

LCR is a ring-based leaderless total order protocol proven to offer a maximum attainable throughput implemented using a set $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ of N , $N \geq 3$ processes as shown in Figure 1.5. P_0 is called the “first process in the ring”, while P_{N-1} is the “last process in the ring”.

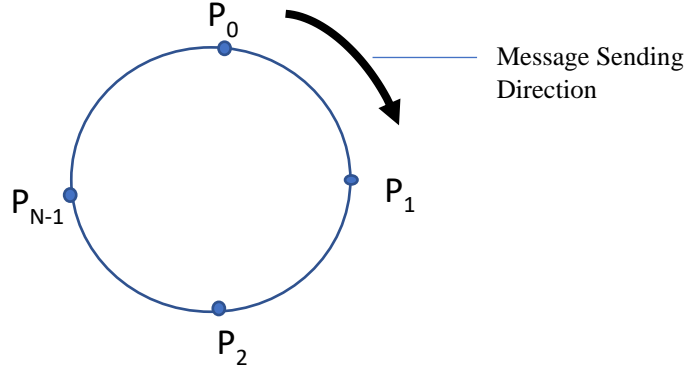


Figure 1.5 LCR Ring Structure

In LCR, messages are sent only in one direction e.g. in clockwise direction as shown in Figure 1.5. So, P_0 sends messages only to P_1 and receives from P_{N-1} . LCR uses a *vector clock*, VC and a *fixed last process* to order concurrent messages irrespective of the process that sends the message. In this system, each process maintains a vector clock, which is an array of integers, where each entry corresponds to a distinct process replica in the ring. When a process desires to send a message, it first increments its index in its vector clock, takes a copy of its vector clock, and assigns the copy as the vector timestamp VT , for the message before sending the message. Let the vector timestamp of a message m be denoted as m_vt , when a process P_j receives a message m , it sets $VC_j[i] = \max \{VC_j[i], m_vt[i]\}$ for every $i, 0 \leq i \leq N - 1$. Let's assume that the message m received by P_j has $m_vt = [5, 0, 4, 4]$, and the current vector clock of P_j is $VC_j = [3, 1, 4, 2]$. Then, after receiving m , the new vector clock of P_j is $VC_j = [5, 1, 4, 4]$. This demonstrates how vector clocks accurately capture causal relationships between events by ensuring that each process maintains the most recent logical times observed. Thus, in a distributed cluster with $N = 4$ processes, the vector clock of process P_1 is initialized to $VC_1 = [0, 0, 0, 0]$. In this vector clock: the first integer at index 0 corresponds to P_0 , the second integer at index 1 corresponds to P_1 , the third integer at index 2 corresponds to P_2 , and the fourth integer at index 3 corresponds to P_3 which is the last process replica in the cluster. So, when the LCR execution for m terminates, both the sender and every process within the cluster environment deliver m , and this delivery event is referred to as total order delivery of m , $TO\ delivery(m)$ for short.

On the other hand, Raft is a leader-based total order protocol created by [3] primarily for supervising a replicated log. Raft was designed to be simple to comprehend, use, and maintain.

Raft protocol seeks to guarantee that a collection of process replicas in a distributed system agree on the order to execute received input requests. This is accomplished by electing a leader from among the process replicas, and only the leader determines that an input request has been correctly replicated by receiving quorum feedback from other processes called followers. This implies that enough numbers have the exact copy of the log as the leader in case the leader crashes. In Raft's cluster, initially, each process starts as a follower. A follower changes to a candidate and starts the election process if it does not hear from the leader for a predetermined amount of time, called election time out. The candidate then votes for itself and requests votes from followers in the cluster. If it wins the majority of the votes, it becomes the new leader and starts supervising new entries to the log. The easy understandability of Raft is one of its primary characteristics, unlike other leader-based order protocols like Paxos, which is quite challenging to understand.

In the Raft protocol, clients send requests to the leader, which then replicates the request as a message to the followers within the cluster. Each follower then sends an acknowledgement message to the leader once it has successfully received the leader's message. Once the leader receives a quorum of acknowledgements from the followers, it commits the sequence number attached to the message to its log, and announces the commit to all followers, the leader and followers execute the message in the committed order. When the leader crashes, a new leader is elected through a leader election process. The new leader begins to append new log entries, and in this way, the system is guaranteed to operate despite some process crashes.

1.1 Problem Statement

The research problems in this thesis are structured as (i) potential drawbacks in leaderless LCR, and (ii) potential drawbacks in leader-based RAFT.

(i) Potential Drawbacks in Leaderless LCR

In the LCR protocol, processes are arranged in a logical ring, and the flow of messages is unidirectional as earlier described. However, LCRs' design may lead to performance problems, particularly when multiple messages are sent concurrently within the cluster: firstly, it uses a vector timestamp for sequencing messages, and secondly, it uses a fixed idea of "last" process to order concurrent messages. Thus, in the LCR protocol, the use of a vector timestamp takes up more space in a message, increasing its size. Informally, vector timestamps are a snapshot of vector clocks at specific events, allowing distributed systems to track causality and guarantee

consistent operation. Consequently, the globally fixed last process will struggle to rapidly sequence multiple concurrent messages, potentially extending the message-to-delivery average maximum latency. The size of a vector timestamp is directly proportional to the number of process replicas in a distributed cluster. Hence, if there are N processes within a cluster, each vector timestamp will consist of N counters or bits. As the number of processes increases, larger vector timestamps must be transmitted with each message, leading to higher information overhead. Additionally, maintaining these timestamps across all processes requires greater memory resources. These potential drawbacks can become significant in large-scale distributed systems, where both network bandwidth and storage efficiency are critical. Thirdly, in the LCR protocol, the assumption $N=f+1$ implies that $f=N-1$, where f represents the maximum number of failures the system can tolerate. This configuration results in a relatively high f , which can delay the determination of a message as crashproof. Specifically, when $f=N-1$, a message must be received by every process in the cluster before it can be delivered. Under high workloads or in the presence of network delays, this requirement introduces significant delays, increasing message delivery latency and impacting system performance. While the assumption $N=f+1$ is practically valid, it is not necessary for f to be set at a high value. Reducing f can enhance performance by lowering the number of processes required to determine the crashproofness of a message. In this thesis, we propose reducing f as a means to minimize overall message delivery latency, thereby enhancing system efficiency.

(ii) Potential Drawbacks in Leader-based RAFT

There exists a possibility of encountering a bottleneck, particularly in a leader-based order protocol such as Raft, where the leader process bears the responsibility of processing all messages, including acknowledgements and AppendEntries remote procedure call (RPC), for the majority, if not all, of the processes. In the Raft, it is assumed that when a leader process broadcasts a message, it must receive a majority of acknowledgements from the followers before the sequence number attached to such message is committed to the state machine for execution. Then subsequently, the leader process sends a reply to the client. This aspect of the Raft protocol tends to impede the performance of the leader, particularly when frequent client requests are directed to distributed applications, exacerbating the overall performance of the Raft protocol. Therefore, it might be simpler and more beneficial to examine how to tackle the leader load in the Raft protocol to improve the overall system performance while still maintaining the integrity of the Raft protocol using a novel approach.

In general, the research reported in this thesis tackles the challenges posed by the LCR protocol by developing a new leaderless ring-based total order protocol tailored for high workloads, which offers lower latency and higher throughput than LCR while considering greedy sending and fairness-controlled approaches. In addition, the research introduces two significant modifications to the Raft order protocol, aiming to decrease both inbound and outbound message traffic at the leader process, ultimately enhancing overall performance.

1.2 Our Approach

The structure of our approach to addressing the potential drawbacks identified in this thesis is organized into two main parts: (i) the approach for addressing the potential drawbacks in leaderless LCR, and (ii) the approach for addressing the potential drawbacks in leader-based RAFT. We evaluated the effectiveness of our approach by comparing it with state-of-the-art methods. To do so, we measure two key performance metrics: *latency* and *throughput*. Latency is defined as the time elapsed between a process's initial multicast of a message m within the cluster and the point when all members in the destination group have delivered m to their respective high-level application processes. Throughput is defined as the average number of total order messages successfully delivered by any process over a given time interval.

(i) Approach for Addressing Potential Drawbacks in Leaderless LCR

In this thesis, we look at strategies to improve LCR total order protocol performance, especially under heavy workloads, by focusing on a novel idea of the last process for ordering concurrent messages within the cluster environment. As a result, this study focuses on circumstances in which the LCR protocol may be a performance bottleneck. This performance bottleneck can happen for a variety of reasons as discussed in Section 1.1, including (i) LCR employs a vector timestamp and (ii) a notion of a "last" process to order concurrent messages, which remains fixed regardless of which processes transmitted those concurrent messages. To put it another way, we reduced message latency within the LCR protocol, which is a ring-based leaderless total order protocol. To achieve this, we used Lamport's logical clocks for timestamping messages and a different concept of the last process, which is the one closest to a sending process in the opposite direction to message flow. As a result, the last process is unique to each process and not globally fixed. The choice of ring topology in our new leaderless total order protocol is because they are particularly relevant in scenarios requiring equal participation of all nodes, such as distributed databases, token ring networks, and other specific communication

systems. They are also useful in systems needing total order protocols, where the ring can act as a sequencer to ensure all processes agree on the order of operations [46-48]. Consequently, we first consider a set of constrained fault assumptions that form the basis of our solution: each process crashes independently of others, and at most f number of processes involved in a group communication can crash. Hence, the maximum number of processes that can crash in a cluster of N processes is bounded by $f = \left\lfloor \frac{N-1}{2} \right\rfloor$, which implies $N = 2f + 1$. This means that the system can continue to operate correctly as long as the number of crashes does not exceed this bound. This is crucial in ensuring that even in the presence of some crashes, the protocol can still guarantee a consistent total order of events across the distributed system. For example, in an $N=5$ processes, $f = 2$. As a result, at least two correct processes are always operational and connected. The parameter, f , is an essential condition for achieving crash tolerance. We have two delivery policies and one of them is crashproof. According to the crashproof policy, when a process sends a message, then $f+1$ operative processes must have received the message before any process can deliver it. Suppose f processes crash, there will be one operative process left to communicate the response. In practical systems such as Raft [3, 43] and Paxos [35, 39], it is common in the literature to assume that $N = 2f + 1$ processes are required to tolerate up to f crash failures which ensures that the majority of the correct processes are always functional. Therefore, we are retaining that assumption in our system. The potential benefit of this assumption is that crashproofness is made quickly for a given message which could in turn reduce the latency. This could be one of the reasons our system might do better than LCR. However, DCTOP is generally designed (see Section 3.2) and will also work if $f = N-1$ as assumed in the LCR with minor changes. The only consequence will be that achieving crashproofness will take longer, affecting delivery times and leading to higher latencies. Secondly, we let each sender's last process determine the messages' stability within a *greedy* sending approach. A greedy sending approach ensures that a process sends all its messages before forwarding messages from other processes. It then communicates this stability by sending an acknowledgement message to other processes. As shown in Figure 1.6, if P_0 sends m to P_1 , the last process of P_0 , which is the message originator, is P_4 . Thus, a message, m , is stable when the last process (P_4) of the message originator has received the message. In addition, we introduced a new concept of "deliverability requirements" to guarantee the delivery of *only stable messages*, which automatically implies crashproof, in total order. We defined a hop as the number of jumps a message m makes from P_i to P_j and is denoted as $Hops_{ij}$. Then, for a cluster of $N = 5$ process replicas, the maximum number of crashes tolerated is 2, that is $f = 2$.

A message is crashproof if the number of message hops, $Hops_{ij} \geq f + 1$; that is, a message must make at least $f + 1$ hops before it is termed crashproof. For example, using Figure 1.6 as a reference, when P_0 sends a message m to P_1 , if the message is received by P_1 , it has made 1 hop; if P_1 forwards the message to P_2 and P_2 receives it, the message has made 2 hops.

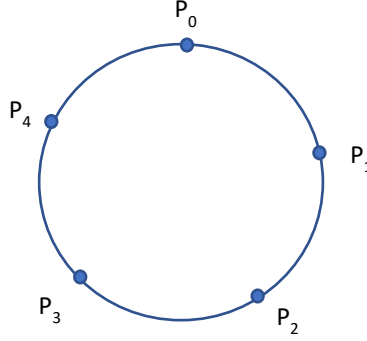


Figure 1.6 Five Process Cluster in LCR

Also, when P_3 receives m , then m has made 3 hops. At this point, the message is crashproof because $Hops_{ij} \geq f + 1$ is true since $f = 2$. Then when P_4 receives the message, the message is stable and also the message has made 4 hops satisfying $Hops_{ij} \geq f + 1$. This illustration shows that stable messages at P_4 from P_0 automatically imply crashproof. Thus, the delivery of a message is subject to meeting both deliverability and order requirements. Thirdly, we defined fairness as every process P_i has an equal chance of having its sent messages eventually delivered by all processes within the group; $0 \leq i < N - 1$, where N is the group size. Every process ensures messages are forwarded in the order they were received before sending their own message. We proposed incorporating a fairness control primitive into our solution and that of the LCR protocol to prevent any process from having message-sending priority over another.

(ii) Approach for Addressing Potential Drawbacks in Leader-based RAFT

We present Raft protocol variations, which, unlike the Raft protocol, our approach involves building two new Raft variations: (a) Chain Raft (RaftCh) and (b) Balanced Fork Raft (RaftBf). In the RaftCh, the leader L , communicates with followers (F_1, F_2, \dots, F_{N-1}) using unicast communication, similar to Raft. However, unlike Raft, only the last follower in the chain (F_4 in an $N=5$ cluster, see Section 5.6.1) sends an acknowledgment to the leader. Once the leader receives this acknowledgment, it commits the sequence number associated with the message and notifies all followers of the commit. The leader and followers then execute the message in the committed order. RaftCh is expected to achieve outstanding latency when $N=3$. However,

as $N > 3$, latency may degrade because the message must traverse a longer chain before the leader receives the acknowledgment from the last follower (F_4).

The RaftBf does not have the limitations of ChainRaft and can be used for any $N > 3$ (preferably odd numbers). In RaftBf, the leader (L) manages two balanced forks of followers. For an $N=5$ cluster, followers are divided as F_1, F_2 on one fork and F_3, F_4 on the other (see Section 5.6.2). The leader (L) receives a single acknowledgment from the last follower on either fork (either F_2 or F_4) before committing the sequence number assigned to the client message. After committing, the leader announces the commit to all followers, and both the leader and followers execute the message in the committed order. The Balanced Fork Raft is expected to outperform ChainRaft across all group sizes of N . Furthermore, we specifically designed the Raft variations discussed above to operate under the fault-tolerant assumptions made in the traditional Raft. As a result, they provide a viable substitute for the original Raft order protocol. It is also crucial to note that the Raft variants we suggest in this study differ from Raft only in the normal (fail-free) section of Raft. They are demonstrated to retain all invariants required to use the crash-recovery part of Raft unmodified.

1.3 Thesis Contributions

The research presented in this thesis makes the following significant contributions:

- (i) Design and evaluation of daisy chain total order protocol (DCTOP), a new ring-based leaderless total order protocol for enhancing latency and throughput.
- (ii) A new system model, fairness controlled DCTOP, which offers fairness in message sending among process replicas within the DCTOP cluster. We modified the fairness control algorithm of the Fixed Sequencer and Ring (FSR) protocol and applied it to our design. We build this fairness control into the LCR protocol for performance comparison.
- (iii) Design and evaluation of two Raft variants, RaftCh and RaftBf, for tackling the leader load in Raft thereby improving the latency and throughput of the Raft protocol.
- (iv) An extensive performance evaluation of DCTOP and LCR in a greedy sending environment (non-fairness-controlled) and fairness-controlled environment. The evaluation includes Raft, RaftCh, and RaftBf protocols and encompasses the evaluation of Raft-variants with DCTOP.

1.4 Thesis Structure

This section outlines the structure of the thesis, providing a concise summary of each chapter. The thesis structure is summarized in Figure 1.7.

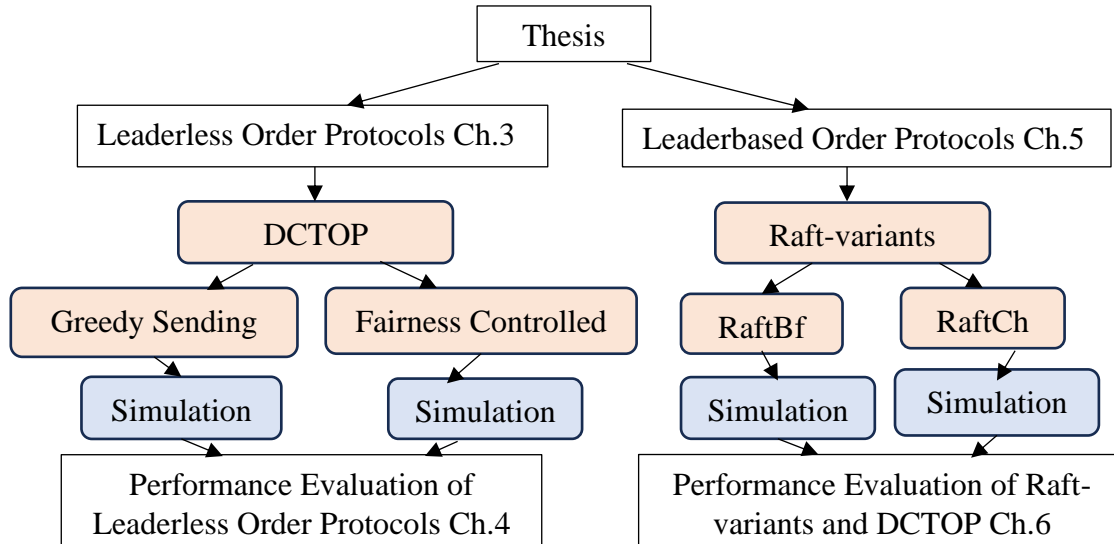


Figure 1.7 Thesis Structure. Topics are given in boxes with square corners. Algorithmic contributions are illustrated with rounded boxes and shaded in orange; the performance evaluation technique used is shaded in blue.

Chapter 2- Background

Provides critical essential information required to understand the problem domain.

Chapter 3- Daisy Chain Total Order Protocol-DCTOP

Presents the rationale, design assumptions, and essential implementation details for the DCTOP primitives, as well as detailing Fairness control primitive for DCTOP.

Chapter 4- Comparative Performance Evaluation of Leaderless Protocols

Provides a thorough performance evaluation of the DCTOP and the LCR protocols, comparing both using without fairness control and with fairness control primitive.

Chapter 5- Tackling Leader Load

Presents the rationale, design assumptions, and essential implementation details for the Raft variants, RaftCh and RaftBf, to enhance the performance of the Raft protocol.

Chapter 6- Performance Evaluation of Raft-variants and DCTOP

Provides a thorough performance evaluation of the Raft, RaftCh and RaftBf protocols, including comparing Raft-variants and the DCTOP.

Chapter 7- Conclusions

Provides a summary of the findings presented throughout this thesis and proposes potential future research based on our results.

Chapter 2

Background and Related Works

Addressing data inconsistencies arising from process-to-process communication within a cluster of environments often involves the implementation of an *order primitive*. An order primitive in a distributed system is a technique that establishes a specified order for events occurring at distinct processes within a distributed cluster. Order primitives are critical for preserving consistency and reliability in distributed systems, particularly when numerous processes are involved, and events can occur simultaneously. They serve an important role in ensuring that the system operates reliably and that processes can agree on the chronological order of occurrences, which is necessary for correct and reliable operation. This chapter offers comprehensive background information on prevalent order protocols, pivotal to our research study. It delves into related works on total order protocols, presenting various techniques for achieving high-performance distributed computing. Additionally, the chapter provides a concise overview of the discrete event simulation architecture, utilized in tandem with Java to simulate our proposed solutions.

2.1 Distributed System Replications

Distributed systems are built as a collection of services in commodity computers implemented by server processes or simply processes and called upon by client processes. Consequently, these systems are susceptible to process failures, presenting challenges in various distributed system domains, including but not limited to e-commerce, finance, telecommunication, booking reservation, and industrial control systems. This becomes notably crucial in scenarios where identifying and addressing process failures can consume valuable time. For instance, in critical systems like air traffic control, prompt detection and handling of process failures are paramount. Delays in recognizing a failed process can impede the system's ability to manage air traffic effectively, potentially leading to disruptions and safety concerns. Similarly, in financial trading platforms, rapid identification and resolution of process failures are essential

to prevent interruptions in trading operations, ensuring the timely execution of transactions, and minimizing financial risks.

In the contemporary technological landscape, users demand seamless experiences without any tolerance for downtime. Consequently, leading corporations including but not limited to Google, Amazon, and Microsoft, allocate billions of dollars annually to maintain the reliability of their cloud infrastructures. They acknowledge the crucial importance of ensuring uninterrupted service to meet the demands of their users [49]. To withstand failures and sustain functionality, developers of distributed systems have integrated replication schemes and related technologies. This integration aims to harness the advantageous effects of fault tolerance in distributed computing. Consequently, the design and implementation of fault-tolerant distributed systems have become integral components in the comprehensive strategy to address total system failures within a distributed system framework. Furthermore, a replication mechanism is defined as an act of making a redundant process and guaranteeing that it has the same attributes as other existing processes. Replication methods exist primarily to ensure fault tolerance, as developing a system that is completely immune to future process failures is a practically impossible task. Fault tolerance encompasses the system's ability to perform its intended function even in the presence of process failures [50]. Replication comes with several benefits [33, 51]:

- (i) Replication enhances the reliability of data applications. By distributing data across multiple processes, the application ensures that its data remains accessible even in the event of a hardware failure on any process within the replication group.
- (ii) Replication improves data read performance. Replication helps to distribute data reads over different processes in a distributed network. This allows system developers to increase the read performance of data applications significantly.
- (iii) Replication enhances data durability guarantee. Therefore, by employing multiple processes to ensure data writes in a distributed system, replication enables one to minimise the issue of a single point of failure.
- (iv) Replication of processes allows the system's scalability, which is a more significant addition to building a fault-tolerant distributing system.
- (v) Replication of processes guarantees safety- at no time does it return erroneous results under every non-Byzantine condition.

2.2 Replication Schemes

There are two available replication schemes, active and passive [52]. In this section, we discuss the relevance of these schemes in achieving fault tolerance in distributed systems. We explored the challenges associated with these schemes in achieving enhanced performance in fault-tolerant distributed systems and examined possibilities for mitigating these challenges.

2.2.1 Active Replication

Active replication is also called the *state machine approach* [53]. The state machine approach is a comprehensive method for implementing fault-tolerant services through server replication and coordinated client interactions with these replicas. It also offers a framework for understanding and designing replication management protocols. Numerous protocols involving data or software replication-whether for failure masking or to enable cooperation without centralized control-can be derived from the state machine approach [4]. In contrast to passive replication, active replication does not have a leader control mechanism, and all processes have an equal role to perform. In this scheme, a client request is executed by every process present within the cluster. To achieve this, the process is expected to be deterministic. Deterministic implies that, given a series of requests with an identical initial state, every process must output a similar sequence of responses and arrive at the same terminal state. A state machine approach was used in [5, 43] to explain the deterministic nature of processes during active replication. This state-machine approach uses a state-machine to represent a collection of processes whose job is to execute client requests arriving at the same final outcome. The initial state produces output of the same state within the process cluster even though some processes may have been down with faults. The fundamental idea behind the state machine technique is that all processes receive and execute requests in the same order, and how requests are distributed across processes may be broken down into two criteria [54]:

1. Agreement: Every request is received by every non-faulty state machine process.
2. Order: Every non-faulty state machine replica executes the requests it receives in the same relative order. Thus, if a process completes the execution of the request m_1 before completing the execution of request m_2 , then no process completes the execution of the request m_2 before completing the execution of the request m_1 .

Order protocol ensures that all processes receive identical operations or abort. Thus, order protocol guarantees the enforcement of the requirements outlined above. In practice, several real processes are non-deterministic, which is one of the active replication's demerits. However,

active replication is needed in real-time systems that need to produce quick responses despite the presence of a fault.

When a client sends a message m containing request, as depicted in Figure 2.1, it sends it to all the process replicas within the cluster. Each process executes the request, modifies its state, and sends a message m containing the reply to the client. Two things happen before the client receives the reply: either (i) it takes delivery of the first reply or (ii) it takes delivery of a quorum of similar replies.

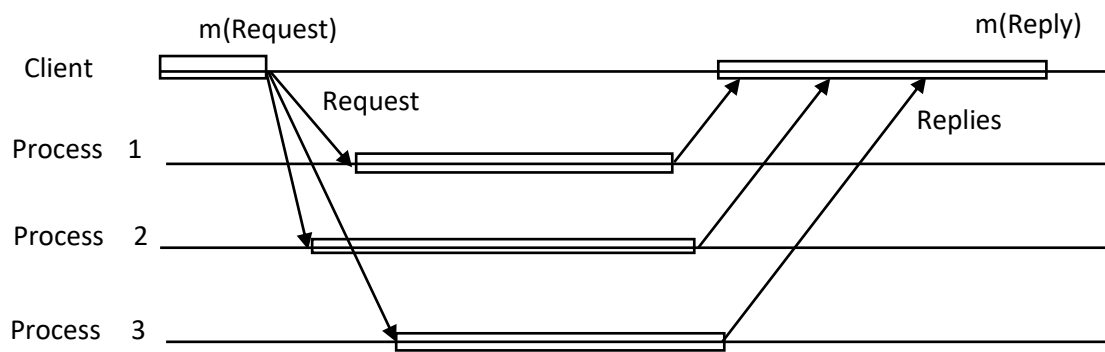


Figure 2.1 Active Replication [52]

For Byzantine failures, a fault-tolerant state machine group must consist of at least $3f+1$ processes to ensure correct total ordering; however, with correct total order ensured, $f+1$ identical results are enough to select the correct response. In scenarios as depicted in Figure 2.1, where Byzantine failures are excluded, and only crash assumption is considered, as in our assumption and similar to the Raft protocol, a fault-tolerant system with $2f+1$ processes are sufficient for correct ordering. However, in a crash assumption, there are two models: asynchronous and synchronous, see Table 2-1. In an asynchronous model, the maximum bound on transmission delay of messages is not known. This means a process might be mistakenly suspected of having crashed, even though it is still operational. As a result, accurate failure detection is not possible in asynchronous model. To ensure correct total order despite these limitations, $N=2f+1$ processes are required to tolerate up to f crash faults. This configuration ensures that, even if up to f processes fail, the remaining $f+1$ operational processes form a majority, allowing the system to maintain correct ordering and any response can be selected for the output. In contrast, synchronous model has a known and constant bound on transmission delay of messages. In this model, when a process is suspected of crashing, it is guaranteed to have actually crashed, enabling accurate failure detection. This reliability reduces the required process to $N=f+1$ for reaching total order. After ensuring correct total order, any response from

among the $f+1$ process replicas, is enough to select the correct response [53].

Table 2-1 Fault Type

Fault Type	Ordering Requirements for f	Output Selection
Byzantine Fault	$N=3f+1$	$(f+1)$ identical results
Crash (Asynchronous)	$N=2f+1$	First response
Crash (Synchronous)	$N=f+1$	First response

One of the challenges of active replication is request ordering when multiple clients are involved in the presence of network delays or failures. An appropriate communication paradigm is needed to maintain the order of arriving requests and responses. This is where order protocol plays a significant role by ensuring the clients' requests are received, processed, and replies sent to the clients according to a definitive order.

2.2.2 Passive Replication

A dedicated process called the primary is solely responsible for handling client requests in this scheme and updates the status of every secondary or backup process within the cluster. This means that the primary receives a client message m containing request, executes it, and then sends state updates to the secondary processes before sending a response to the client [55]. In Figure 2.2, a client sends a message m containing request to the primary process. The primary process receives the request and processes it (Phase 1). When the primary finishes the execution, a reply to the client is ready and updates its state.

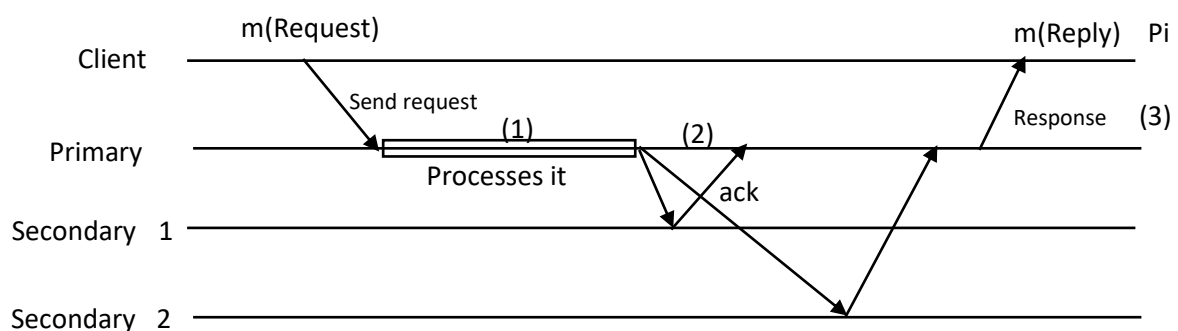


Figure 2.2 Passive Replication [52]

After this, the primary communicates to the secondary with a message consisting of the request-id, response, and state update. When the secondary receives the communication, they update their state and send feedback to the primary (Phase 2).

The moment the primary has taken delivery of the feedback from every correct secondary process, it sends a reply back to the client (Phase 3) [14].

As long as the primary process works correctly, the scenario depicted in Figure 2.2 guarantees linearizability. A consistency model known as linearizability [56] for distributed systems offers a solid assurance over the apparent order of operations. It guarantees that the result of each execution is the same as if all of the actions were carried out sequentially by respecting the order of the operations in real-time. The order by which the primary receives the $m(\text{request})$ determines the total order of every request to the message m . The challenge arises if linearizability must be guaranteed despite the failure of the primary in a cluster of several processes. The failure of the primary triggers three scenarios:

Scenario 1:

The system update is lost when the primary fails before sending the state update message to the secondary. The client will likely have timed out if there is no response from the primary. The challenge here is electing a new primary and having the client resend its request so that the new primary will re-execute it. For the client to resend its request, it must be aware that the old primary has failed. Reliable multicast is required to correct this abnormality, and some order algorithms in the literature have implemented primary election, for instance, the Raft Protocol. Reliable multicast ensures that message broadcast by a sender reach all intended destinations despite a crash.

Scenario 2:

The primary fails after sending state updates to the secondary, but the client has not received a reply. This is more challenging to handle as atomicity needs to be maintained: either all the processes receive the state update, or none receives it. When none of the secondary receives the update, it is the same as **Scenario 1**. When all the secondary received the state update, it means that their state reflects the execution of the client's request, but the client has not received a reply. Hence, a new primary needs to be elected, and the client must resend their request. Atomic multicast ensures that the new primary does not execute the same request twice. This mechanism will ensure that the client attaches a unique id to its request (id, request) before sending it. So, when the primary receives the request, it crossmatches the new request id with the previously

executed request id. If there is a match, it immediately replies to the client. However, if the id does not match any of the already processed requests. It then proceeds to execute the request, updates the secondary, and responds to the client.

Scenario 3:

The primary fails after the client has taken delivery of the reply. This is easy to implement when there is a reliable failure detection mechanism in place and very challenging to implement if there is an unreliable failure detection mechanism. Researchers [57-59] have developed numerous strategies for mitigating this problem.

Data Replication Trade-offs:

Designing replication schemes in distributed systems requires navigating a delicate balance of trade-offs to optimize system performance and reliability. At the heart of this challenge lies the CAP theorem [60], which posits that a distributed system can achieve at most two out of three guarantees consistency, availability, and partition tolerance in the midst of a network partition. Beyond these foundational elements, other factors such as read and write performance, latency, fault tolerance, and storage overhead must also be considered. The trade-off between availability and consistency is especially critical. Consistency ensures that all processes in the distributed cluster possess identical copies of data at the same time, providing clients with a coherent and predictable view of the system. Conversely, availability guarantees that the system can execute and respond to client requests, even during network partitions. In scenarios that prioritize high availability, consistency may be sacrificed, leading to temporary divergences in data across replicas. These trade-offs are influenced by the consistency model adopted, ranging from strong consistency to eventual consistency. Healing or state reconciliation after partitions plays a central role in managing these trade-offs, particularly in scenarios involving partitions. When partitions occur, healing involves detecting and recovering from disruptions to restore a consistent and unified system state. This healing procedure typically includes:

1. **Failure Detection:** Mechanisms such as heartbeat messages, timeouts, or failure detectors identify faulty processes or network partitions.
2. **State Reconciliation:** Processes exchange data to resolve inconsistencies. Techniques like quorum-based replication, conflict-free replicated data types (CRDTs) help synchronize states across partitions.

3. **Reintegration of Recovered Nodes:** When failed nodes recover, they must synchronize with the current state of the system. This is often achieved through snapshot synchronization, where a recovering node receives a full or partial snapshot of the latest state from operational nodes, or incremental update propagation, where only changes made during the downtime are sent to the recovering node to bring it up to date.

Healing ensures that despite temporary inconsistencies or disruptions, the system eventually converges to a consistent state, maintaining fault tolerance and reliability. Read and write performance are additional critical considerations in designing replication schemes. Data replication improves read performance by enabling clients to access data from nearby replicas, reducing latency. However, write performance is often impacted by the need for coordination among replicas to preserve consistency. For example, synchronous replication—where writes must be acknowledged by a quorum before completing—introduces latency due to communication overhead. One key benefit of data replication is its contribution to fault tolerance, enabling systems to continue functioning even in the face of partitions. Fault tolerance ensures that the system remains operational during partitions, but maintaining consistency during partitions adds complexity.

Healing processes, as described earlier, are essential for addressing this challenge. However, replication also introduces storage overhead, as multiple copies of data must be maintained across nodes. Techniques like data sharding [61], where data is distributed across nodes in chunks, and partial replication [62], where only a subset of nodes holds replicas of specific data, help mitigate storage costs.

In conclusion, designing an efficient replication scheme requires a thorough understanding of these trade-offs. Decisions should be driven by the specific goals and requirements of the distributed system, such as the desired balance between consistency, availability, fault tolerance, and performance. Additionally, implementing robust healing mechanisms is essential for ensuring that the system can recover gracefully from failures and converge to a consistent state. Striking the right balance among these factors is critical for building resilient, high-performance distributed systems.

Of the three axes of CAP theorem, we assure consistency and availability by disallowing network partitions. We not only assume that partitions do not occur but also that communication is reliable: every sent message is received by its destination(s). These assumptions mean that (i) the task of state reconciliation need not be addressed and (ii) availability is assured except for periods when process crashes are being dealt with.

2.3 Atomic Broadcast and Multicast Protocols

Atomic broadcast and multicast protocols are network protocols required to send messages across a group of processes to ensure that such messages are sent reliably and arrive at their target destinations in the same series of patterns sent. These protocols are called atomic because all messages arrive in the same sequence they were sent, or the message delivery is terminated. The following requirements must be adhered to maintain atomicity while sending a message to its destination [18, 63]:

- R1.** Validity: if a correct process TO-broadcasts a message m , then it eventually TO-delivers m .
- R2.** Uniform Agreement: if a process TO delivers a message m , then all correct processes eventually TO-delivers m .
- R3.** Uniform Integrity: For any message m , every process TO-delivers m at most once, and only if m was previously TO-broadcast by sender(m).
- R4.** Uniform Total Order: if processes P_i and P_j both TO-deliver messages m_i and m_j then P_i TO-delivers m_i before m_j , if and only if P_j TO-delivers m_i before m_j .

The delivery cannot be reversed when a message is delivered to its target process. Therefore, the protocol is unable to alter any violation of the specified requirements. The meeting of **R1-R4** raises two issues, **S1** and **S2**, that must be addressed by all multicast protocols.

Suppose m is to be sent to a group of targets or destinations, m_dst , where m_dst holds the source of the message sent by the sender. S1 and S2 are expressed as follows:

- S1.** If a correct $r \in m_dst$ receives a message m , then every correct $r' \in m_dst$ must also receive m . This ensures that R1 and R2 are obeyed and not breached.
- S2.** Every r that receives m should decide the best period to deliver m to ensure that R3 and R4 are obeyed and not breached.

It is challenging to ensure that S1 and S2 are obeyed, but in [18], group membership and quorum-based protocols were used to bring the desired solution.

2.3.1 Broadcast

A broadcast is described in [18, 63] as a one-to-many network protocol that only allows messages to be transmitted between a single destination set, with each broadcast received by all destinations. For instance, suppose that we have five target processes; then it means that $|t.p| = 5$ will always be true, provided that none of them failed and no new device was added.

The limiting of one-to-many destinations to a singular destination can yield efficient performance compared to protocols allowing multiple destination sets, particularly when the number of target machines is less than the total number of processes within the cluster ($|t.p| < 5$). However, it is noteworthy that the performance of broadcasts diminishes as the number of target processes increases.

2.3.2 Multicast

Multicast protocols [64, 65] provide a platform where message transmission is sent simultaneously to a group of destination processes. It transmits messages from a single process to a group of processes that indicate interest in receiving such messages. There are two groups in multicast- the ones that enable disjoint target sets and those that allow overlapping target sets. An overlapping multicast ensures that a single process can belong to many multicast groups at the same time. Conversely, disjoint multicast guarantees that a single process can only belong to one multicast group at a time. Each group in a disjoint multicast is mutually exclusive, and processes must explicitly join or leave one to switch to another. The design of the disjoint multicast protocol is not a challenging task since a quorum of the broadcast protocols can be changed to disjoint protocols with only a few modifications. The design of a multicast protocol that allows overlapping target sets is a non-trivial task since R4 requirements must be maintained for every message going to the different target or destination sets.

Assuming process P_i sends m_i to a target or destination = $\{e, f, g\}$ and process P_j sends m_j to a target = $\{f, g, h\}$, the problem is how to ensure that a similar destination $\{f, g\}$ deliver these messages in a similar order i.e. either m_i before m_j or m_j before m_i . The sustenance of S2 is problematic also since we must make sure that $\{f, g\}$ does not miss m_i or m_j in a manner that will not be antagonistic to performance.

2.3.3 Reliable Multicast

Reliable multicast (*rmcast*) protocol provides a platform that helps a process to multicast (*mcast*) a message to a set of processes in a manner that guarantees all faultless destinations receives a similar message even though a few of the processes and the multicast originator might be maliciously faulty [66]. *rmcast* is a foundational communication protocol that is essential for ensuring the security of distributed computing in many forms. It does not entangle itself with message ordering but is only concerned with guaranteeing message delivery.

However, reliable FIFO, reliable Causal, reliable Total, or reliable Hybrid protocols may be built in addition to *rmcast*.

When the originator of a multicast fails, it is possible that some correct processes did not receive the message while some correct processes received the message. This problem is solved by allowing receivers to aid the originator of the multicast by multicasting the received message to every correct set of processes within the cluster.

Suppose that two correct processes, P_i and P_j are such that P_i received a multicast m while P_j did not receive the multicast. Then P_i will have to send the multicast m sequentially to every correct set of processes, which includes P_j , thereby guaranteeing that P_j will receive m . This is how the reliability of the *rmcast* protocol is achieved, bringing about the solution to the problem identified in Scenario 2 of passive replication.

2.3.4 Atomic Multicast

Atomic multicast (*amcast*) provides a platform that guarantees that if a message is delivered, it must be delivered to every correct process (agreement) in a similar order of priority to every process (order) and within a fixed time of known interval (termination). The sending process sends the message and must always be delivered unless a few of the processes within the group are unreachable (validity) [67]. Atomic multicast's main objective is to ensure that, regardless of faults such as process crashes, either every process in the multicast group receives the message or none of them does.

Atomic *mcast* is a fundamental communication paradigm for scalable and highly available systems or applications. It is used to generalize the concept of the atomic broadcast. Like atomic multicast, atomic broadcast ensures that the same set of messages are delivered by all correct processes in the same order.

Finally, atomic multicast and atomic broadcast are both methods for ensuring the delivery of messages in distributed systems. The main distinction is their scope: atomic broadcast provides atomic delivery to every process in the system, whereas atomic multicast ensures atomic delivery to a particular group of processes. Both protocols are essential when reliability and consistency are crucial, although they have more overhead than simple multicast or broadcast protocols.

2.4 Mechanisms For Message Ordering

In the literature [1, 18], five total order broadcast protocols for coordinating message ordering within a cluster were identified. A participating process can assume three distinct roles within the protocols: sender, destination, or sequencer. A sender process is one from which a message originates. A destination process is the recipient to which a message is directed. In contrast, a sequencer process is not inherently a sender or destination but plays a role in the ordering of messages. It is noteworthy that a particular process may concurrently undertake multiple roles, such as being a sender, sequencer, and destination simultaneously. In this section, we will examine them because of their relevance to our study and also introduce failure detection and the consequences of each type of failure detections since process replicas can fail in practice.

2.4.1 Fixed-Sequencer Protocol

In fixed sequencer protocols [68-73], as shown in Figure 2.3, one of the processes is chosen through election as the sequencer and is in charge of ensuring the ordering of messages. The sequencer is unique and does not hand over its role to another unless it fails. So, if there is no failure, the sequencer maintains its role. A process that needs to send a message m must route it through a sequencer. The sequencer on receiving m , attaches a sequence number to the message m and send it to its destination. The destination process must deliver the message following the sequence number.

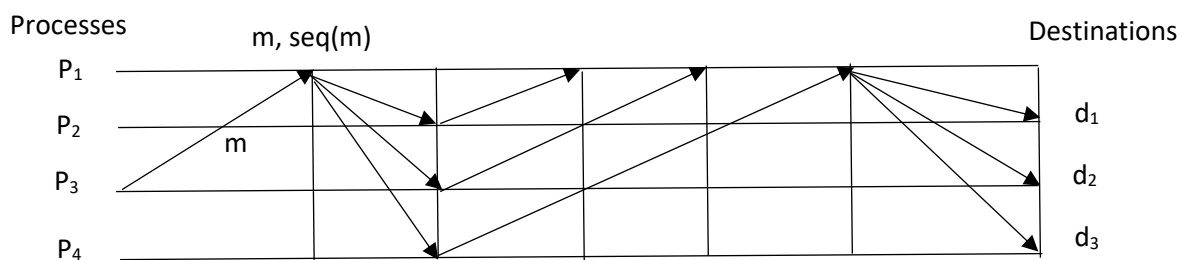


Figure 2.3 Fixed-Sequencer Protocol [18]

According to [74], there is an existence of three variants, and every one of them uses separate communication strategies:

- (i) Unicast-Broadcast (UB): The protocol involves initiating a unicast communication with the sequencer (see Figure 2.4a), which is then followed by a broadcast from the sequencer. This variation results in a reduced number of messages and represents the most straightforward of the three approaches. This approach was adopted in [75].

- (ii) **Broadcast-Broadcast (BB)**: The protocol entails initiating a broadcast to all destinations and the sequencer, followed by a second broadcast from the sequencer (see Figure 2.4b). While this approach results in a higher message count compared to the previous method, it can be advantageous in broadcast networks. Additionally, it has the potential to alleviate the load on the sequencer and facilitate better tolerance for the sequencer's crash. This approach was adopted by the Isis toolkit [76].
- (iii) **Unicast-Unicast-Broadcast (UUB)**: The protocol unfolds through the following sequence of steps as shown in Figure 2.4c: the sender initiates a unicast request for a sequence number from the sequencer. The sequencer responds with a sequence number through unicast communication. Subsequently, the sender broadcasts the sequenced message to the destination processes. It is important to note that the protocol can withstand failures, but the process is complex in nature.

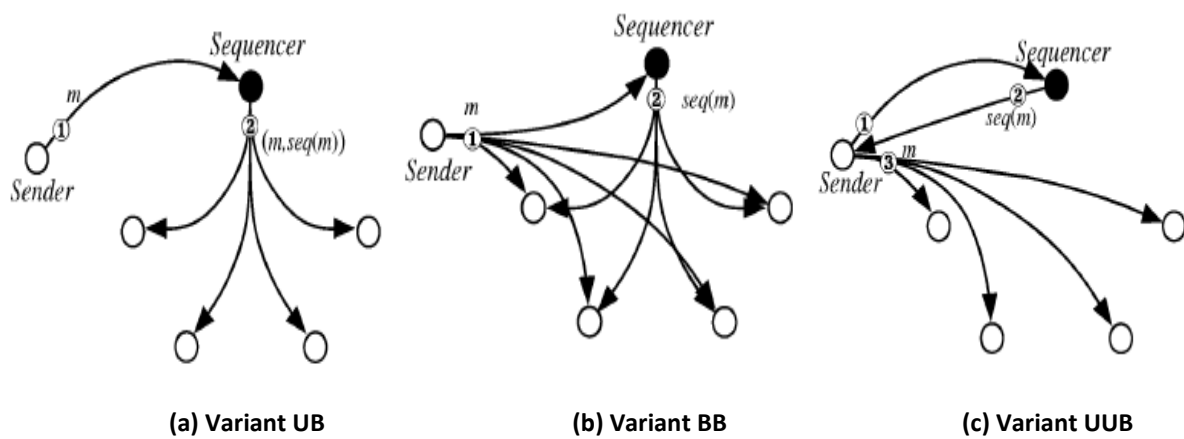


Figure 2.4 Common Variants of Fixed Sequencer Protocol [18]

In situations where preserving an ordered sequence of events is essential, like distributed file systems, distributed database systems, or other applications where consistency is a top priority, fixed sequencer protocols are frequently utilised. The deterministic order that the sequencer provides can make the development of some distributed algorithms easier while guaranteeing a coherent picture of the system state, notwithstanding possible scalability issues. The sequencer also becomes a problem since it must receive all the feedback from the processes and take delivery of the messages to be sent. This protocol shows a linear latency compared to the number of processes, N , but has a decreased throughput.

2.4.2 Moving Sequencer Protocol

This protocol is similar in operation to that of fixed sequencer protocol, except that the duty of a sequencer is transmittable to other processes even amid failures. Thus, protocols utilizing moving sequencers [77-80] have been proposed to address the limitation of fixed sequencer protocols. Figure 2.5 illustrates a 1-to-N broadcast of a single message and the sequencer is selected from a pool of processes. It is evident from the figure that delivering one message per round is impractical for the moving sequencer protocol. This limitation arises because the token must be received simultaneously with the broadcast messages, preventing the protocol from achieving high throughput. However, the code executed by each process is more intricate compared to a fixed sequencer, contributing to the preference for the latter approach. A moving sequencer aims to spread the load among the sequencers so that the problem of using a fixed sequencer is avoided. If a process P_i desires to send a message m , it broadcasts m to every other process within the system. When the processes have taken the delivery of the message, they cache it in their receive queue. If the process P_j holding the token decides to send a message, it attaches a sequence number to the first message in its receive queue and sends the message in conjunction with the token.

Processes

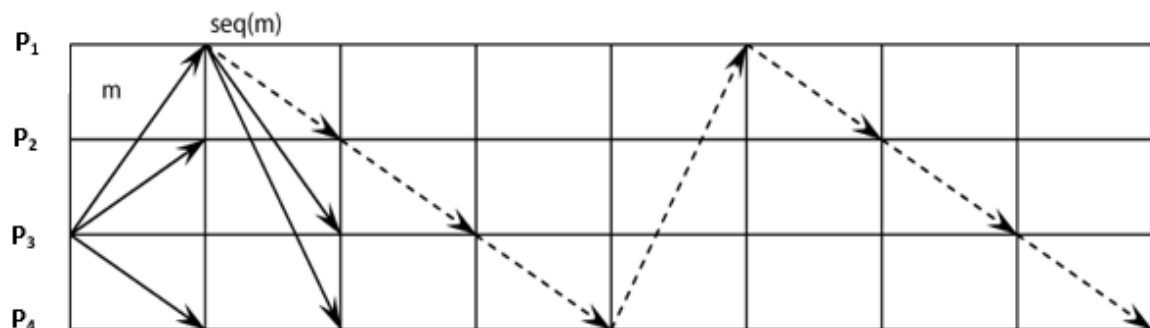


Figure 2.5 Moving Sequencer Protocol [18]

A message is termed delivered when the sender has received feedback from every process acknowledging the receipt of the message. The token is responsible for gathering feedback or acknowledgement. However, a fixed sequencer is preferable to a moving sequencer for the following reasons: firstly, the fixed sequencer is easier and simpler to implement. Secondly, the fixed sequencer has the advantage of better latency than the moving sequencer, though it has a better throughput. Thirdly, a fixed sequencer is more reliable, trusted, connected, and faster than a moving sequencer.

2.4.3 Communication History

In communication history protocols [13, 81-84], similar to privilege-based algorithms, the sequencing of message delivery is dictated by the senders. However, in contrast to privilege-based protocols, processes have the flexibility to broadcast messages at any point in time, and the establishment of total order is achieved by strategically delaying the delivery of messages. Typically, these messages are equipped with timestamps, using logical clocks. This helps the process to check the messages received in order to know when to send their messages without breaching the message total ordering.

The destinations, monitor the messages generated by other processes along with their timestamps, forming a record of the communication history within the system. This history is then leveraged to determine when the delivery of a particular message will no longer violate the total order constraint.

There exist two fundamentally distinct variants of communication history algorithms. In the first variant, termed causal history, these algorithms utilize a partial order derived from the causal history of messages. This partial order is transformed into a total order, with concurrent messages arranged based on a predefined function. In the second variant, referred to as deterministic merge, processes send independently timestamped messages (not necessarily reflecting causal order), and the delivery unfolds according to a deterministic policy for merging the streams of messages from each process. However, communication history-based protocols suffer from poor throughput as they depend on a quadratic number of messages exchanged for each message to be total order-delivered.

2.4.4 Destination Agreement

In this protocol, the delivery order is a product of agreements between processes. The agreement to be made comes in three variations: (i) agreement on the message sequence number, (ii) agreement on the message set, or (iii) agreement on the message order acceptance as proposed. The protocols [57, 82, 85-87] using destination agreement have poor throughput performance since a lot of messages will be produced for every message broadcast. Furthermore, their style of achieving agreement is costly concerning latency and message complications.

2.4.5 Failure Detection

The concept of failure detection was introduced by Chandra and Toueg [57, 88] and is defined in terms of an abstract module established at each process within a distributed system so that it

can provide information related to the operational status of the other processes in the system. In other words, each process has access to a local failure detector module and each local module monitors a subset of the processes in the system and maintains a list of those that it currently suspects to have crashed. Chandra and Toueg assumed that each failure detector module can make mistakes by erroneously adding processes to its list of suspects: that is, it can suspect that a process p has crashed even though p is still running. If this module later believes that suspecting was a mistake; it can remove p from its list. Thus, each module may repeatedly add and remove processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects.

It is important to note that the mistakes made by an unreliable failure detector should not prevent any correct process from functioning according to its specification, even if that process is (erroneously) suspected to have crashed by all the other process replicas. For example, consider an algorithm that uses a failure detector to solve atomic broadcast in an asynchronous system. Suppose all the failure detector modules wrongly (and permanently) suspect that correct process p has crashed. The atomic broadcast algorithm must still ensure that p delivers the same set of messages, in the same order, as all the other correct processes. Furthermore, if p broadcasts a message m , all correct processes must deliver m .

Chandra and Toueg [57] demonstrated that certain failure detectors can solve consensus in systems regardless of the number of process failures, while others require a majority of processes to be correct. Particularly noteworthy is $\Diamond W$, the weakest class of failure detectors. Informally, a failure detector belongs to $\Diamond W$ if it satisfies the following two properties:

Completeness: There is a time after which every process that crashes is permanently suspected by some correct process.

Accuracy: There is a time after which some correct process is never suspected by any correct process.

Such a failure detector can make an infinite number of mistakes: each local module can repeatedly add and then remove correct processes from its list of suspects, reflecting the inherent difficulty of distinguishing between slow processes and crashed ones. Additionally, some correct processes may be erroneously suspected of having crashed by all other processes for the entire duration of the execution. However, Chandra and Toueg demonstrated in their research paper that $\Diamond W$, a failure detector that offers minimal information about which processes have crashed, is sufficient to solve agreement in asynchronous systems with quorum

of correct processes. Thus, any failure detector used to solve consensus must provide at least as much information as $\diamond W$. Consequently, $\diamond W$ is indeed the weakest failure detector capable of solving consensus in asynchronous systems with a majority of correct processes.

Formal Definitions

Some basic terminologies related to failure detectors are defined as follows [57, 88]:

1. **Failure Patterns:** Processes can fail by crashing, that is, by prematurely halting. A failure pattern is defined as a function $F: T \rightarrow 2^{\Pi}$, where $F(t)$ represents the set of processes that have been crashed till time t . When a process is crashed, it does not recover that is $F(t) \subseteq F(t+1)$. Similarly, the set of processes is defined as $\Pi = c + f$ where Π , c and f denote the set of processes, set of correct processes, and set of faulty processes respectively.
2. **Failure Detector History:** According to Chandra and Toueg [57], failure detector history is defined as a function that provides the list of suspected processes, which have been suspected by a failure detector till a given time. Mathematically, a failure detector history is defined as a function $H: \Pi \times T \rightarrow R$, where $R=2^{\Pi}$ is a range, T is a time, Π is a set of processes and H is a failure detector history.
3. **Failure Detector:** A failure detector D is defined as a function $D: F \rightarrow D(F)$, where F and $D(F)$ denote the failure pattern and set of failure detector histories respectively. It maps the failure pattern to a set of failure detector history. The behaviour of a failure detector for failure patterns was shown in each respective history. Every process in a system has its local failure detector module.

Failure Detectors Properties

Completeness and accuracy properties are used to measure the reliability of the failure detectors [57]. Broadly speaking, *completeness* requires that a failure detector eventually suspects every process that actually crashes, while *accuracy* restricts the mistakes that a failure detector can make.

Completeness:

There are two types of completeness properties which are satisfied by a failure detector D . These properties are described as follows:

- *Strong Completeness:* Eventually every process that crashes is permanently suspected by every correct process.

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in \text{crashed}(F), \forall q \in C, \forall t' \geq t: p \in H(q, t')$$

- *Weak Completeness*: Eventually every process that crashes is permanently suspected by some correct process.

$$\forall F, \forall H \in D(F), \exists t \in T, \forall p \in \text{crashed}(F), \\ \exists q \in C, \forall t' \geq t: p \in H(q, t')$$

Accuracy:

There are four variations of accuracy property that a failure detector D can satisfy:

- *Strong Accuracy*: No process is suspected before it crashes.

$$\forall t \in T, \forall p, q \in (\prod - F(t)): p \notin H(q, t)$$

- *Weak Accuracy*: Some correct process is never suspected

$$\forall F, \forall H \in D(F), \exists p \in C, \forall t \in T, \forall q \in (\prod - F(t)): p \notin H(q, t).$$

Since it was very difficult to achieve both accuracy properties so Chandra and Toueg [57], proposed an eventual accuracy property which is further classified into two more properties as follows:

- *Eventual Strong Accuracy*: There is a time after which correct processes are not suspected by any correct process. Formally, D satisfies eventual strong accuracy if:

$$\forall F, \forall H \in D(F), \exists t \in T, \forall t' \geq t, \forall p, q \in (\prod - F(t)): p \notin H(q, t')$$

- *Eventual Weak Accuracy*: There is a time after which some correct process is never suspected by any correct process. Formally, D satisfies eventual weak accuracy if:

$$\forall F, \forall H \in D(F), \exists t \in T, \exists p \in (\prod - F(t)), \forall t' \geq t, \forall q \in (\prod - F(t)): p \notin H(q, t')$$

Failure Detector Classes

A failure detector is said to be Perfect if it satisfies strong completeness and strong accuracy. The set of all such failure detectors called the class of Perfect failure detectors, is denoted by \mathcal{P} . Eight classes of failure detectors are defined with the combination of two completeness and four accuracy properties as shown in Table 2-2. These eight classes of failure detector can be group into a perfect and imperfect failure detector. However, the consequence of using a perfect failure detector, \mathcal{P} , is that it eventually detects all crashed processes (*strong completeness* property) and never outputs false detections, that is, never detects a non-crashed process (*strong accuracy* property). Analogously to a perfect failure detector, the time required to detect a failure in an eventually perfect failure detector depends on the timeout delay. The difference is that the initial timeout for $\diamond\mathcal{P}$ can be set with shorter intervals, allowing it to react faster to failures than \mathcal{P} . While this may lead to false suspicions, these are permissible under the specification of $\diamond\mathcal{P}$ and thus do not cause harm, unlike in the case of \mathcal{P} .

Moreover, an application using the eventually weakest failure detector, $\Diamond W$, may lose *liveness* but not *safety*. This occurs if a crash goes undetected by all processes, leading to correct processes being perpetually and falsely suspected. Thus, if a consensus algorithm assumes the properties of $\Diamond W$ but the actual failure detector misbehaves continuously, processes may be prevented from deciding, but they will never decide on different values (or a value that is not allowed). Similarly, with an atomic broadcast algorithm, processes may stop delivering messages, but they will never deliver messages out of order.

Table 2-2 Eight Classes of Failure Detectors [57]

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect \mathcal{P}	Strong \mathcal{S}	Eventually Perfect $\Diamond \mathcal{P}$	Eventually Strong $\Diamond \mathcal{S}$
Weak	\mathcal{Q}	Weak \mathcal{W}	$\Diamond \mathcal{Q}$	Eventually Weak $\Diamond \mathcal{W}$

2.5 Leaderless Ring-Based Order Protocols

Leaderless order protocols are a group of protocols designed without a designated leader process [89]. Each process within this group of protocols performs equal responsibility; this means that clients can send their request to any process and get their response after the request has been executed without obstructing other parallel commands within the cluster. One of the benefits of this approach is that it removes the consequences of routing every client request through a single process designated as a leader, necessitating evenly distributing loads among the process within the leaderless-based cluster [90].

The research of [91] identified some limitations of leader-based protocols. They believed that routing every client's request through a central coordinator (leader) continuously has the following consequences: Firstly, it undermines scalability by allowing the leader to carry a variable high load as it processes all the requests that arrive in the system [92]. Secondly, there is high latency as the client communicates with a remote leader during geo-replication execution [93, 94]. Thirdly, availability can be disrupted significantly due to long-term leader failures [39, 42, 95]. When a leader fails, the system is unavailable until another leader is elected. This section discusses two leaderless total order (TO) broadcast protocols relevant to our study, identified in the literature [24, 33].

2.5.1 Privilege-Based Protocol

Privilege-based protocols [1, 44, 45, 96, 97] employ a ring topology of processes to establish a process-to-process communication within a cluster and utilize a token that is passed among processes to grant each process the privilege of broadcasting a message within the system. It has been found in [1] that these protocols give a high throughput within the 1 – to – N and N – to – N communication pattern but not in the case of k – to – N ($k \neq 1, N$). That is, throughput increases when one sender sends to every process or when many senders (N) send to many processes (N). But when two (k) processes want to send messages at the same time to every process, it reduces throughput since each sender will be sharing tokens with the other sender to achieve fairness. In this section, we will briefly examine this class of protocols.

Rotating Token (Rottoken) Protocol

In Rottoken protocol [98], a token revolves around a group of processes participating in broadcast communication. It is similar to the Totem single-ring protocol [99], which was used to develop and build a Totem system [45]. When a process desires to broadcast a message, it waits to receive a token before broadcasting the message. The process buffers its messages while waiting. As soon as it takes delivery of the token, the process must broadcast the messages buffered within that period and deliver the token to the succeeding process of the group.

Two things happen when a process receives a token. Firstly, it has to check if it has a message to broadcast; when this is true, it sends the message, augmenting the token to the last message. Secondly, if it has no message to broadcast, it must produce a unique message, attach the token, and send it. This protocol is configured to offer an exceptionally high throughput for programs where every process continuously broadcasts messages to every process within the group. The number of participating processes within each application affects the magnitude of the latency of messages experienced. For instance, if there are N processes with the system, the sender of a message will have to wait for an average of $\frac{N}{2}$. In the worst scenario, token passes can wait for N – 1 token passes, irrespective of the number of senders within the system, even if it is one sender. In real life, this high latency is stabilised by the knowledge that the moment a message is broadcasted, it is done without conflict within the network.

Rotfc Protocol

Rotfc Protocol [44] is similar to Rottoken except that whenever the process holding the token decides to send a message, it puts all the messages together and sends them simultaneously. In this scenario, the token is augmented on the message.

Reqtoken Protocol

This is an enhanced form of Rottoken [44] for cases where only a few subgroups of the entire process broadcast messages while the remainder of the processes are message receivers. In contrast to Rottoken, a process that desires to send a message must first send a token request before receiving the token. On getting the clue of the token requests, the token holder selects one of the processes from among the group of processes requesting the tokens and sends the token to it. As soon as it takes delivery of the token, it puts together all the buffered messages and broadcasts them at once. After this, if it takes knowledge of an impending token request, it augments the message to the token in addition to a new token request and sends. If not, it will hold the token and can send messages anytime the application program wants it to send them. Further enhancement ensured that a process does not send a new token request if it has not taken delivery of the earlier proposal that is, it remembers that it has sent an earlier request and does not need to resend it. The improvement to piggyback a token request on the messages to be sent is such that if a group of processes continuously sends messages, there will be no need to send a token request. The implication is that the token is revolving only among the senders. In a unique scenario where we have only a single sender, no token will be passed around. Hence, the magnitude of latency rests on the real senders. For instance, when there are N processes and only $k < N$ senders, the message sender waits for $\frac{k}{2}$ token passes only, and in the worst case, it waits for $k - 1$ token passes. An additional merit of this protocol is that when a message is broadcasted, there is small or no conflict within the network.

Dynord Protocol

This protocol [44] is designed such that whenever a process desires to send a message, it does that instantly. Nevertheless, whenever a process takes delivery of a broadcast message, it is restricted from sending it out immediately. It must wait for the orderer. The orderer is the process selected to bring order to a broadcast message before it is sent out. The orderer comes up with an order message stipulating the order in which a message will be sent out. Any process that receives the order message must broadcast it according to the order specified in the order

message. The orderer sends out received messages instantly since the orderer must always order messages that it receives.

Further performance enhancement ensures that the order message is augmented with the regular message sent by the orderer. Another process could become the orderer if the current orderer fails its duty or becomes less active. This protocol utilises the ideas used in developing ISIS [85, 100], although most of the enhancement in ISIS was not used in Dynord.

The conceptual latency of this protocol rests on the following factors: first, the one-way delivery of the regular message; second, the one-way delay of the order message; and the time spent between two successive order messages. In real life, one-way delay of messages is anticipated to increase provided that every process is permitted to broadcast messages without restriction, and this reduces throughput.

Dynamic Sequencer (DynSeq) Protocol

This protocol [100] selects a unique process as the sequencer within the system. If any process desires to send a message, it first sends it to the sequencer. The sequencer then sends the message to every participating process and delivers it internally. If the sequencer becomes less functional, the role of a sequencer is passed to another process. This protocol was utilised in building the Amoeba project [101]. The conceptual latency of this protocol is the finest since it needs two-one-way latency only. Moreover, it generates a high network conflict in real life, increasing latency and decreasing throughput afterwards. Also, it reduces the conceptual bandwidth of the entire system by a factor of no less than two.

Dysfc Protocol

This protocol [44] closely resembles DynSeq but modifies the timing of message transmission. In this variant, processes are restricted from sending messages continuously. Instead, messages are buffered, and every d millisecond, they are packed and transmitted as a single packed message. In the experimental setup, d was set to be one millisecond, a duration shorter than the anticipated minimal one-way user-to-user latency. The concept is to introduce a brief delay to each message, significantly boosting throughput. By transmitting only one (packed) message every 1 millisecond, both network contention and the burden on receiving processors are markedly reduced. This reduction in contention and load leads to lower user-to-user latencies compared to scenarios where messages are sent without this deliberate blocking strategy.

Pinwheel Protocol

This protocol [96] was designed and developed using a combination of carefully chosen sequencer and token-based protocols. It used the concept of a revolving sequencer to implement ordering when sending the systems' updates as a good trade-off for a sequencer and token-based protocol. A similar idea was used in the CM protocol [77], although there is a big difference between the CM and pinwheel protocols.

Assumptions of Pinwheel Protocols:

- (i) It assumed that processes are joined together as a network of a distributed system.
- (ii) Processes use the asynchronous mode of communication to send messages among themselves.
- (iii) A message could be lost or reach its desired destination at the end of its allocated time range.
- (iv) The message that gets to its destination is not corrupted.
- (v) Group membership protocol [102] ensures that processes reach an agreement for the 'system's initial state.
- (vi) No process crashes occur, and each process within the system is recognised by its rank (a non-negative integer between 0 and $(N - 1)$)

The implementation of this protocol showed a considerable increase in throughput even in the midst and absence of communication failures over the individual sequencer and token-based protocols.

Token-Based Protocols Using Failure Detectors

In token-based protocols, a token moves around the processes that make up the system, and the process holding the token is responsible for ordering the broadcasted messages, although the process having the token is allowed most times to broadcast messages. Atomic broadcast protocols use unreliable failure detectors (FD) [57] and group membership (GM) [103] to accommodate crashes within the system. Group membership is used to constantly communicate the status of each process to every group member within the system. One of the significant characteristics is the removal of processes presumed to have failed, whereas the reverse is the case of unreliable failure detectors. FD could inform process k that m has failed and simultaneously inform process j that m is active. Both strategies have their peculiar inherent problems, though the problem with FD is lighter when compared to GM. When a GM

removes a presumed failed process, it introduces two inaccurate operations: removing a process and adding a new process.

In contrast, FD does not remove any presumed crashed process. Existing token-based algorithms depend on GM for their operation, though it is implemented ad-hoc. Token-based algorithms using FD were designed and implemented in [97]. The FD token-based model presumed a system that consists of N processes derived from the set $\Pi = \{P_0, \dots, P_{N-1}\}$ with an implied order on the group of processes. The k^{th} successor of a process P_i is $P_{(i+k) \bmod N}$, which is, from now on, simply noted P_{i+k} for the sake of clarity. Similarly, the k^{th} predecessor of P_i is simply denoted by P_{i-k} . The processes communicate by message passing over reliable channels. Processes can only fail by crashing (no Byzantine failures). A process that never crashes is said to be correct, otherwise it is faulty. At most f processes are faulty and the system is augmented with unreliable failure detectors. It ensures that R1-R4 is obeyed to maintain agreement within the system.

Model characteristics: This model is designed to be token-based if:

- (i) Processes are arranged in a logical circle or ring.
- (ii) A single process P_i is built with a FD module that gives the detailed status of this immediate predecessor P_{i-1} .
- (iii) A single process interacts with its $f + 1$ predecessors and successors, where f denotes the accommodated number of failures.

Failure Detector \mathcal{R} :

A new failure detector denoted by \mathcal{R} (stands for Ring) is defined for token-based algorithms in [97]. Given P_i , the failure detector attached to P_i only gives information about the immediate predecessor P_{i-1} . For every process P_i , \mathcal{R} ensures the following properties:

- (i) Completeness: if P_{i-1} crashes and P_i is correct, then P_{i-1} is eventually permanently suspected by P_i and,
- (ii) Accuracy: if P_{i-1} and P_i are correct, there is a time t after which P_{i-1} is never suspected by P_i

The relation *weaker/stronger* between failure detectors has been defined in [57]. Here, [97] shows that (a) $\diamond\mathcal{P}$ is strictly stronger than \mathcal{R} (denoted $\diamond\mathcal{P} > \mathcal{R}$), and (b) \mathcal{R} is strictly stronger than $\diamond\mathcal{S}$ if $N \geq f(f+1)+1$ ($\mathcal{R} > \diamond\mathcal{S}$).

Lemma 1: $\Diamond\mathcal{P}$ is strictly stronger than \mathcal{R} .

Proof: The proof of this lemma can be found in [97].

Failure Detector $\Diamond S2$:

To establish the relation between \mathcal{R} and $\Diamond S$ we introduce the failure detector $\Diamond S2$ defined as follows:

- (i) Strong Completeness: Eventually every process that crashes is permanently suspected by every correct process and,
- (ii) Eventual Double Accuracy: There is a time after which two correct processes are never suspected by any correct process.

(a) $\Diamond S2$ is strictly stronger than $\Diamond S$

$\Diamond S$ and $\Diamond S2$ differ only in their accuracy property: $\Diamond S$ requires that eventually one correct process is no longer suspected by all correct processes, while $\Diamond S2$ requires this for two correct processes. Consequently, by definition, $\Diamond S2$ is stronger than $\Diamond S$, denoted as $\Diamond S2 \succ \Diamond S$.

(b) \mathcal{R} stronger than $\Diamond S2$ if $N \geq f(f+1) + 1$

Lemma 2: Consider a system with $N \geq f(f+1) + 1$ processes and the failure detector \mathcal{R} . The transformation \mathcal{R} into $\Diamond S2$ guarantees that eventually, all correct processes do not suspect the same two correct processes.

The proof of this lemma and the transformation of \mathcal{R} into $\Diamond S2$ can be found in [97]. The transformation of \mathcal{R} into $\Diamond S2$ ensures the Eventual Double Accuracy property if $N \geq f(f+1) + 1$. Since all processes except two correct processes are suspected, the Strong Completeness property also holds. Consequently, if $N \geq f(f+1) + 1$ we have $\mathcal{R} \succcurlyeq \Diamond S2$.

Token Circulation

The token revolves around the processes in circular patterns to bolster the model characteristics, as follows [97]:

- (i) To checkmate the loss of a token due to failures, Process P_i dispatches the token to its $f+1$ successor. i.e., to $P_{i+1}, \dots, P_{i+f+1}$.
- (ii) If process P_i needs the token, it expects to receive it from P_{i-1} except if it doubts the correctness of P_{i-1} .

(iii) If process P_i doubts P_{i-1} , it can receive the token from any of its predecessors.

The work of [97] has shown that failures can be accommodated directly, i.e. it can accommodate incorrect failure suspicions. Hence, failure detector token-based protocols have better merit over group membership in terms of accommodating incorrect failure suspicion and eventual real failures. This protocol also has a better advantage in terms of throughput and latency.

2.5.2 Fixed Sequencer and A Ring Topology (FSR) Protocol

FSR [1] is an extension of the round-based model with the following modifications: for every round r , each process P_i is expected to (i) determine the message $m(i, r)$ to be sent for a round (ii) broadcast $m(i, r)$ to all processes, and (iii) receive single feedback within the same r , except the process fails that send the message. FSR was implemented with a fixed sequencer for message ordering and a ring topology for dispatching the message to achieve high throughput and less complexity for system developers. The ring could also serve as a sequencer apart from transmitting messages. FSR ensures uniform total order property is maintained while sending messages despite the failure of t processes where $t < N$ and N is the sum of the processes within the group. This means that FSR assumes crash failures only, then any response is correct, so $N=t+1$ which implies $t = N-1$.

In this protocol, a process does not send a message straight to the sequencer but to its successor. In a ring structure, it takes at most $(N - 1)$ hops for a round to be completed, that is, for P_i to send a message to P_{i+1} and receives from P_{i-1} at most $(N - 1)$ hop is needed. A round means 1 hop from P_i to P_{i+1} . Two scenarios can be deduced from Figure 2.6: Firstly, if a standard process P_i desires to send a message m , it sends the message m_1 to its successor P_{i+1} , the successor then send it to their own successor, and the circle continues until the message m_1 gets to the leader P_0 .

The leader attaches a uniquely ascending sequence number to the message to ensure they are ordered before forwarding it. The leader then sends the message and the sequence number pair (m_2) to every backup process until the last backup process P_t receives it. So far, the leader and backup processes have not TO-deliver the message. It is the duty of the last backup process P_t TO deliver the message. P_t sends the message m_3 until it gets to the P_{i-1} process and both finally deliver m . Process P_{i-1} sends a feedback message m_4 acknowledging the receipt of m to every Process replicas until it gets to P_{t-1} . Then, at this point, every process can TO-deliver m when they have received m_4 .

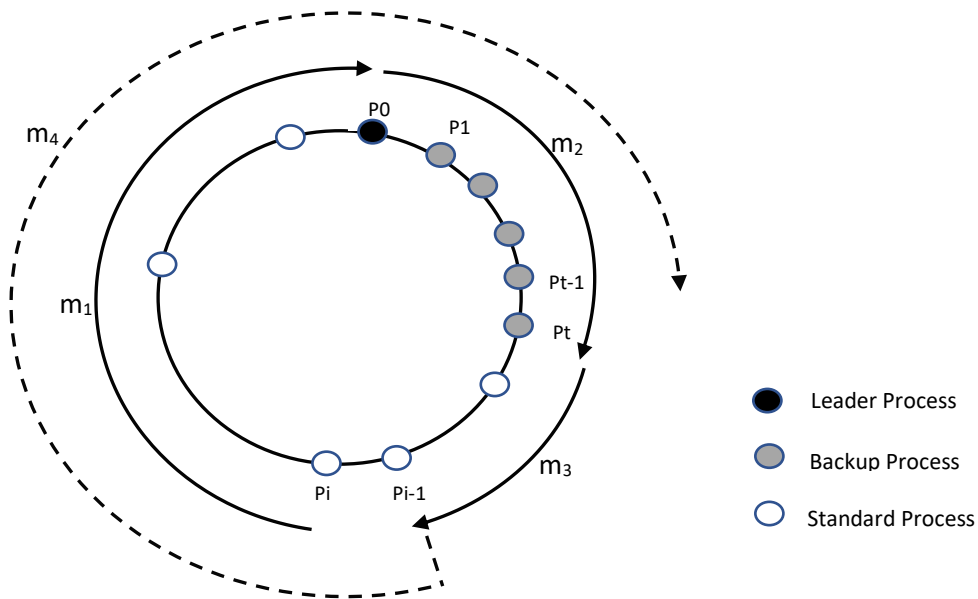


Figure 2.6 FSR Protocol Design [1]

Secondly, if a backup process P_b ($0 < b \leq t$) desires to broadcast a message, it sends the message until it gets to the leader P_0 (this first message is evidently neglected if the leader initiates the TO-broadcast). The leader then sends the message and sequence number until it gets to the process P_{b-1} . From there, an acknowledgement is sent till it gets to P_t . However, backup processes are not allowed TO-deliver m until every process has received the acknowledgement; then, they can TO-deliver m . This scenario demands that a message circulate through the system several times. Moreover, to ensure that high throughput is achieved, the actual message to be TO-broadcast needs to go around the system once.

FSR Fairness

Fairness embodies the principle that each process is afforded an equal opportunity to have its messages ultimately delivered in total order by all processes. Essentially, fairness entails that no single process receives priority over others when broadcasting messages. For example, if two processes are simultaneously broadcasting numerous messages, each process should ideally have a similar number of messages ultimately delivered in total order by all processes. Fixed sequencer protocols discussed in Section 2.4.1 inherently exhibit fairness: each process broadcasting a message sends it directly to the sequencer, which processes incoming messages on a first-come, first-served basis. If multiple messages arrive at the sequencer simultaneously, it serves them in a round-robin manner.

In the FSR protocol, messages intended for total-order broadcast (TO-broadcast) are not directly sent to the sequencer; instead, they are forwarded to the successor process. If all

processes intend to TO-broadcast messages, then in each round, a process can either commence a new TO-broadcast by dispatching a message to its successor or forward messages from its predecessor. Achieving fairness in FSR involves employing a specific mechanism to determine whether a process can initiate a new broadcast or must first forward messages stored in its incoming buffer. In essence, each process maintains a list, called the forward list, of processes to which it has forwarded messages since its last broadcast. When a process initiates a TO-broadcast, it first forwards messages in its incoming buffer that originated from processes not present in the forward list. Figure 2.7 depicts the incoming buffer and forward list of a process p_i aiming to initiate a TO-broadcast. Process p_i forwards messages, m^3_{p2} and m^5_{p3} , before transmitting its own message m . This straightforward mechanism ensures that no process obstructs others from TO-broadcasting their messages.

Incoming buffer	Forward list
m^3_{p2}	p1
m^2_{p4}	p4
m^5_{p3}	p5
m^6_{p3}	

Figure 2.7 Incoming buffer and forward list of a process initiating a TO-broadcast [1]

FSR Latency and Throughput

Several factors influence latency and throughput. We will examine them briefly.

Latency:

For every process, the latency of FSR can be expressed as follows: $L(i) = 2N + t - i - 1$ where i is the position of the process broadcasting within the system with the leader at position 0. The following can be deduced from the expression:

1. The latency is linear when compared with the number of processes N involved, suggesting that FSR could scale very well.
2. The latency is linear when compared to the number of failures, t , accommodated.

3. The position of the sending process within the system has a significant impact on latency. So, the leadership role can be moved to the next process within the system periodically for uniform latency distribution.

Latency Calculation:

Principle of Delivery: A message can be processed once all processes are known to have received m and $(t + 1)$ processes have recorded the message plus sequence number.

For all N processes to receive $m \rightarrow (N - 1)$ hops.

Case 1: Sender is P_i , $i > t$. P_i to P_{i-1} takes $(N - 1)$; all have received and not all know that $(t + 1)$ backups have recorded message plus sequence number; say P_{i-1} completes another full round. The total hops are $2(N - 1)$; of these, P_t to P_{i-1} are redundant (inclusive), and they all know that $(t + 1)$ backups have been stored. So, we have: $L(i) = 2N + t - i - 1$.

Case 2: P_i sends and $0 \leq i \leq t$.

One full circle from P_i to P_{i-1} will ensure all receive that message; one more circle from P_{i-1} to P_{i-2} will ensure the message is backed up with sequence number at $(t + 1)$ processes, to all except for P_{i-1} to P_{t-1} (inclusive); so we need a third round from P_{i-2} to P_{t-1} .

So, $L(i) = 2N - 2 + (t - 1 - (i - 2)) = 2N + t - i - 1$.

Two Remarks:

When $i = 0$, the distance of P_0 from itself should be regarded as N ; for P_0 take $i = N$.

$$L(0) = N + t - 1.$$

Secondly, $L(i) = 2N + t - i - 1$ can also be written as:

$$L(i) = 2N - 1 + t - i, \text{ or}$$

$$L(i) = 2N - 2 + t + 1 - i, \text{ or}$$

$$L(i) = 2(N - 1) + (t + 1) - i.$$

Throughput:

FSR's throughput is at least one. This means that each round completes at least one TO-broadcast (a complete TO-broadcast of message m indicates that all processes TO-delivered m). More categorically, (1) The throughput does not depend on the number of processes that

broadcast messages simultaneously. It takes the latency of $2N + t - i - 1$ for the first message to be delivered by every process within the system, and only a single message is delivered within a particular round. (2) The throughput does not depend on the number N of processes that make up the system. (3) The throughput does not depend on the number t of processes likely to fail. However, although the integration of a fixed sequencer and a ring topology resulted in optimal throughput, the fixed sequencer may pose a performance bottleneck, particularly under high concurrent message transmission. This is because the fixed sequencer determines the message order within the cluster by assigning a sequence number to the messages, potentially causing a negative impact on message latency when multiple messages are transmitted simultaneously.

2.6 Leader-Based Order Protocols

Leader-based protocols are order protocols that use a dedicated leader to achieve total order within the cluster environment. Protocols offering this delivery type must ensure the leader is always alive. If the leader fails, a new leader must be elected with low latency, as the absence of a leader keeps such a cluster environment inconsistent. In this section, we discussed some of the leader-based order protocols relevant to our study.

2.6.1 Paxos

Paxos is a well-known order protocol for achieving fault-tolerant in distributed system applications introduced by Lamport [35, 39]. Paxos belongs to a family of protocols that can reach an agreement or total order in distributed applications [39] in a single decision, for instance, a single replicated log entry. The Paxos order protocol [35] was extensively theoretically discussed in [104] and applied to the community [105-107] for nearly two decades. It achieves total order by ensuring that the value proposed by one of the processes, called the proposer, is received by all other processes (or a majority of them), called acceptors, even if some processes might fail. Paxos enforces the safety requirements for agreement as follows [108]:

- Processes may choose only a value proposed by a process (the proposer).
- Only a single value is chosen which cannot be changed, and
- A process only becomes aware that a value has been chosen if it actually chose such value.

In Paxos, there are three unique roles a process could perform: Proposers, Acceptors, and Learners. A process may assume more than one role. The proposers propose a value to the

cluster's acceptors (processes) based on the client's request. If the proposer receives a quorum of feedback from the acceptors, the value is chosen and stored in the log; otherwise, the value cannot be chosen or stored in the log. Instead, the value returns to the proposer, which resends it with a new id. This process of proposing a value continues until most acceptors receive it. Hence, the value passes through many stages before reaching its final state, the chosen state. This scenario is captured diagrammatically in Figure 2.8.

The proposer must receive a majority $\left(\frac{N+1}{2}\right)$ of acceptors' responses before choosing the value, where N is the number of acceptors. Hence, a Paxos needs $2f + 1$ acceptor processes to accommodate f failures. Several optimisations have been proposed as an improvement to the core Paxos protocol and these improvements are documented in the following works of literature [91, 92, 109-112].

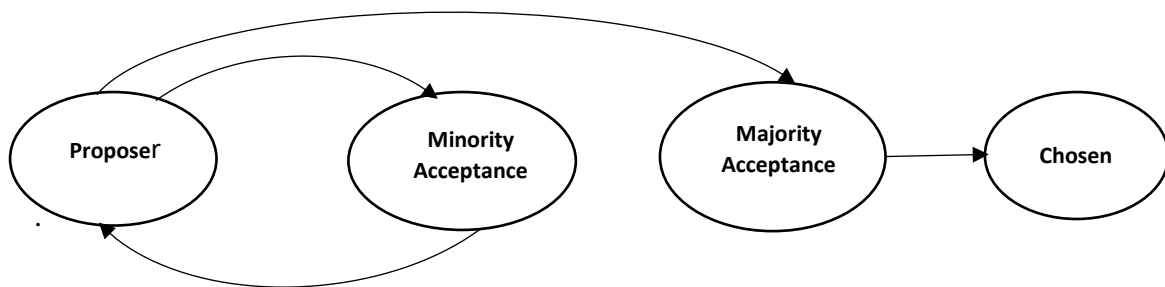


Figure 2.8 States of Values [35]

However, along the many benefits of Paxos, it also has two notable demerits. Firstly, Paxos may be difficult to understand since the original presentation mystified its many readers. Many attempts have been made to present Paxos with simple terminology to aid understanding [105, 113], yet this explanation, which uses a single-decree subset, is still demanding. Secondly, Paxos lacks a good framework for developing practical systems. This is because there is no generally acceptable algorithm for multi-Paxos. Also, implementing a fault-tolerant replicated log on Paxos is a non-trivial task. Some of the complexities are due to flaws found in the real world, such as hard disk failures or finite resources. The algorithmic solutions to these challenges in core Paxos were discussed extensively and theoretically in [38]. Due to the many complexities and problems associated with Paxos, [3] opined that Paxos is neither suitable for system development nor educational purposes.

2.6.2 Ring Paxos

In the contemporary leader-based work, Ring Paxos [114], a logical ring is partially used: IP-multicast for the leader to disseminate the message replacing direct node-to-node

communication and opting for broadcast-style communication instead, unicasts for acknowledging the leader and a logical ring for the leader to confirm the ordering for messages. Ring Paxos functions similarly to regular Paxos, primarily differing in its communication protocol. In Ring Paxos, a designated coordinator node receives proposals from clients. However, before assuming its role as coordinator, the node must validate itself through Phase 1 of the Paxos protocol. To achieve this, the coordinator utilizes IP-multicast to broadcast a message containing a ballot/round number and the ring configuration, including the designated starting node, to all acceptors. Upon receiving the message, each acceptor compares the received ballot with its own knowledge and only acknowledges the node as the new coordinator if the received ballot is the highest it has encountered thus far. Each acceptor independently responds to the coordinator, providing feedback on the success of its validation. Additionally, the coordinator receives information on any incomplete or uncommitted values that require recovery.

Ring Paxos offers a throughput closer to optimal throughput but reduces ordering latency due to the use of IP-multicasts unlike LCR [2], which offers slightly better optimal throughput because of utilising the ring topology but exhibits a higher latency which increases linearly with the number of processes within the ring.

2.6.3 ChainPaxos

ChainPaxos [115] is an innovative leader-based order protocol combining the ring topology and is designed for the high-throughput replication of deterministic services. The primary objective is to reduce the communication cost of the protocol, aiming for optimal throughput in both read and write operations. To achieve this goal, several complementary techniques were employed. For enhancing write performance, the protocol leverages an efficient pipelined communication pattern among processes. This pattern, previously explored and proven effective in approaches like Chain Replication [116], minimizes and distributes the number of messages propagated by each process during consensus. This strategic approach significantly contributes to maximizing the throughput of write operations. To address read operations, a novel scheme was introduced for linearizable reads served by a single process. Importantly, this is achieved without incurring additional communication costs, albeit with a slight increase in latency. This innovative approach not only minimizes the communication overhead of ChainPaxos but also contributes to boosting its overall throughput. In general, ChainPaxos integrates these techniques to optimize communication efficiency, reduce overhead, and enhance throughput for both read and write operations in the replication of deterministic

service. Unlike many recent proposals, ChainPaxos does not delegate membership management to an external coordination service like Zookeeper [95]. Instead, it incorporates an internal membership management solution. This approach allows continuous operation during reconfigurations, ensuring that the system's fault tolerance remains independent of external services. Consequently, increasing the number of processes in ChainPaxos effectively boosts the system's tolerance to faults. In contrast, systems relying on an external coordination service have their fault tolerance linked to that service. Recent findings also highlight the challenges associated with using an external coordination service, making the system more susceptible to network partitions [117] and requiring additional logic to maintain correctness. The primary objectives of ChainPaxos's design are twofold: (i) to minimize the message processing load on each process during fault-free operations, ensuring uniform load distribution for maximum throughput, and (ii) to integrate an efficient fault-handling mechanism into the algorithm, utilizing Paxos messages to eliminate dependence on external services.

2.6.4 Chubby

Chubby [41, 42] is a distributed order protocol (services) designed to incorporate lock service within a Google distributed system. The motive of Chubby's lock service is to enable its clients to harmonise their tasks and achieve agreement within their domain of operations. A lock service ensures that among multiple processes that might try to do the same write operation, only one eventually does it (at least one at a time). This prevents concurrent processes from accessing the same write operation, changing the system's state. If the lock fails and two processes work on the same request concurrently, the result is inconsistent data, violating the total order principle. So, when a process performs a write request, it puts the lock (exclusive) in place and only releases it when it has replicated the outcome of the write operation to other processes, thereby maintaining order within the Chubby cell.

Chubby Operations

A typical chubby cell comprises a group of five processes. This number of processes is needed to lower the chances of parallel failure. Thus, Chubby system requires $N=2f + 1$ nodes to tolerate f node crashes. The processes elect a master among themselves using a Paxos consensus protocol. To become a master, the process must receive a quorum of the votes cast by other processes plus a master lease, which is like a promissory note that the processes will not elect a new master within that particular time interval.

The processes occasionally renew the master lease provided the master sustains its mastership by winning a quorum of the vote. The master is solely responsible for carrying out the read and write request of the client and guarantees that other processes' state is updated, especially when it carries out a write request. The client request is term concluded as soon as most processes acknowledge the write operation. Figure 2.9 conveys the communication patterns between a client and the chubby processes. The client initially sends a request looking for the location of the master (Phase 1). If a non-master process receives the request, it replies to the client with the actual location of the master (Phase 2). Once the location of the master is known, the client sends a read/write request to the master (Phase 3); the master accepts the request and broadcasts it to every process (Phase 4) before executing the request internally and sending a reply to the client (Phase 5).

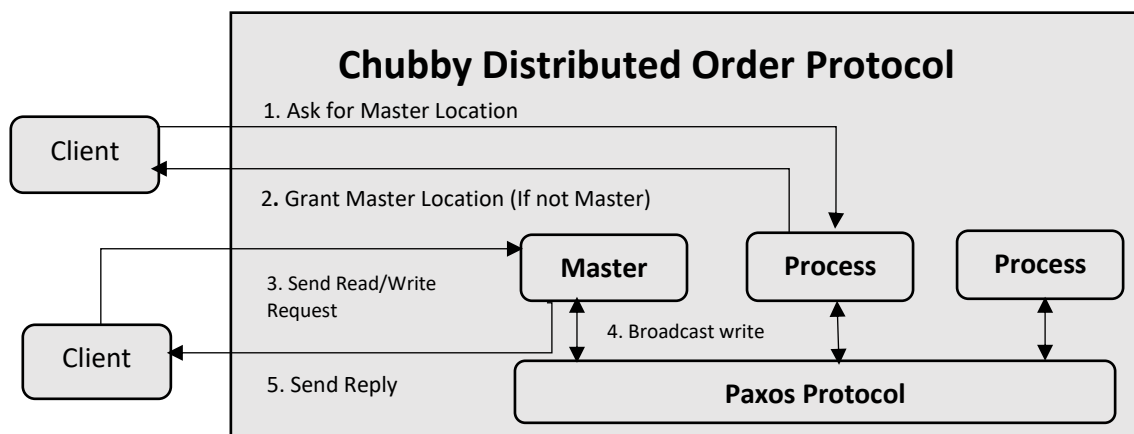


Figure 2.9 Chubby's Read/Write in Master/Process Scheme [64]

However, one of the demerits of Chubby is its continuous use of a single master in executing read/write requests, which will hinder the system's throughput when client requests increase. Chubby has become Google primary name service; it has been utilised by several Google applications. For instance, Google File System [118] used Chubby for nominating the GFS master server, Bigtable [119] used Chubby for nominating the master process and enabling the master process to know the processes under its supervision, and MapReduce [12]. During Chubby's development, the significant considerations were reliability, availability, and the choice of semantics that were not demanding to understand.

2.6.5 Zookeeper

Zookeeper [40, 120] is a protocol for implementing the coordination of operations in distributed computing and applications supported by Yahoo and Apache Software Foundation.

The principal purpose of a zookeeper is to deliver an easy and high-performance kernel for developing more compound coordination primitives at the client. It was designed to provide services for distinct coordination needs in large-scale distributed systems, including configuration maintenance, leader election, group membership, replicated state storage, failure detection, and distributed synchronisation. Zookeeper is designed as a highly scalable, reliable, and robust centralized service for enforcing coordination in distributed applications. It offers system developers with a simple interface that allows them to directly integrate centralized coordination services into their applications [121]. It permits software developers to pay attention to the vital business logic of their applications and depend wholly on the zookeeper to bring the desired coordination service, thus relieving developers of the problem of developing the coordination service from the beginning. Zookeeper is designed to operate a wait-free data object and resemble a file system like Chubby [41]. Although, unlike Chubby, Zookeeper does not implement a lock service mechanism. Many applications and companies use Zookeeper for their operations. For example, Yahoo uses Zookeeper as a fetching service for its crawler, which helps in master failure recovery and leader election; Kafka uses it for coordination services, Facebook uses it for data record storage, failover and service detection, and Rackspace uses it for email management [122].

ZooKeeper assumes crash-stop and crash-recovery failures using a quorum-based approach. ZooKeeper group size typically consist of $N=2f+1$ nodes. To maintain consistency and availability, ZooKeeper requires a majority (quorum) of nodes to be operational while a minority of nodes f can crash. When f node crashes, the remaining majority can still make progress and elect a new leader if needed. ZooKeeper achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by employing fast reads with watches, both served by local replicas.

2.7 Background

2.7.1 LCR Order Protocol

This section introduces LCR [2] as the first throughput optimal uniform total order broadcast protocol. The protocol was designed for use in a ring-based network for a set of a homogenous cluster of processes. The remainder of this section is structured as follows: First, we introduce the rationale behind the LCR protocol before detailing the protocol's requirements and assumptions. Finally, we discuss the potential limitations of LCR protocol in the context of total order service and propose the need for a new total order unicast solution.

Rationale

The traditional ring-based uniform total order (UTO) broadcast protocol, detailed in Section 2.5.1, passes a token among the processes to grant them the privilege of broadcasting a message within a ring topology. This arrangement is potent in achieving high throughput only in scenarios where either one process is broadcasting to all process replicas (1-to- N) or when all processes are simultaneously broadcasting to each other (N – to – N). However, it results in lower throughputs in alternative scenarios. For instance, in a cluster with $N = 4$ processes, if two processes attempt to transmit messages simultaneously, the throughput diminishes due to the token being passed sequentially between the transmitting processes.

However, the LCR protocol [2] which is an extension of the FSR protocol [1], is an ordering protocol that guarantees that processes within a cluster deliver received messages in the same order. It combines the ring topology for high throughput dissemination, a vector timestamp, VT, and a fixed last process for ordering concurrent messages sent within a ring-base network. Thus, a process can send a message as soon it becomes available instead of the process waiting for a token before sending messages, as used in traditional ring-based protocols. This improves throughput while maintaining a reasonable latency.

System Model

LCR protocol was built around a small cluster of N sets of homogeneous processes, $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ interconnected by a local area network in a ring-based network. Each process creates a TCP connection to a single process and maintains this connection during the entire execution of the protocol unless the process fails as shown in Figure 2.10.

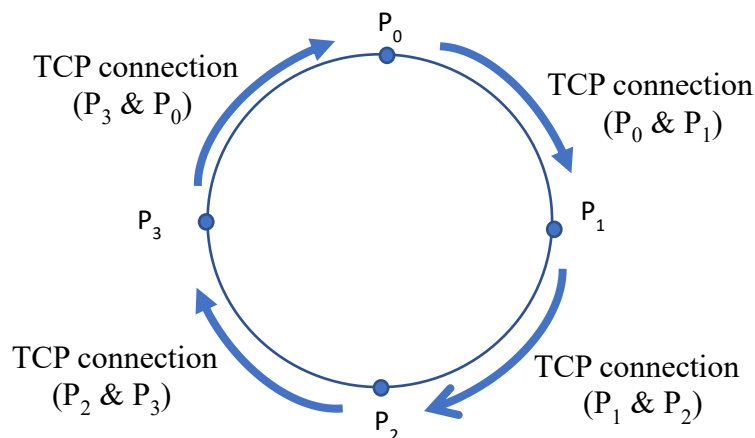


Figure 2.10 LCR System Model [2]

For instance, suppose we consider a set of $N = 4$ processes (see Figure 2.10), P_0 maintains a TCP connection with P_1 , P_1 with P_2 , P_2 with P_3 , and P_3 with P_0 . This means that P_0 sends messages only to P_1 and received messages from P_3 ; P_1 sends messages only to P_2 and received messages from P_0 ; P_2 sends messages only to P_3 and received messages from P_1 ; P_3 sends messages only to P_0 and received messages from P_2 .

LCR assumes that a process can crash but not the TCP connection. Thus, if an operative process, P_1 finds the TCP connection fails, then P_1 assumes that the process, P_0 on the other side of the TCP connection has failed: P_1 can detect that P_0 has failed, which means that the process-to-process communication in LCR is reliable. Therefore, LCR implemented a perfect failure detector abstraction to enable processes to detect process failures reliably.

Performance Model

The performance model of an LCR defines the behaviour and efficiency of the LCR protocol and is hinged on the definition of a round. A round is defined as follows:

- In a given round k , every process P_i sends at most one message, $m(i, k)$, to one or more destinations; i is the identity of the process sending m , while k is the number of jumps m has made within the ring network.
- Also, it receives at most one message in the same round.

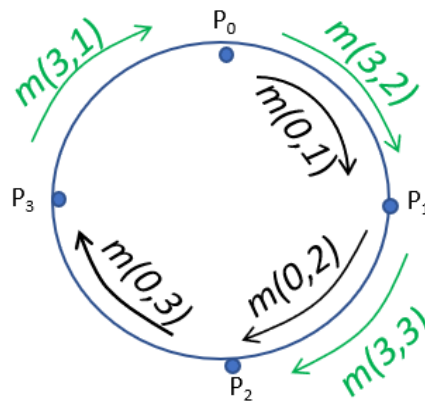


Figure 2.11 Performance Model [2]

As illustrated in Figure 2.11:

At round $k = 1$, P_0 sends its message, $m(0,1)$, to P_1 and receive $m(3,1)$ sent by P_3 ;

At round $k = 2$, P_0 forwards the message, $m(3,2)$ received from P_3 to P_1 while P_1 forwards the message, $m(0,2)$ received from P_0 to P_2 ;

At round $k = 3$, P_1 forwards the message, $m(3, 3)$ received from P_0 to P_2 while P_2 forwards the message, $m(0, 3)$, received from P_1 to P_3 .

Hence, in a cluster of $N = 4$ processes, it takes at most 3 rounds for every process to receive the messages sent by any process P_i .

LCR Protocol Overview

Let $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ denotes the set of N processes implemented in a ring network as described earlier. P_0 is called the “first process in the ring”, while P_{N-1} is the “last process in the ring”. Every process P_i within the ring cluster, has a predecessor ($PRED_i$) and successor (SUC_i). We define $PRED_i$ and SUC_i as follows: for any process P_i , $0 \leq i \leq N - 1$

- $SUC_i = P_{i+1}$ or $SUC_i = P_0$, if $i = N - 1$, and
- $PRED_i = P_{i-1}$ or $PRED_i = P_{N-1}$, if $i = 0$

In LCR, messages are sent only in the successor direction and received in the predecessor direction; P_i receives messages from $PRED_i$ and send messages to SUC_i in the ring (see Figure 2.12).

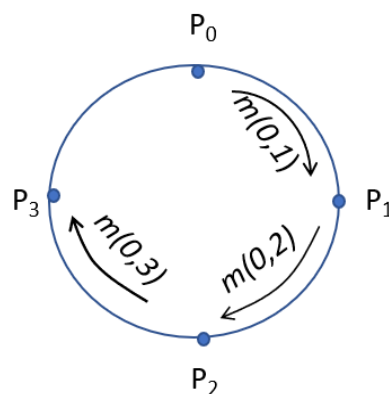


Figure 2.12 LCR Communication Design [2]

Figure 2.12 shows the communication design of the LCR protocol required for processes to communicate with each other. As noted earlier, in round $k = 1$, P_0 sends a message, $m(0, 1)$, to its SUC_0 which was received in the same round. In round $k = 2$, P_1 forwards a copy of the received message, $m(0, 2)$, to its SUC_1 which is also received in the same round. Finally, P_2 forwards a copy of the message, $m(0, 3)$, to its SUC_2 and SUC_2 received the message in the same round.

Furthermore, every process P_i , maintains a local vector clock $VC_i = vc_{k,k=0,\dots,N-1}$ used for vector timestamping the messages to be transmitted and a fixed last process for ordering

concurrent messages. $VC_i[i]$ is the number of messages sent by the process P_i while $VC_j[i]$ is the number of messages sent by the process P_i that P_j has received. For instance, if P_j has received five messages from P_i then $VC_j[i] = 5$, and $VC_i[i] = 5$.

Send Rule:

The process P_i increases its $VC_i[i]$ by 1, vector timestamp (vt) m_i with its $VC_i[i]$, $m_{i_vt} = VC_i[i]$, before sending m_i to its successor. For instance, if initially $VC_0[0] = [0000]$, and P_0 wants to send a message, m_0 , it increases its $VC_0[0]$ i.e., $VC_0[0] = [1000]$, then $m_{0_vt} = [1000]$, before sending m_0 .

Receive Rule:

When P_j receives m_i , it increases its $VC_j[i]$ by 1. For instance, if initially $VC_1[0] = [0000]$, when P_1 receives m_0 , it increases its $VC_1[0]$ i.e., $VC_1[0] = [1000]$, and $m_{0_vt} = [1000]$. The vector timestamp, vt, of the message, remains the same throughout the entire execution, substantiating the reason m_{0_vt} is the same at P_1 .

If P_j sends m_j after processing m_i it will timestamp m_j to show that it received m_i before sending m_j . For simplicity, we will henceforth represent $m(i, k)$ as m_i and implicitly track the number of rounds m_i has completed in the ring.

LCR Protocol Guarantees

LCR protocol is a uniform total order broadcast (UTO-broadcast) that combines the two primitives, `utoBroadcast` and `utoDeliver`, to ensure that messages are delivered reliably and in the same order at all destinations, P_i . Processes within the LCR cluster must maintain the following guarantees to ensure that a broadcast is delivered in total order at the various process destinations [18, 57, 123]:

1. **Validity:** if a correct process, P_i `utoBroadcasts` a message m , then P_i eventually `utoDelivers` m .
2. **Integrity:** For any message m , any correct process P_j `utoDelivers` m at most once, and only if m was previously `utoBroadcast` by some correct process P_i .
3. **Uniform Agreement:** If any operative process P_i `utoDelivers` any message m , then every correct process `utoDelivers` m .
4. **Total Order:** For any two messages, m_1 and m_2 , if any process P_i `utoDelivers` m_1 without having delivered m_2 , then no process P_j `utoDelivers` m_2 before m_1 . For

example, if the process P_i delivers m_1, m_2, m_3, m_4 and then crashes; no process P_j will deliver m_5 , before m_1, m_2, m_3 , and m_4 .

Total Order in LCR

As explained in the preceding, the sending of a message requires a process to send the message to its successor and forward it upon reception until every process within the cluster has received the message. In LCR protocol, to ensure that processes receive messages in the same order, the total order of messages is defined as the order according to which messages are received by the last process in the ring, i.e., process P_{N-1} . This definition utilizes the last process rule, such that if process P_i and P_j send m_i and m_j respectively, assuming $i < j$, which means that j is closer to the last process, then m_j will be ordered before m_i . In the last process rule, messages are usually ordered according to the sender; whichever sender is closer to the last process takes priority, but this applies only to concurrent messages. There are two possibilities to see if a message is not concurrent: Two messages are either concurrent or have a “happened before” relation. Therefore, in normal circumstances (using the last process), m_j will be ordered before m_i except in one special case.

Special Case:

if m_j was sent after receiving m_i (m_i happened before m_j), then m_i is ordered before m_j

Special Case Rule:

if m_i_vt and m_j_vt are the vector timestamps of m_i and m_j , then message m_i is ordered before m_j if and only if $m_i_vt[i] \leq m_j_vt[i]$ when $i < j$ and $m_i_vt[i] < m_j_vt[i]$ when $i = j$. This rule implies that if the sending of m_i happened before the sending of m_j , m_i will be ordered before m_j provided that $m_i_vt[i] \leq m_j_vt[i]$ and $i < j$. Also, if $m_i_vt[i] < m_j_vt[i]$ and $i = j$ then, m_i will be ordered before m_j . Thus, when m_i and m_j are sent concurrently if m_j originates from a process that is closer to the last process, which in this case is valid, m_j will be ordered before m_i else m_i will be ordered before m_j . However, our study considered only the concurrent message transmissions within the cluster. Thus, Figure 2.13 shows the concurrent unicast from the process P_1 and P_3 , respectively.

Initially, every process P_i has $VC_i = [0000]$, if P_1 and P_3 send messages, m_1 and m_3 concurrently to their successor using the special case rule, the following scenario happens: There are two messages m_1 and m_3 , while m_1 comes from the smaller process origin (P_1). Therefore m_1 will determine the order as follows: if $m_{1_vt}[1] < m_{3_vt}[1]$ and $P_1 < P_3$ then m_1 is ordered before m_3 otherwise, m_3 is ordered before m_1 .

At round $k = 1$, as shown in Figure 2.13a.

P_1 and P_3 sends **green** (m_1) and **red** (m_3) messages concurrently. P_3 increments its VC_3 i.e., $VC_3 = [0001]$ and vector timestamp, vt, the message m_3 , $m_{3_vt} = [0001]$ before sending m_3 to its SUC_3 . Also, P_1 increments its VC_1 i.e., $VC_1 = [0100]$ and vector timestamp m_1 , $m_{1_vt} = [0100]$ before sending m_1 to SUC_1 . The SUC_1 and SUC_3 received the messages within the same round. When P_0 receives m_3 it sets $VC_0 = [0001]$, and $m_{3_vt} = [0001]$; the change in the fourth index of VC_0 indicates that it has received a message from P_3 . Also, When P_2 receives m_1 it sets $VC_2 = [0100]$, and $m_{1_vt} = [0100]$; the change in the second index of VC_2 indicates that it has received a message from P_1 .

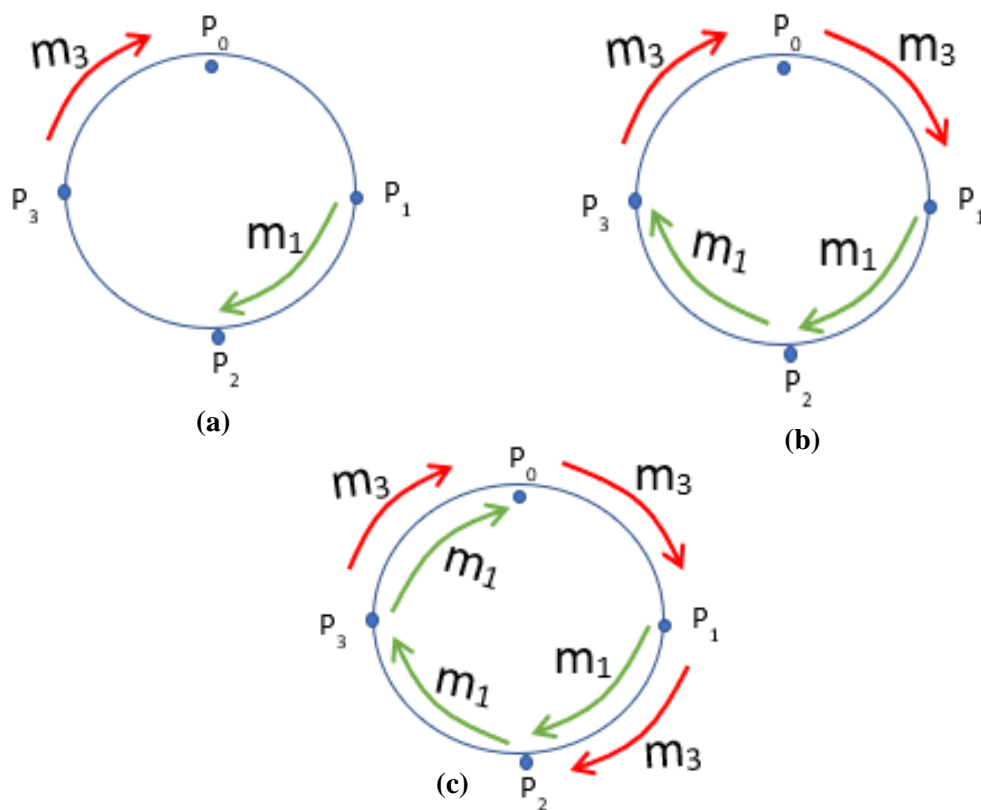


Figure 2.13 Concurrent unicast ordering in LCR [2]

At round $k = 2$, as shown in Figure 2.13b.

P_0 and P_2 forwards **red** (m_3) and **green** (m_1) concurrently to the successors. The receiving process performs the same operation as explained in round $k = 1$. It increases its local vector clock by 1 and forwards a copy of the message while the vector timestamp, vt of the message remains the same.

At round $k = 3$, as shown in Figure 2.13c.

When P_2 receives m_3 it sets $VC_2 = [0101]$, $m_{3_vt} = [0001]$. The 1's at the second and fourth index of the VC_2 is a suggestion that P_2 has received one message each from P_1 and P_3 ; since $m_{1_vt}[1] < m_{3_vt}[1]$ does not satisfy the special case rule and $P_1 < P_3$. Therefore P_2 cannot order m_1 before m_3 ; hence P_2 orders m_3 before m_1 .

Also, when P_0 receives m_1 it sets $VC_0 = [0101]$, $m_{1_vt} = [0100]$. The 1's at the second and fourth index of the VC_0 is a suggestion that P_0 has received two messages: one message from P_1 and one message from P_3 ; since $m_{1_vt}[1] < m_{3_vt}[1]$ does not satisfy the special case rule and $P_1 < P_3$. Therefore P_0 cannot order m_1 before m_3 ; hence P_0 orders m_3 before m_1 . The LCR protocol will ensure that the exact order of m_3 before m_1 is replicated at P_1 and P_3 .

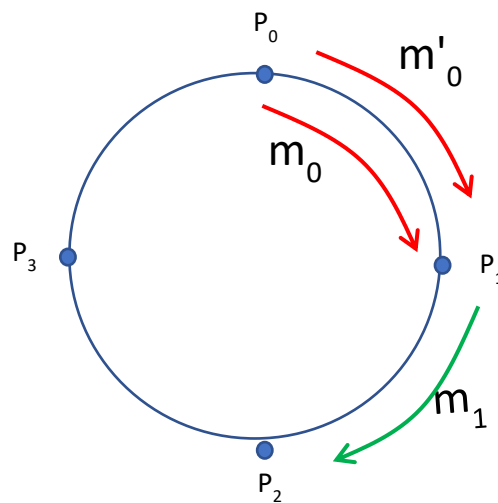


Figure 2.14 Complex Scenario

Figure 2.14 shows a case where the process P_1 received multiple messages and simultaneously sent its own message after receiving the messages.

Case 1: P_1 receives two messages m_0 and m'_0 (see Figure 2.14)

When P_1 receives m_0 it sets $VC_1 = [1000]$, and $m_{0_vt} = [1000]$; the change in the first index indicates that it has received a message from P_0 . Also, When P_1 receives

m'_0 it sets $VC_1 = [2000]$, and $m'_0_vt = [2000]$; the change in the first index indicates that it has received two messages from the same process P_0 . Then P_1 will order m_0 before m'_0 since $m_0_vt[0] < m'_0_vt[0]$ satisfies the condition and $i = j$ ($P_0 = P_0$).

Case 2: P_1 sends own message after receiving m_0 and m'_0 (see Figure 2.14)

If P_1 now desires to send its own message m_1 then it sets $VC_1 = [2100]$, and $m_1_vt = [2100]$. Thus, P_1 performs a comparison of the message vector timestamps, m_vt , in its internal buffer to determine the order of all the messages as follows:

- a. It compares $m_1_vt[1]$ and $m_0_vt[1]$; since $m_1_vt[1] < m_0_vt[1]$ does not satisfy the rule; this means that m_0 happened before m_1 then P_1 orders m_0 before m_1 .
- b. It compares $m_1_vt[1]$ and $m'_0_vt[1]$; since $m_1_vt[1] < m'_0_vt[1]$ does not satisfy the rule; this means that m'_0 happened before m_1 then P_1 orders m'_0 before m_1 .
- c. Since P_1 had already ordered m_0 before m'_0 in its internal buffer, then P_1 finally, order the messages as follows: m_0 , m'_0 , and m_1 in its pending list.

We noted that when there are k messages already in the internal queue or list of any process if a new message arrives, processes will perform k comparisons to determine the new order of the messages, as explained in case 2 above.

Failure-Free Behaviour

In this section, we explained the behaviour of LCR protocol in the absence of process crashes; when a process P_i desires to send a message, it sends the message m_i to its SUC_i in the ring network. The message m_i is then forwarded until it reaches the $PRED_i$, the last process in the ring. Processes forward messages in the order in which they receive them. For example, if P_i received m_1 before m_2 then P_i must forward m_1 before it forwards m_2 . Each process guarantees total order delivery by ensuring that before delivering any message m_i , that it will not receive any successive message that is ordered before m_i .

Moreover, each process ensures before delivering m_i that every process within the cluster has received m_i , that is, m_i is stable. This maintains uniform delivery among the processes. This guarantee relies on a local list, called pending, used by every process to store the messages before they are delivered. The messages in the pending list of any process are ordered based on the last process order definition.

Total Order Delivery

The problem with process-to-process message communication is that every process must deliver the messages in the same order. When a process P_i sends a message m_i , it stores a copy of the message in its pending list and sends a copy of m_i to the successor. The successor, upon receiving m_i forwards it to its successor. This continues until m_i is received by every process within the group. When the last process, P_{N-1} , that is the $PRED_i$ receives the message m_i , it knows that m_i is stable and attempts to deliver m_i if it is at the head of the list. It then communicates the stability of m_i by generating an acknowledgement message, $ACK(m_i)$, and sending it along the ring network. The $ACK(m_i)$ is then successively forwarded until it reaches the predecessor of the process that sent it. When any process receives the $ACK(m_i)$, it knows that m_i is stable and that it has already received all messages that were ordered before m_i . Once a message becomes stable, it can be delivered as soon as it becomes the first message at the head of the pending list.

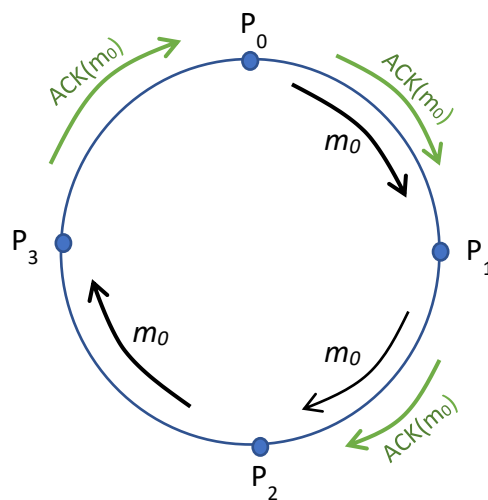


Figure 2.15 Total Order Delivery [2]

Figure 2.15 depicts a single message multicast's total order delivery arrangement in LCR protocol.

At round $k = 1$

P_0 sends a message m_0 to the SUC_0 (P_1) after storing a copy of m_0 in its pending list. Also, m_0 is received by P_1 in the same round.

At round $k = 2$

P_1 forwards a copy of m_0 to its SUC_1 (P_2) after storing a copy in its pending list. P_2 also receive m_0 in the same round.

At round $k = 3$

P_2 forwards a copy of m_0 to its SUC_2 (P_3) after storing a copy in its pending list. When P_3 receive m_0 ; it knows that m_0 is stable and that every process within the group has received m_0 . P_3 then attempts to deliver m_0 if it is the first message on the head of its pending list.

At round $k = 4$

P_3 then generates an $ACK(m_0)$ message and sends it to its SUC_3 (P_0). When P_0 receive the $ACK(m_0)$ message; it knows that m_0 , which originated from itself, is stable, and then it attempts to deliver m_0 if m_0 is the first message at the head of its pending list.

At round $k = 5$

P_0 forwards the $ACK(m_0)$ to its SUC_0 , that is, P_1 . When P_1 received the $ACK(m_0)$ message, it knows that m_0 , which originated from P_0 is stable, and then attempts to deliver m_0 if m_0 is the first message at the head of its pending list.

At round $k = 6$

P_1 forwards the $ACK(m_0)$ to its SUC_1 , that is, P_2 . When P_2 received the $ACK(m_0)$ message, it knows that m_0 , which originated from P_0 is stable, and then attempts to deliver m_0 if m_0 is the first message at the head of its pending list. P_2 stops forwarding the $ACK(m_0)$ message and discard it since it is the $PRED_3$ that sent the $ACK(m_0)$.

Therefore, in a failure-free scenario of the LCR protocol and for $N = 4$ process cluster, it takes at most six rounds for a unicast to be delivered by all the processes within the system cluster.

LCR Algorithm Principles

This section discussed the protocol principles (see Figure 2.15) and the messages' data structures.

Principles:

1. Any process P_i generate, vector timestamp message m with its local vector clock, VC_i and sends m to its SUC_i with $m.origin = P_i$
2. When P_j receives m whose message origin is not its successor, $m.origin \neq SUC_j$, it forwards a copy of m to the SUC_j but,

3. When P_j receives m whose message origin is its successor $m.origin = SUC_j$, then P_j knows that m is stable and attempts to deliver m .
4. P_j makes a stability declaration using an $ACK(m)$, which it sends along the ring network.
5. The $ACK(m)$ is forwarded along the ring until the $PRED_j$ receives it and then stops forwarding.
6. Any process P_k that receives the $ACK(m)$ knows that m is stable and attempts to deliver m in total order.

Data Structures:

Here, we discuss the data structures for any process P_i , the two messages transmitted within the protocol: m being the data message being exchange among the processes and $ACK(m)$ being the acknowledgement of m used to communicate the stability of m to other processes.

Each process P_i has at least the following data structures:

1. **Vector clock** (VC): This field holds the value used for vector timestamping messages before it is transmitted. The i th index $VC[i]$ of the sending process is increase by 1 before the message m is transmitted; also, when process P_i receive m , the i th index $VC[j]$ of the sending process P_j within the VC is incremented by 1. Thus, the $VC[i]$ also indicates the number of messages P_i has sent and received at any time interval.
2. **Pending List** ($pendingList_i$): This holds the incoming messages and own messages for any process P_i . All the messages in the pending list are ordered according to the order they were received.
3. **Garbage Collection Queue** (GCQ_i): This holds the delivered messages for any process P_i .

Each message m must have at least the following data structures:

1. **Message field** (m): this indicates the type of message being sent by the process P_i .
2. **Process Identity** (P_i): this identifies the sending or receiving process within the system
3. **Message Vector timestamp** (m_vt): this holds the value of the VC at the time the m was sent, that is, $m_vt = VC[i]$; it remains the same through out the transmission of m .
4. **Stability field**: a Boolean field that shows whether a message is stable. A message is stable when the stability field is true, but when it is false, the message is not stable.

Each $ACK(m)$ message must have at least the following data structures:

1. **ACK field** ($Ack(m)$), which indicates the type of message.
2. **Process Identity** (P_k): this identifies the process that generates and sends $ack(m)$ message within the system.
3. **ACK Vector Timestamp** ($Ack(m)_{vt}$) holds the vector timestamp when a process P_i sends the ack message.

Benefits of LCR protocol

LCR protocol was implemented using a combination of a vector clock and a ring topology; the ring topology ensures that each process always sends messages to the same process, thus avoiding any possible collisions. To eliminate bottlenecks observed in the FSR protocol, messages in LCR are sequenced using a vector timestamp, the sending and forwarding of messages along the ring is simpler compared to FSR and a fixed last process is used for ordering concurrent messages. Additionally, the vector timestamps enhance the reliability and efficiency of distributed system applications, which are essential for preserving consistency, resolving conflicts (using the causality of events to compare events when multiple processes update shared data concurrently), and offering a logical framework for event ordering.

Throughput

Throughput measures the number of messages multicasts that the processes within the system cluster can complete per unit of time; that is, it captures the average number of completed multicasts per unit of time. A completed message broadcast implies that all processes completed the delivery of the messages m within a specific time. When there is a single message multicast, the throughput of the completed broadcast at each process is 1. Nevertheless, when a multitude of messages, for instance, 10,000, are transmitted within a specified time period, ascertaining the number of accomplished multicasts at each process within that temporal scope becomes intricate. Therefore, this underscores the necessity for a discrete event simulation to determine the average maximum throughput within a specific simulation time frame.

Latency

In the LCR protocol, latency is defined theoretically as the number of rounds that are necessary from the initial multicast of message m until every process P_i delivers m . Consequently, the latency, L , for a multicast to be delivered in total order by all nodes is $L = 2N - 2$ rounds. Hence, in a cluster of $N = 4$ processes, as illustrated in Figure 2.15, it takes

six rounds for every process to deliver m . In addition, latency measures the time needed to complete a single message multicast without contention. However, in a scenario where multiple concurrent messages, say 10000 messages are sent within a specific time interval, it becomes a complicated case to determine the latency as we cannot tell what the latency is by just using $L = 2N - 2$ relation. This justifies the need to perform a discrete event simulation on LCR protocol to determine the average maximum latencies for such scenarios of large concurrent message multicasts. The pseudocode of the LCR protocol is illustrated in Figure 2.16.

Procedures executed by any process p_i

```

1: procedure initialize(initial_view)
2:   pending $i$   $\leftarrow \emptyset$                                      {pending list}
3:    $C[1 \dots n] \leftarrow \{0, \dots, 0\}$                          {local vector clock}
4:   view  $\leftarrow$  initial_view

5: procedure utoBroadcast( $m$ )
6:    $C[i] \leftarrow C[i] + 1$ 
7:   pending  $\leftarrow$  pending  $\cup [m, p_i, C, \perp]$ 
8:   Rsend  $\langle m, p_i, C \rangle$  to successor( $p_i, view$ )                {broadcast a message}

9: upon Rreceive  $\langle m, p_j, C_m \rangle$  do
10:  if  $C_m[j] > C[j]$  then
11:    if  $p_i \neq predecessor(p_j, view)$  then
12:      Rsend  $\langle m, p_j, C_m \rangle$  to successor( $p_i, view$ )            {forward the message}
13:      pending  $\leftarrow$  pending  $\cup [m, p_j, C_m, \perp]$ 
14:    else
15:      pending  $\leftarrow$  pending  $\cup [m, p_j, C_m, stable]$           {m is stable}
16:      Rsend  $\langle ACK, p_j, C_m \rangle$  to successor( $p_i, view$ )          {send an ACK}
17:      tryDeliver()
18:       $C[j] \leftarrow C[j] + 1$                                     {update local vector clock}

19: upon Rreceive  $\langle ACK, p_j, C_m \rangle$  do
20:  if  $p_i \neq predecessor(predecessor(p_j), view)$  then
21:    pending[ $C_m$ ]  $\leftarrow [*, *, *, stable]$                       {m is stable}
22:    Rsend  $\langle ACK, p_j, C_m \rangle$  to successor( $p_i, view$ )            {forward the ACK}
23:    tryDeliver()

24: procedure tryDeliver()
25:  while pending.first =  $[m, p_k, C_m, stable]$  do
26:    utoDeliver( $m$ )                                                {deliver a message}
27:    pending  $\leftarrow$  pending -  $[m, p_k, C_m, stable]$ 

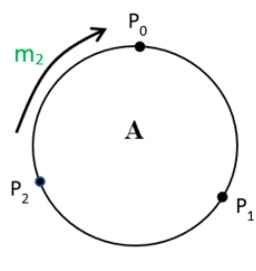
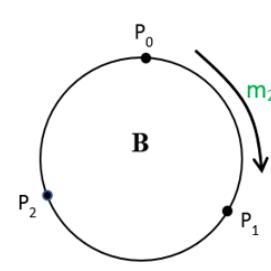
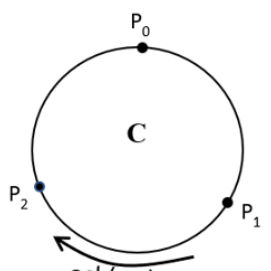
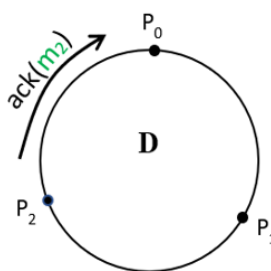
```

Figure 2.16 Pseudocode of LCR Protocol [2]

LCR Operations

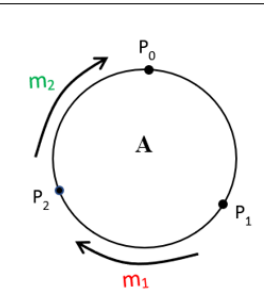
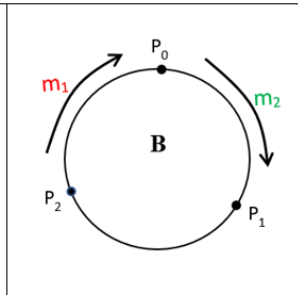
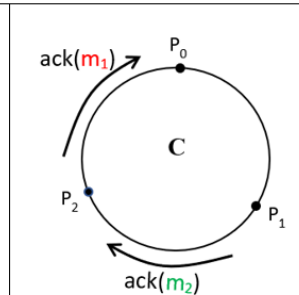
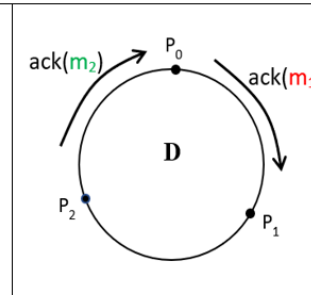
In this section, we used a cluster of $N = 3$ processes to illustrate the complete operation of LCR to provide simplified workings of the LCR protocol. We examined scenarios involving a single multicast, and two concurrently sent multicast messages at the initial start of the LCR protocol (with unequal timestamps).

Table 2-3 LCR Operation with Single Multicast

																																																			
<p>After 1 hop</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀=[001]</td><td></td></tr><tr><td>P₁</td><td></td></tr><tr><td>VC₁=[000]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₂=[001]</td><td></td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ =[001]		P ₁		VC ₁ =[000]		P ₂	(m ₂ , 2, [001])	VC ₂ =[001]		<p>After 2 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀=[001]</td><td></td></tr><tr><td>P₁</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₁=[001]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₂=[001]</td><td></td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ =[001]		P ₁	(m ₂ , 2, [001], stable)	VC ₁ =[001]		P ₂	(m ₂ , 2, [001])	VC ₂ =[001]		<p>After 3 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀=[001]</td><td></td></tr><tr><td>P₁</td><td></td></tr><tr><td>VC₁=[001]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₂=[001]</td><td></td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ =[001]		P ₁		VC ₁ =[001]		P ₂	(m ₂ , 2, [001], stable)	VC ₂ =[001]		<p>After 4 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₀=[001]</td><td></td></tr><tr><td>P₁</td><td></td></tr><tr><td>VC₁=[001]</td><td></td></tr><tr><td>P₂</td><td></td></tr><tr><td>VC₂=[001]</td><td></td></tr></table>	P ₀	(m ₂ , 2, [001], stable)	VC ₀ =[001]		P ₁		VC ₁ =[001]		P ₂		VC ₂ =[001]	
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ =[001]																																																			
P ₁																																																			
VC ₁ =[000]																																																			
P ₂	(m ₂ , 2, [001])																																																		
VC ₂ =[001]																																																			
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ =[001]																																																			
P ₁	(m ₂ , 2, [001], stable)																																																		
VC ₁ =[001]																																																			
P ₂	(m ₂ , 2, [001])																																																		
VC ₂ =[001]																																																			
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ =[001]																																																			
P ₁																																																			
VC ₁ =[001]																																																			
P ₂	(m ₂ , 2, [001], stable)																																																		
VC ₂ =[001]																																																			
P ₀	(m ₂ , 2, [001], stable)																																																		
VC ₀ =[001]																																																			
P ₁																																																			
VC ₁ =[001]																																																			
P ₂																																																			
VC ₂ =[001]																																																			

From Table 2-3, the process P_2 sends a green message m_2 to its successor, P_0 , after incrementing its vector clock and using it to vector timestamp m_2 denoted as $m_{2_vt} = [001]$. After 1 hop, P_2 retains a copy of the transmitted message, m_2 , in its pending list, while simultaneously, process P_0 receives the message and stores it in its pending list. After 2 hops, m_2 has made a complete cycle at P_1 making m_2 stable and is TO delivered by P_1 . Subsequently, P_1 prepares and sends $ack(m_2)$ to P_2 , which P_2 receives after the third hop, confirming the stability of m_2 and allowing for its TO delivery by the process P_2 . Thereafter, P_2 forwards a copy of $ack(m_2)$ to P_0 . After receiving the $ack(m_2)$ by P_0 in the fourth hop, P_0 knows that m_2 is stable and also TO deliver m_2 . Therefore, for the $N = 3$ process cluster and using a single multicast, the earliest TO delivery of LCR occurs after two hops and the latest TO delivery occurs after four hops.

Table 2-4 LCR Operation with Concurrent Multicast

																																																			
<p>After 1 hop</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀= [001]</td><td></td></tr><tr><td>P₁</td><td>(m₁, 1, [010])</td></tr><tr><td>VC₁= [010]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₂= [011]</td><td>(m₁, 1, [010])</td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ = [001]		P ₁	(m ₁ , 1, [010])	VC ₁ = [010]		P ₂	(m ₂ , 2, [001])	VC ₂ = [011]	(m ₁ , 1, [010])	<p>After 2 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀= [011]</td><td>(m₁, 1, [010], stable)</td></tr><tr><td>P₁</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₁= [011]</td><td>(m₁, 1, [010])</td></tr><tr><td>P₂</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₂= [011]</td><td>(m₁, 1, [010])</td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ = [011]	(m ₁ , 1, [010], stable)	P ₁	(m ₂ , 2, [001], stable)	VC ₁ = [011]	(m ₁ , 1, [010])	P ₂	(m ₂ , 2, [001])	VC ₂ = [011]	(m ₁ , 1, [010])	<p>After 3 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001])</td></tr><tr><td>VC₀= [011]</td><td>(m₁, 1, [010], stable)</td></tr><tr><td>P₁</td><td>(m₁, 1, [010], stable)</td></tr><tr><td>VC₁= [011]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₂= [011]</td><td>(m₁, 1, [010])</td></tr></table>	P ₀	(m ₂ , 2, [001])	VC ₀ = [011]	(m ₁ , 1, [010], stable)	P ₁	(m ₁ , 1, [010], stable)	VC ₁ = [011]		P ₂	(m ₂ , 2, [001], stable)	VC ₂ = [011]	(m ₁ , 1, [010])	<p>After 4 hops</p> <table><tr><td>P₀</td><td>(m₂, 2, [001], stable)</td></tr><tr><td>VC₀= [011]</td><td>(m₁, 1, [010], stable)</td></tr><tr><td>P₁</td><td></td></tr><tr><td>VC₁= [011]</td><td></td></tr><tr><td>P₂</td><td>(m₁, 1, [010], stable)</td></tr><tr><td>VC₂= [011]</td><td></td></tr></table>	P ₀	(m ₂ , 2, [001], stable)	VC ₀ = [011]	(m ₁ , 1, [010], stable)	P ₁		VC ₁ = [011]		P ₂	(m ₁ , 1, [010], stable)	VC ₂ = [011]	
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ = [001]																																																			
P ₁	(m ₁ , 1, [010])																																																		
VC ₁ = [010]																																																			
P ₂	(m ₂ , 2, [001])																																																		
VC ₂ = [011]	(m ₁ , 1, [010])																																																		
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ = [011]	(m ₁ , 1, [010], stable)																																																		
P ₁	(m ₂ , 2, [001], stable)																																																		
VC ₁ = [011]	(m ₁ , 1, [010])																																																		
P ₂	(m ₂ , 2, [001])																																																		
VC ₂ = [011]	(m ₁ , 1, [010])																																																		
P ₀	(m ₂ , 2, [001])																																																		
VC ₀ = [011]	(m ₁ , 1, [010], stable)																																																		
P ₁	(m ₁ , 1, [010], stable)																																																		
VC ₁ = [011]																																																			
P ₂	(m ₂ , 2, [001], stable)																																																		
VC ₂ = [011]	(m ₁ , 1, [010])																																																		
P ₀	(m ₂ , 2, [001], stable)																																																		
VC ₀ = [011]	(m ₁ , 1, [010], stable)																																																		
P ₁																																																			
VC ₁ = [011]																																																			
P ₂	(m ₁ , 1, [010], stable)																																																		
VC ₂ = [011]																																																			

From Table 2-4, the processes P_2 and P_1 send green and red messages, m_2 and m_1 concurrently. For the LCR protocol, just before sending the messages, P_2 and P_1 vector timestamp m_2 and m_1 after incrementing their local vector clock index as, $m_{2_vt} = [001]$ and $m_{1_vt} = [010]$ respectively. After 1 hop, P_2 has 2 messages in its pending list since it also receives m_1 from P_1 in the same hop. In LCR, if m_i_vt and m_j_vt are the vector timestamps of m_i and m_j , message m_i is ordered before m_j if and only if $m_i_vt[i] \leq m_j_vt[i]$ when $i < j$ and $m_i_vt[i] < m_j_vt[i]$ when $i = j$.

Therefore, at P_2 since the message m_1 comes from a smaller process origin, m_1 will determine the order as follows: if $m_{1_vt}[1] < m_{2_vt}[1]$ and $P_1 < P_2$ then m_1 is ordered before m_2 otherwise m_2 is ordered before m_1 . Consequently, P_2 orders m_2 before m_1 since $m_{1_vt}[1] < m_{2_vt}[1]$ does not hold. This reinforces the idea that LCR orders concurrent messages using a fixed last process. After 2 hops, m_2 has made a full cycle at P_1 while m_1 made a complete cycle at P_0 . However, P_0 cannot TO deliver m_1 since m_2 is not stable at P_0 even though P_0 knows that m_1 is stable. if $m_{1_vt}[1]$ is equal to the $m_{2_vt}[1]$, P_0 would have deduced that both m_2 and m_1 are stable simultaneously and TO deliver both.

Consequently, P_0 cannot determine that m_2 is stable even though both messages were sent concurrently. So, after 2 hops, P_1 only TO delivers m_2 . Subsequently, P_1 and P_0 prepare and send $ack(m_2)$ and $ack(m_1)$ to their respective successors. After the third hop, P_2 and P_1 TO deliver m_2 and m_1 simultaneously since they are stable (because of the $ack(m_2)$ and $ack(m_1)$)

and appear at the head of the pending list. Finally, in the fourth hop, P_0 TO deliver m_2 and m_1 simultaneously after receiving $\text{ack}(m_2)$ since m_1 was stable after the third hop while P_2 TO delivers m_1 after receiving $\text{ack}(m_1)$. Therefore, for two concurrent messages, m_1 and m_2 using $N = 3$, LCR has the earliest TO delivery after 2 hops while the latest TO delivery occurs after 4 hops.

A New Total Order Solution is Required

Unlike the privilege-based protocols, LCR is a uniform total order broadcast protocol, an extension of the FSR protocol [1]. In LCR, each process sends a message to the same neighbour, called successor, and received from the predecessor. The messages in LCR are sequenced by the processes in the ring using a combination of a local vector timestamp, vt and a last process to order concurrent messages. These attributes of LCR ensure throughput efficiency regardless of the type of traffic. While these attributes ensure throughput efficiency across diverse traffic types, we anticipate potential bottlenecks in the use of a fixed last process for ordering concurrent messages, particularly in scenarios of high message concurrency within a clustered environment.

Furthermore, the vector timestamp contributes to increased information overhead as the number of processes in the LCR cluster increases. In our simulation evaluation, considering up to $N = 9$ processes, the vector timestamp can extend to nine digits for each process. Recognizing these challenges, we posit the need for a novel total order multicast protocol to enhance the effectiveness of a ring-based total order protocol. This proposed protocol must address the identified bottlenecks in LCR, eliminating the dependency on a "fixed last process" for concurrent message ordering and mitigating the space requirements of vector timestamps. It should also incorporate crash tolerance mechanisms to ensure message delivery resilience in the presence of process crashes. The goal is to satisfy the requirements of a dynamic and effective ring-based total order protocol by providing high-throughput, low-latency solutions.

2.7.2 The Raft Protocol

Raft order protocol was developed in [3, 43], especially for overseeing a replicated log [124-127]. A replicated log implies that each process within the cluster environment maintains the same log update of client requests. The goal of the order protocol is to maintain consistency in the replicated log. In the Raft order protocol, the leader receives client requests and stores them

in its log. It then sends the requests to other processes, ensuring that they store the same number of requests in the specific order they were received despite some failed processes. The requests are said to be committed when they are correctly replicated. Each process's state machine executes the committed request in the order they appear in the log and sends a response back to the clients. Thus, the processes within the cluster environment seem to be a single, very dependable state machine. As shown in Figure 2.17 server *S1* is assumed to be the leader and receives a request from a client for processing; it stores it in its log and sends the request to other servers. When it gets an acknowledgement message from other processes that the request it sends has been replicated, it commits the sequence number attached to the request, announces this commit to other processes and its state machine and that of other processes execute the request in the same committed order. Then *S1* sends the execution outcome to the client.

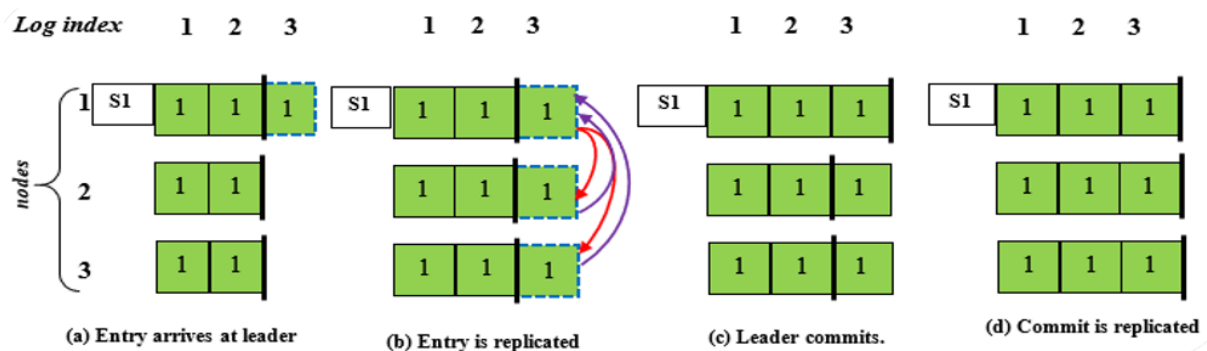


Figure 2.17 A Replicated Log [3]

Raft System Model

These days, distributed systems are common in computing, from large-scale databases to cloud infrastructure. In these systems, maintaining consistency amongst remote processes is a major difficulty. This problem is solved by order protocol, like Raft, which allows a collection of processes to agree on a single value even if some processes fail. The rationale behind Raft is to provide an ordering protocol that is easier to understand and use than its predecessors, such as Paxos. Raft improves learning and practical application in distributed systems by providing a modular and understandable design.

Raft cluster consists of many group of processes N , $N \geq 3$ that are fail-independent and fully connected, typically made up of at most 3-9 servers. To provide fault tolerance, high availability, and swift read performance, processes keep copies of the application state and are

mutually replicating. Any one of the N processes is where Raft clients can submit their requests. Requests can be broadly classified as read or write; the former does not seek state alteration, while the latter does. Write requests, as shown in Figure 2.18, are executed by all servers in the predetermined order after first being submitted to total ordering via a Raft protocol execution under the coordination of the leader process. In Raft, all client requests (read/write) are coordinated only by the leader process and is also responsible for sending responses to the clients. It should be noted that the application state following each write request execution is the same at all processes since the write requests are processed at each process in the same sequence. Therefore, the Raft order protocol plays a critical role in replicating the state of the Raft cluster so that clients can continue to see it as a single, crash-tolerant process.

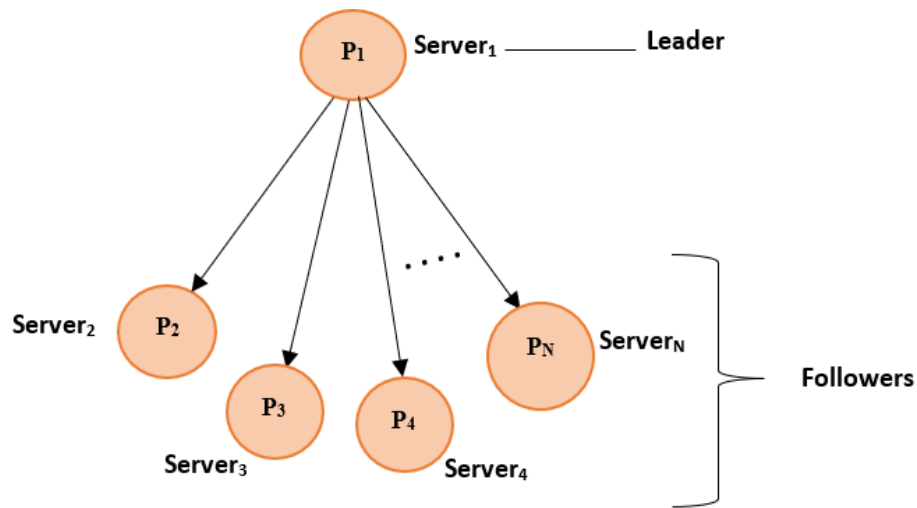


Figure 2.18 Server Hosting Processes in a Typical Raft Cluster [3]

Let $\Pi = \{P_1, P_2, \dots, P_N\}$ represent the set of Raft processes hosted by servers as depicted in Figure 2.18. Raft is an asymmetric protocol: one Raft process is designated as the leader while the rest as followers, and for all N group of Raft processes, $n = N - 1$ denotes the number of followers within the group. Similar to the Two-phase commit protocol [7], the leader exclusively triggers the atomic broadcasting of a message m received from clients, denoted as $abcast(m)$, in response to state-modifying requests. Followers independently respond to each request they receive from the leader. Consequently, when a follower receives a write request (m) for ordering, it forwards the request to the leader to initiate the $abcast(m)$. Once the execution of (m) concludes within the Raft cluster, both the leader and followers uphold the consistent order of the execution outcome while the leader communicates the execution outcome of (m) to the client.

However, each process can take one of the following states: leader, candidate, or follower. Initially, all process states are in followers as shown in Figure 2.19 until the election time has elapsed, one of the processes can become a candidate and request a vote to become an elected leader. When a leader is elected, the rest of the processes become followers. The followers are submissive to the leader. They respond only to the dictates of the leader and candidate.

Raft partitions time into terms of random length. An election starts each term, where one of the processes scrambles to be elected leader. When elected, the process serves the remaining of the term time as the leader. In some cases, the election leads to a split vote, and no leader is elected for that term, but a new term will start immediately, together with another election. At any time, Raft guarantees that one leader is elected. In Raft, the logical clock [7] represents terms which enable processes to discover a leader with incorrect information.

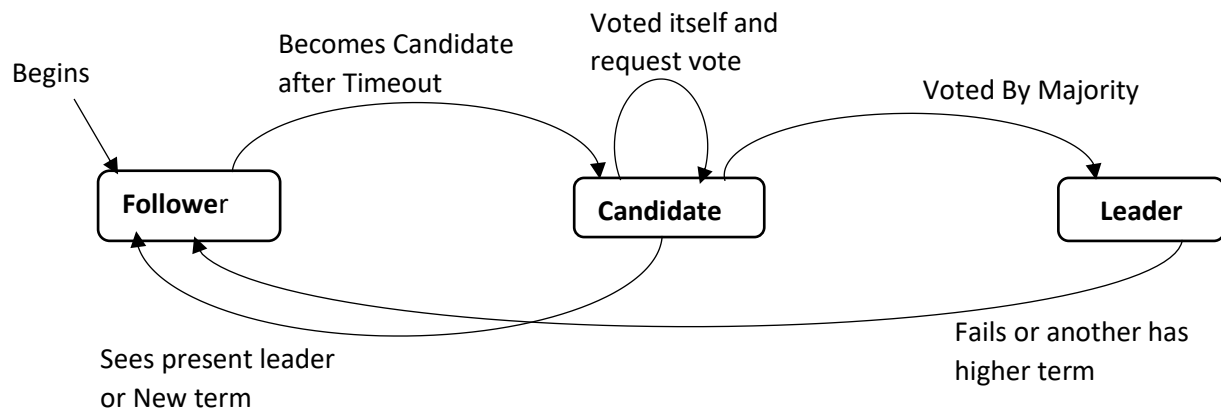


Figure 2.19 Raft Server States [3]

Suppose a process is in term 5 and receives a message from a supposed leader in term 3, the process knows that the supposed leader's information is incorrect since it has missed communication in terms 4 and 5. When processes converse, they share current terms. Anytime a process's current term is small (maybe it crashed and revived) compared to others, it modifies its current term to reflect the current term of the majority. A candidate or leader changes their state to followers anytime its term is old. A process does not receive a request from another process whose term number is stale. Raft processes communicate with others through remote procedure calls (RPCs), primarily using RequestVoteRPC (used during elections) and AppendEntriesRPC (used during log replication). In the subsequent discussions, we simplified the diagram in Figure 2.18 by excluding the servers and processes while using L to denote the leader and F_1, F_2, \dots, F_{N-1} as the followers as shown in Figure 2.20.

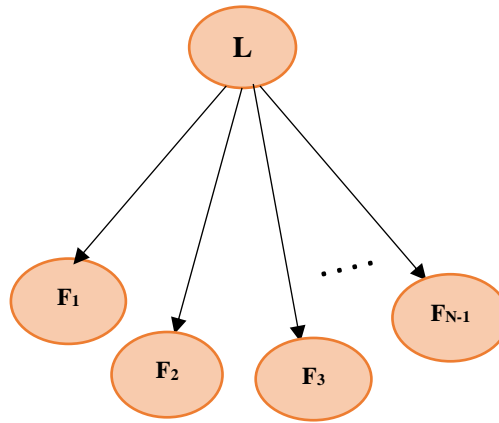


Figure 2.20 Raft Star Structure [3]

Assumptions

A21- Server Crashes

A server process is susceptible to crashing at any moment and may recover after an unpredictable period of downtime. Both the leader and followers in the system maintain stable storage, ensuring that log contents persist even in the event of a crash. When the leader transmits an abcast(m) to the followers, Raft mandates that at least a majority or quorum of processes record (m) in their logs. In the event of a recovery from a crash, the process reads the locally logged atomic broadcasts from stable storage and replays all the logs in-memory. Hence, Raft assumed that the number of crashes that can occur in its N process cluster is bounded by $f = \left\lfloor \frac{N-1}{2} \right\rfloor$ which means $N=2f+1$. f is known as the degree of fault tolerance. Similarly, at least $\left\lceil \frac{N+1}{2} \right\rceil$ processes are operational at any time. This means that if a crash occurs which is in the minority, the majority (quorum) of the processes will still be functional. Thus, for a cluster of 5 processes, Raft can tolerate at most 2 process crashes while at least 3 processes remain operative throughout the Raft protocol executions.

A22- Reliable Communication

Processes are linked through a dependable communication subsystem where messages are unfailingly received in the sequence they are sent, and no messages are lost. Specifically, when a leader transmits a message (m), all functioning followers receive

(m) within a finite timeframe of an unknown delay bound. Additionally, if a leader transmits m_1 followed by m_2 , all operative followers will receive m_1 before m_2 . Raft employs First-In-First-Out (FIFO) channels for all communication, ensuring that messages are transmitted in a sequential order. The preservation of correct ordering is maintained as long as messages are processed in the sequence of their reception.

View stamped replication [128, 129] protocol have several similarities to the Raft order protocol, but the Raft protocol has some unique characteristics as follows:

- (i) **Strong Leader:** Raft uses a leader to coordinate the activities of other processes. For instance, the leader receives log entries and passes them to other cluster processes. This helps Raft coordinate control over the replicated log and brings understanding to its users.
- (ii) **Leader Election:** Raft elects leaders according to a specified condition. A randomised election timeout guarantees that voting problems are sorted out easily and quickly. When a leader fails, other operative processes must elect another leader.
- (iii) **Membership Changes:** When the need for membership changes arises, Raft uses a joint consensus approach to allow the quorum for the two configurations to overlap while the transition is ongoing. This enables the cluster to continue working correctly while configuration updates are made.

Leader Election Requirements

A leader must be elected when the present leader fails or when Raft process clusters begin operation. Initially, the process starts in a follower state and continues in this state as long as the current leader or candidate communicates with them. This communication is `appendEntriesRPC` (heartbeat) with no log in them. The leader does this to maintain its leadership role within the cluster. A follower may elect another leader if it does not receive any communication from the current leader after a certain period called election time-out. At this time, the follower changes its state to the candidate, increases its term counter, votes itself, and sends a communication to the other servers simultaneously asking for their vote. Thus, three things could happen to the candidate as follows, as illustrated in Figure 2.21.

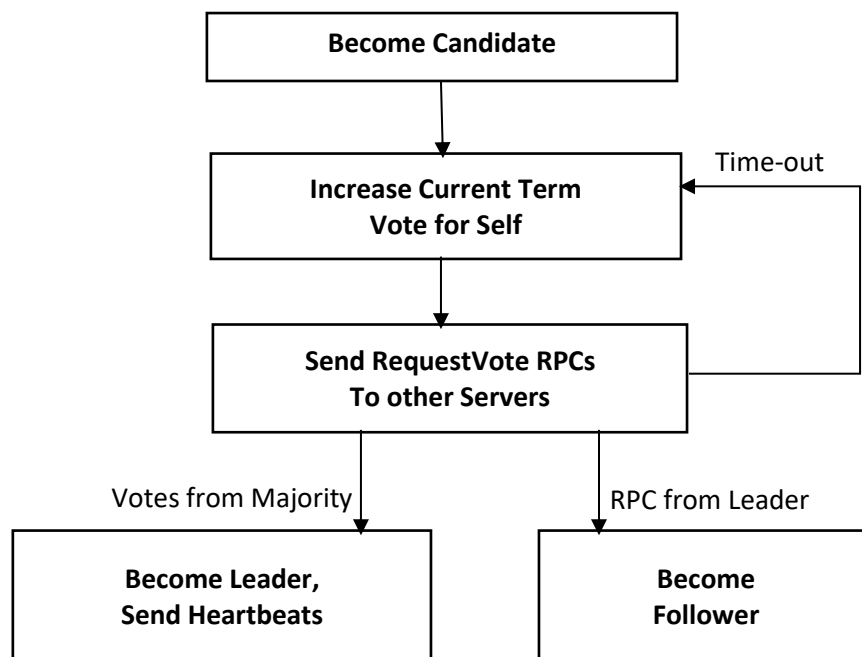


Figure 2.21 Raft Leader Election [3]

- (i) The candidate wins the election provided its log is up-to-date for the current term and a quorum number of processes voted for the candidate. Hence, it becomes the leader after receiving a quorum number of votes from the followers and starts to send a heartbeat to followers to maintain its leadership within this term.
- (ii) Another process could communicate with the candidate assuming to be the leader. Suppose the assumed leader's term time exceeds the candidate's current term. In that case, the candidate's election is thwarted, and the candidate goes back to the follower state and acknowledges the leader as the right leader. However, if the assumed leader's term number is less than the candidate's, the candidate rejects the assumed leader's communication and continues as the candidate.
- (iii) Followers could become candidates simultaneously and request votes. This causes a shared vote (deadlock), and no winner emerges. When this happens, a new term time begins with a new election.

Raft uses arbitrary election timeouts to ensure the shared vote does not occur indefinitely. The election time-out is chosen between the random time of 150-300ms. This clear illustration of leader election is what makes Raft highly understandable.

Log Replication

Log replication is supervised by the leader responsible for receiving client requests. The client request contains the command to be processed by the replicated state machine within the Raft cluster. When the leader receives the request, it adds it to its log as a new entry after attaching a sequence id to the request. The leader then broadcasts a copy of the received request to the followers as a message using AppendEntry RPC. There is a possibility that some of the followers may not be available at the time of sending, so it continues to send the new log entry to the followers until all servers store all new log entries. When the leader receives a quorum of responses from the followers that the log entry is replicated, the leader commits the entry to its local state machine. This helps the leader to commit the previous log entries it stored. The follower also commits its log when it learns that the leader has committed it. This helps to guarantee consensus of log entries among servers within the cluster and ensures that the safety rule of log matching is considered. During the failure of the leader, the logs go out of sync between the followers since the leader's log may not have been completely replicated. The new leader will try to solve this inconsistency in the log state among the followers by making sure that followers replicate its log. To handle this scenario, the new leader will try to find where its log entry matches with each follower's log and erase the follower's log entry after that point. The new leader then sends the follower its log entries from that point on so that the follower can restore its log to match with the new leader's log. This sequence of actions brings back the log consistency within the cluster for the remainder of the term.

Safety

The safety property of Raft establishes the rules for electing a new leader and the requirements needed before a leader is elected. The rules are outlined below:

- (i) Only a single leader is elected per term time.
- (ii) Raft ensures that log entries have a single direction flow, from leaders to followers, and the leader will never change their previous log entries.
- (iii) When two logs consist of entries with identical index and terms, then the log entries are the same throughout the entire index.
- (iv) A committed log entry at a particular term time must be present in the 'leaders' log for that term.

State Machine Safety Property

This guarantees that no other process can put a contrasting log entry when a process puts a log entry into its state machine at a particular index. The implication is that every process must put identical log entries to their state machine in a similar order.

Follower and Candidate Crashes

When a follower or candidate fails, any RPCs sent to it fail, too. Raft takes care of these failures by continually contacting the failed server. If the failed process restarts, the RPC will go through successfully, but if the process already contains the RPC message, it will reject it.

Timing and Availability

The Raft is built so that its safety is not time-dependent. It should not give a wrong result because some events happen relatively fast or slow, but the system's availability should undoubtedly be time-dependent. Availability here means the ability to attend to client requests swiftly. Leader election is heavily time-dependent and must fulfil the conditions below to maintain constant leadership.

$$\text{broadcastTime} \ll \text{electionTimeout} \ll \text{MTBF}$$

where

- *broadcastTime* is the mean time it takes a process (leader) to broadcast RPCs in parallel to every server within the cluster and get their feedback.
- *electionTimeout* is the period of time during which the expiration initiates a leader election, triggered by the first process to detect the expiration of election time out.

MTBF is the mean time taken between the failure of a single process.

The *broadcastTime*, which is between 0.5ms and 20ms, is designed to be shorter than the *electionTimeout* (which ranges from 10ms to 500ms) thus enabling the leader to consistently provide the heartbeat messages necessary to prevent followers from initiating elections; due to the randomized nature of *electionTimeout*, this inequality relation helps to avoid split votes. The *electionTimeout* is also shorter than *MTBF* (which could be many months or more) to allow consistent progress of the system without interruptions resulting from unnecessary process failures. The system will be down when the leader crashes for about the duration of the *electionTimeout*.

Cluster Membership Changes

Cluster configuration in Raft is assumed to be fixed, but it should be dynamic in real-life applications. For instance, remove a failed process and bring in a new one. The changeover could be done manually as follows:

- (i) Configuration changes may be implemented by taking the entire cluster offline, updating the configuration files, and subsequently restarting the cluster. However, this approach would result in the cluster being temporarily unavailable during the transition.
- (ii) As an alternative, a new process could assume the role of a cluster member by adopting its network address. Nevertheless, the administrator must ensure that the replaced process will remain permanently offline; otherwise, the system might compromise its safety properties (e.g., leading to an additional vote).

The two approaches outlined above have significant demerits, and any manual approach might lead to operator error. To avoid these problems, Raft automated its configuration changes. The safety of Raft configuration changes must guarantee that two leaders are not elected at that point. Allowing processes to switch straight from the old configuration to the new one is unhealthy. If this happens, the cluster may be split into two unconnected quorums of processes. Raft uses a two-way approach to avoid this problem. Firstly, it commits the joint consensus, and secondly, it moves to the new configuration. Hence, a joint agreement contains the combination of the old configuration and the new:

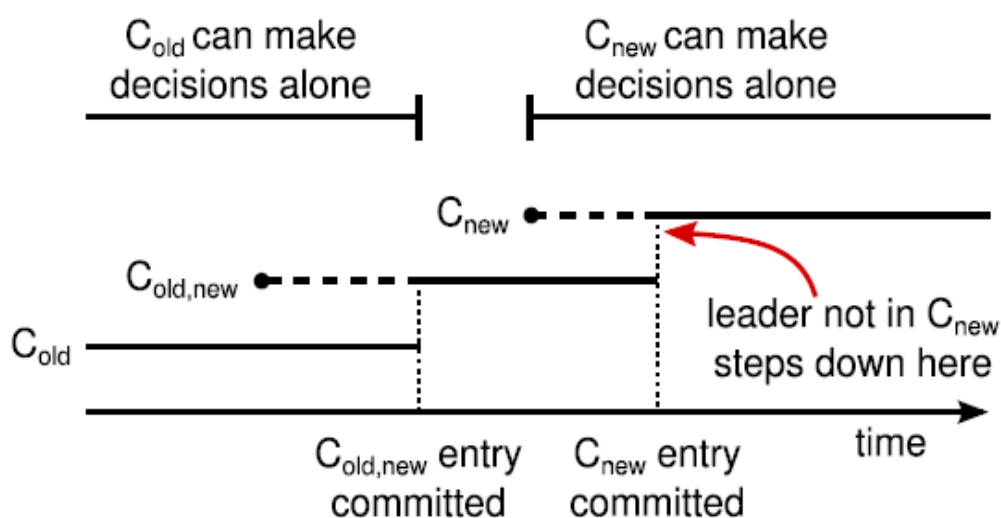


Figure 2.22 Timeline for a configuration change [3]

- Replication of all log entries to all processes happens in both configurations.
- A leader could be elected from any of the configurations.
- Reaching an agreement needs distinct majorities from both configurations.

Dashed lines represent configuration entries that have been created but not yet committed, while solid lines depict the most recently committed configuration entry (see Figure 2.22). Initially, the leader creates the $C_{old, new}$ configuration entry in its log and commits it to both $C_{old, new}$ (C_{old} (a majority) and C_{new} (a majority)). Subsequently, it creates the C_{new} entry and commits it to a majority of C_{new} . There is no moment in time during which C_{old} and C_{new} can independently make decisions.

Log Compaction

Raft's log increases as more client requests are received, but this growth has a fixed bound in a real system. Excessive obsolete information in Raft logs may delay the response time of Raft and bring about availability problems if there is no mechanism to remove such unwanted information from the Raft log. The Raft uses snapshotting to bring about compaction. Snapshotting is the process of writing the accumulated log into a snapshot on reliable storage, and then the snapshotted range of the log is eliminated.

Two fundamental problems affect snapshotting:

- (i) The processes must choose the time to snapshot. When there is frequent snapshotting, disk bandwidth and energy are wasted. If it does not snapshot regularly, storage capacity is wasted. The best time to snapshot is when the log gets to a specific fixed size in bytes.
- (ii) The writing of snapshots can take a reasonable time interval, which might slow down system operations. The copy-on-write approach is used to solve this problem. It can only receive the new update to avoid affecting snapshotting. The basic idea behind Copy-on-write [130] is to delay data copying until it is absolutely required to prevent needless information duplication.

Raft-Client Interaction

The request of Raft's client is sent to the leader for processing, as shown in Figure 2.23. As the client request arrives, it connects to an arbitrarily chosen process (Phase A and B). If the process replica at that time is not the leader of the cluster, it rejects the request from the client

and sends updated information about the current leader to the client. The client resends its request to the leader (Phase 1). The leader puts the request in its log and broadcasts it to all followers after attaching a sequence number to the request (Phase 2). The followers then inform the leader that they have received the request. After receiving a quorum of feedback from the followers, the leader commits the sequence number on the request, announces this commit operation to the followers. Then both the leader and the followers execute the request in their logs via state machine in the committed order while the leader sends a reply to the client (Phase 3). This is the mechanism through which consensus is established among the processes within the clustered environment of the RAFT protocol.

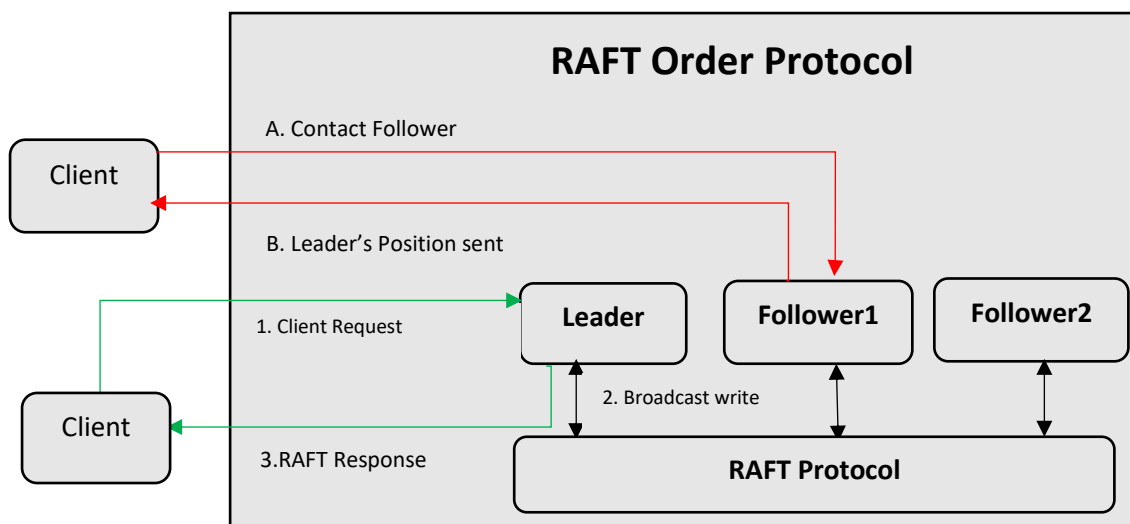


Figure 2.23 Raft Client Interaction [3]

A leader's crash results in a timeout for the requesting client, prompting the client to attempt reconnection with the next leader. To ensure the success of this interaction, Raft incorporates linearization [56], guaranteeing the simultaneous execution of operations exactly once from request invocation to response. Nevertheless, Raft may execute a request multiple times in the event of a leader crash. If the initial leader fails to respond to the client after committing the log due to a failure, the client must connect to another leader, leading to the potential repetition of the request's execution. To avoid this, the client must attach a special number to every request it sends. So, in the event of a leader's crash, the new leader, with the help of the state machine, can track the newest serial number executed for each client. If the unique number it receives for execution is already processed, it sends a response to the client instantly without processing the request anew. It is important to highlight that the leader attaches a sequence to

the client's request to maintain the order of requests within the cluster, ensuring FIFO sequencing. Simultaneously, the unique id attached by the client to its request serves the purpose of identifying whether the request has been executed, particularly in scenarios where leader failure occurs, and a new leader is elected, necessitating the client to resend the request.

Moreover, it is not necessary to write anything into the log when performing read-only actions within the Raft cluster. However, without further safeguards, there is a risk of returning old data because the responding process might be a leader who may have been replaced by a new leader about whom the replying leader is uninformed. Raft takes two additional measures without using the log to ensure linearizable reads do not return stale data: (i) a leader must have up-to-date information about committed entries. While the Leader Completeness Property ensures that a leader has all committed entries, a leader may not know which entries are committed at the start of its term. To remedy this, Raft ensures that each leader logs a blank no-op entry at the start of each term, allowing it to identify committed entries. (ii) To handle a read-only request, a leader must first determine if it has been replaced. If so, its information might be outdated due to the election of a more current leader. Raft manages this by having the cluster's leader send heartbeat messages to at least a quorum of members of the cluster before responding to read-only requests. In addition to preventing the return of outdated data during read-only operations, this procedure guarantees that the leader remains current.

However, the Raft protocol has the following performance weaknesses: (1) The leader represents a potential single point of failure in the system, emphasizing the need for a swift election of a new leader in the event of the current leader's failure. Delays in electing a new leader or an overloaded leader handling client requests can lead to a reduction in system throughput and an increase in latency. (2) The experimental study of giving the quiz to students is not an adequate standard for measuring the understandability of Raft.

2.7.3 Differences among Zookeeper, Paxos and Raft Protocols

In this section, we highlight the distinctions among Zookeeper, Paxos, and Raft protocols concerning leader election, communication mechanisms, the number of outstanding requests, and the number of communication phases. These aspects bear significance in the context of this research, especially in understanding the communication dynamics between a leader and followers during the broadcast phase.

Leader election: Zookeeper and Raft protocols are different from Paxos since they divided their protocol execution into phases. This distinction arises due to the enhanced safety/liveness

property present in Zookeeper [131] and Raft [3]. This robust safety/liveness property guarantees that, at any given moment, there can be, at most, only one leader in these protocols. In contrast, Paxos lacks this strong leader property, leading to the coexistence of multiple leaders simultaneously.

Communication with the process: Zookeeper and Paxos employ a messaging model for communication with processes, involving a minimum of three messages for each state update. In Zookeeper, these messages include proposal(m), ack(m), and commit(m). Conversely, Raft utilizes Remote Procedure Calls (RPC) for communicating with processes during state update replication. However, Raft optimizes the RPC call overhead by employing certain techniques consistently. For instance, the leader in Raft can initiate AppendEntries RPCs to both replicate the entry and send heartbeats. In contrast, Zookeeper involves sending heartbeat messages separately, without piggybacking on abcast

Multiple outstanding requests: Zookeeper supports the execution of multiple concurrent abcasts, assuring that the abcast will be committed following the First-In-First-Out (FIFO) order. In contrast, Paxos lacks a direct provision for such a feature. When a proposer sends abcasts individually in Paxos, the committed order of abcasts may not adhere to the required order dependencies, potentially leading to an inconsistent state among Paxos process. To address this issue, a known solution involves bundling multiple abcasts into a single Paxos abcast, allowing only one outstanding abcast to be processed at any given time. However, this solution has trade-offs, negatively impacting either throughput or latency based on the chosen bundling size.

Number of phases: The Zookeeper protocol encompasses three distinct phases: discovery, synchronization, and broadcast phases. In contrast, Paxos and Raft do not have a dedicated synchronization phase. Instead, in Paxos and Raft, followers maintain synchronization with the leader during the broadcast phase by comparing log index and term values for each entry. While the absence of a specific synchronization phase simplifies the Raft algorithm's implementation, it may potentially impact performance if confronted with a lengthy and inconsistent replication log.

2.8 Discrete Event Simulation (DES)

Simulation refers to the mechanism of mimicking the operation of a real-world system over a range of time. In a simulation, a system's artificial history is created and observed, manually or digitally, to make deductions about the operational features of the actual system [132]. The simulation of total order protocols in a cluster environment can benefit from the discrete nature of the system state. The state only changes its variables based on which of the several predefined events (message sent, message arrival, message received, message ordered, and message delivery) occur. This demonstrates that a discrete-event simulation (DES) approach is appropriate [133]. DES represents a system's operation as a series (discrete) of events that happen across time. Each event occurs at a specific time and represents a shift in the system's state [134]. Since there is no assumption that the system will change between two consecutive events, the simulation time can go directly to when the next event will occur. This technique is known as next-event time progression. Discrete event simulation has been used in several fields of endeavour, and primarily, it is used in computer networks to model novel protocols and diverse system designs (distributed, hierarchical, centralised, and peer-to-peer) before their actual implementations. In such a model, some performance evaluation metrics are defined, such as arrival time, service time, and bandwidth size.

There are three generally accepted approaches in DES:

Activity Oriented

Activity-oriented DES progresses across a predetermined time range and splits time into discrete steps, changing the state when the time of specified events is approached. This type of simulation moves extremely slowly since the simulation must go through every step, even when nothing changes in the system.

Events Oriented

Event-oriented DES assumes that the system's state remains constant between events, negating the need to simulate these times as in an activity-oriented scenario. The simulation keeps track of a list of impending events, and when the next one appears, it advances the simulation's time to coincide with it. The simulation maintains logic for adding new events to the set and how each event modifies the system's state.

Process Oriented

Process-oriented DES is similar to event-oriented DES in that it only addresses events and ignores the intervals between them. On the other hand, the event-oriented DES completes all of its actions in a single serial process or thread that may include logic for multiple entities. In contrast, process-oriented DES uses distinct processes (threads) for every simulation entity. These distinct processes send events back and forth and generate and consume events from the pending events set. This approach has the advantage of being more modular because each entity's logic may stand alone. This makes the simulator easier to understand and allows for the reuse of its many components. However, it complicates the logic of the simulator implementation by adding concurrency and its related issues.

Even though process-oriented DES can produce more modular, reusable code, developing these simulators can be challenging. Therefore, an event-oriented DES technique was used to design and simulate the solutions of our new protocols in a Java programming framework.

2.9 Summary

This chapter presented a comprehensive background for the state of the art in total order processing. LCR is the first throughput optimal uniform ring-based total order broadcast protocol. It only relies on point-to-point inter-process communication and has a linear latency with respect to the number of processes. LCR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes. We determined that the LCR algorithm utilizes a local vector clock to vector timestamp messages and employs a fixed “last” process for ordering concurrent messages. The algorithm assumes crash failures, hence the total number of processes that can fail is $f=N-1$. Also, Raft is a leader-based total order protocol designed for state replication across a distributed system of nodes. In Raft, a leader node is elected from among the system nodes to coordinate all state updates occurring anywhere in the system. This leader collects updates and replicates them to all other nodes (followers). For the sequence number attached on an update to be committed and executed in the leader's state machine, the leader must receive acknowledgements from a majority of the nodes, including itself. Only the leader node sends responses back to the clients. The Raft algorithm can tolerate f crash failures. To tolerate f crash failures, the Raft system must have at least $N=2f+1$ nodes. In Chapter 3 and Chapter 5, we introduce our proposed solutions to the LCR and Raft protocols. We discussed the rationale behind our solutions and the crash-tolerant

assumptions considered in our solutions. The novel approach of a unique last process is used to order concurrent messages, the Lamport logical clock for timestamping the messages including the crash failure assumption that requires $N=2f+1$ nodes in order to tolerate f crash failure. This approach led to a genuine alternative to LCR, called daisy chain total order protocol, DCTOP, for brevity. Additionally, the modification of the Raft logical message dissemination and implementation of a strategy where the leader requires only a single acknowledgement before committing the sequence number assigned to a message in its log resulted in the development of two Raft variants, Chain Raft and Balanced Fork Raft.

Chapter 3

Daisy Chain Total Order Protocol - DCTOP

This chapter develops an alternative protocol to LCR, a ring-based total order protocol within an N -process cluster, $N \geq 3$, as discussed in Section 2.4.1. It is critical that at least a majority of these N -processes are active throughout the protocol execution to maintain uniform total order delivery of messages. The daisy chain total order protocol (DCTOP) uses Lamport *logical clocks* and a dynamic approach for ordering concurrent messages that are distinct from LCR. The goal is to provide high throughput and low-latency total order delivery, *TO delivery*, which will be evaluated in the next Chapter.

Logical Clock was introduced by Leslie Lamport [7] as a method for establishing a sequence of events within a group using just an integer. In Lamport's scheme, each process within a distributed system maintains its own local clock as an integer value. Before a process sends a message, it timestamps the message with the clock value which is then incremented after that message is sent. Note that no activity that could increment the clock should take place between timestamping and sending events. Whenever a process receives a message, it updates its own local logical clock to a value greater than the timestamp of the received message if necessary. This clock usage scheme is essential so that events can be totally ordered correctly.

The remainder of this chapter is organized as follows. We begin in Section 3.1 by discussing the rationale behind the introduction of DCTOP. Section 3.2 presents the system model which outlines the system requirements and assumptions on which the proposed protocol is built. Section 3.3 presents the principles behind the DCTOP approach. Section 3.4 presents a non-uniform daisy chain multicast, `DC_mcast`, a primitive used by DCTOP. This is extended to uniform `DC_mcast` and non-uniform total order multicast. Section 3.5 then presents DCTOP, a uniform and total order multicast. Next, Section 3.6 introduces the DCTOP performance operations using single and concurrent multicast including the performance benefits of DCTOP over LCR. Next, Section 3.7 discusses the fairness control primitives integrated into DCTOP. Finally, Section 3.8 presents the summary of the chapter.

3.1 Rationale

LCR protocol, as discussed in Section 2.7.1, is a ring-based total ordering protocol which ensures that all processes within a cluster deliver received messages in the same order. LCR protocol is an improvement to the FSR protocol [1], which combines the ring topology for high throughput dissemination and uses a vector timestamp and a globally fixed “last” process for ordering concurrent messages.

However, within LCR, the use of a vector timestamp occupies more space in a message, thereby increasing the message size. Suppose we have a cluster of 4 processes, P_0 , P_1 , P_2 , and P_3 . Each process maintains its own local vector clock which contains one integer for each process, $[0, 0, 0, 0]$. Then any message sent from one process to another must include the vector clock information into the message as a vector timestamp to ensure total ordering within the cluster. In this case, the message size is increased by the size of 4 integers. As a result, the globally fixed last process cannot rapidly sequence concurrent messages, and this could elongate the message TO delivery average maximum latencies. In addition, a vector timestamp requires N bits, one for each process. So, there is an increase in information overhead, which means that information within the LCR message is more, so as N increases this information overhead increases (we considered up to $N = 9$ process cluster in our experiment, indicating that a message vector timestamp could extend up to 9 digits). Also, the strict assumption of $f=N-1$ in LCR could increase message delivery latency.

Additionally, for enterprise messaging systems [135-139] that utilize order protocols such as LCR, for total ordering messages, the presence of an increased information overhead can negatively impact the performance of the message queue. The processing and storing of messages with large information overhead may require more time and system resources.

Thus, we believe that improving LCR latency is possible using an alternative design. This led to the development of DCTOP. The main goals of the proposed design are threefold: (i) To enhance the latency by utilizing Lamport logical clocks for sequencing messages. Thereby, the timestamp in our approach is independent of the size of N , unlike vector timestamps, and (ii) To employ a dynamically determined concept of the “last” process for ordering concurrent messages instead of using a fixed last process. That is, every message has its own last process that enforces identical ordering of the concurrent messages. (iii) A more relaxed assumption of $N=2f+1$ might ensure lower latency in message delivery.

3.2 System Model

Figure 3.1 depicts a system of N processes denoted as $\text{Node}_0, \text{Node}_1, \dots, \text{Node}_{N-1}$ interconnected by a network. Each node has a process dedicated to total ordering of messages. Inter-process communication occurs by these nodes exchanging messages with each other. Therefore, the distributed system considered here is made up of a finite number of nodes. For example, Node_0 is hosting process P_0 , Node_1 is hosting process P_1 , ..., and Node_{N-1} is hosting process P_{N-1} as shown in Figure 3.1. Typically, the twisted pair cables (links) connect each node to the communication network, one for sending and another for receiving, also shown in Figure 3.1.

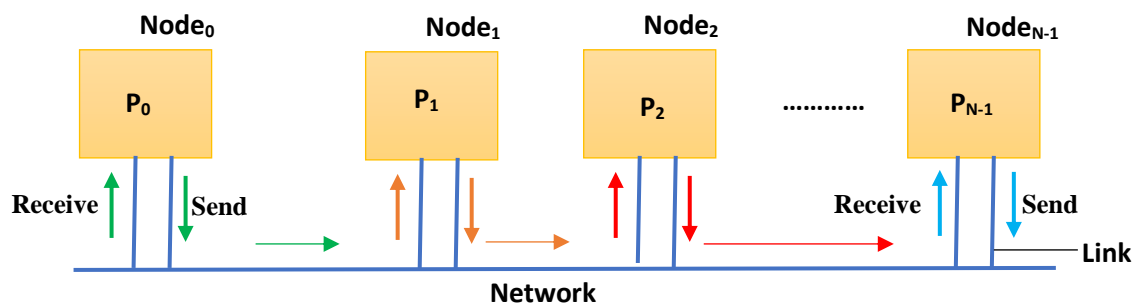


Figure 3.1 Traditional Network System

Data communication is assumed to be error-free meaning that message transmissions are error-free, have a “bounded in expectation” message transmission delays since we use an exponential distribution with a specific mean. In an exponentially distributed system, the actual delay values will vary significantly from instance to instance due the nature of distribution. Some delays will be much shorter than the mean while others will be longer. However, the average delay over a large number of transmissions will converge to the specified mean. The system also maintains FIFO - if one process transmits two messages to a given process, the destination will receive those two messages in the order they were transmitted. Note that a transmitted message from a source process can be its own message or a message it received from another source and chose to forward. The latter will also be referred to as “forwarded” messages.

Moreover, these nodes are arranged in a logical ring as shown in Figure 3.2. Messages are sent in one direction only and this forms the unidirectional and logical ring for the DCTOP. We chose this model because this logical ring enables the system to enhance sender throughput because the sending and receiving of messages occur in parallel at a given node.

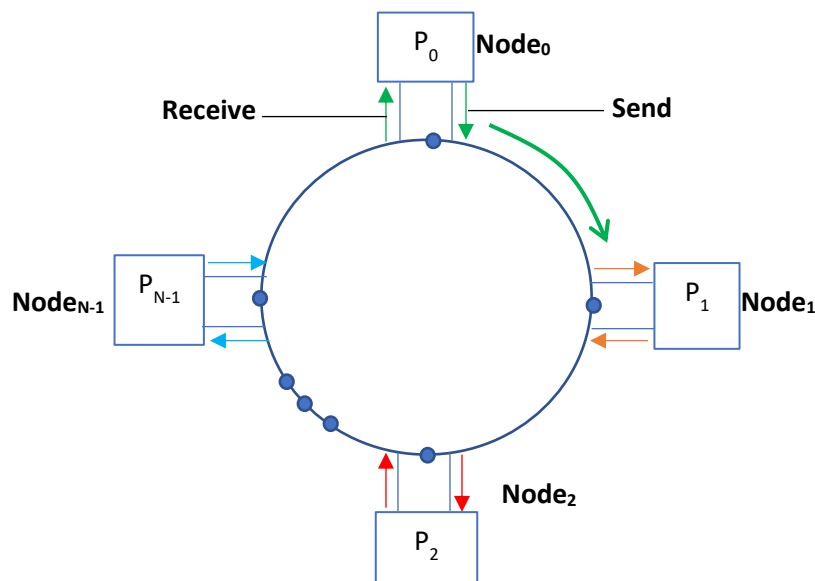
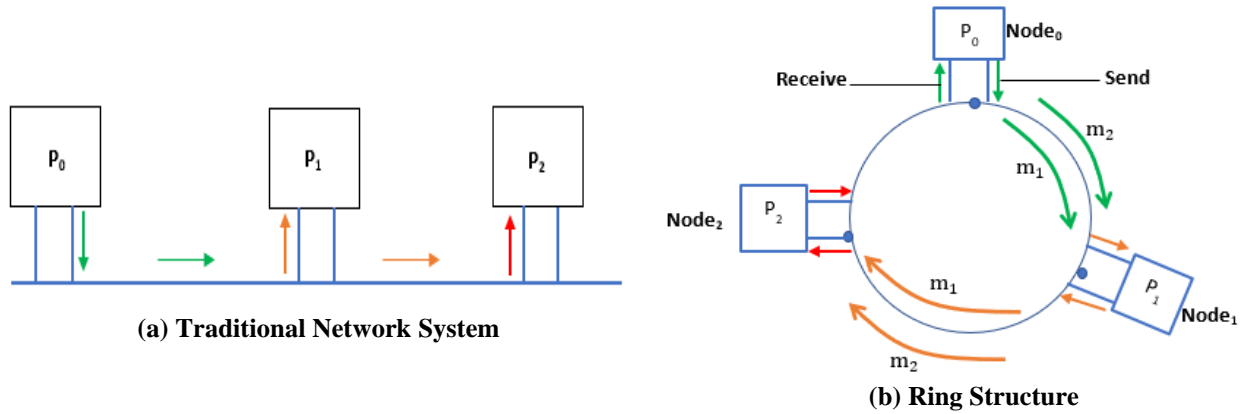


Figure 3.2 Logical Ring in DCTOP Model

To understand this, consider P_0 sending two messages m_1 , and m_2 only to P_1 and P_2 by multiply unicasting using the traditional network system (see Figure 3.3a) and the ring structure (see Figure 3.3b).

Figure 3.3 P_0 Sending m_1 , and m_2 to P_1 and P_2

As depicted in Figure 3.4, P_0 sends m_1 to P_1 in the 1st time step. In the 2nd time step, P_0 sends m_1 to P_2 while P_1 also receives m_1 during the same time step. In the 3rd time step, P_0 sends m_2 to P_1 and P_2 also receives m_1 in the same time step. Then, in the 4th time step, P_0 sends m_2 to P_2 and P_1 also receives m_2 within the same time unit. Finally, P_2 receives m_2 in the 5th time step. For simplicity, we assumed that sent messages take zero transmission time and are received in the next time step. Therefore, the whole transmission of two messages by P_0 to two processes by multiple unicasting using the traditional network system take 2×2 time steps.

Hence to send m messages to N processes, the sender will take $m \times N$ time steps. The throughput is $\frac{1}{N}$ message per time steps. Thus, when the sender is sending to multiple processes, N is large, the sender throughput suffers and the latency increases also.

	m_1		m_2		
Send	P_0 sends m_1 to P_1	P_0 sends m_1 to P_2	P_0 sends m_2 to P_1	P_0 sends m_2 to P_2	
Receive		P_1 receives m_1	P_2 receives m_1	P_1 receives m_2	P_2 receives m_2
Time Steps	1	2	3	4	5

Figure 3.4 Time Steps in Traditional Network System

Conversely, suppose P_0 sends two messages m_1 , and m_2 to P_1 and P_2 . The activities are represented in Figure 3.5 over a ring structure. Here, P_0 sends m_1 to P_1 in the 1st time step. Thereafter, P_0 sends m_2 to P_1 in the 2nd time step. After that, m_1 , and m_2 get propagated to P_2 by P_1 though P_2 still receives m_2 by the fifth time step as in Figure 3.4. Therefore, P_0 only takes two-time steps to transmit two messages to P_1 .

	m_1		m_2		
Send	P_0 sends m_1 to P_1	P_0 sends m_2 to P_1	P_1 sends m_1 to P_2	P_1 sends m_2 to P_2	
Receive		P_1 receives m_1	P_1 receives m_2	P_2 receives m_1	P_2 receives m_2
Time Steps	1	2	3	4	5

Figure 3.5 Time Steps in Ring Structure

Therefore, the ring structure reduces the multiple unicast to a single unicast, this underscores the rationale behind employing the logical ring because DCTOP underneath requires that when a process sends a message to its neighbour, the neighbouring process helps the sending process to send the message to its own neighbour and this continues until every process within the ring has received the message. The sending throughput of the ring structure is 1 message per time step, and this becomes independent of N . However, while the sending process can multiply unicast its messages swiftly, these messages may have to traverse several hops before reaching

every process, leading to increased latency. Therefore, it is crucial to perform performance evaluations.

In the subsequent discussions, we simplified the diagram in Figure 3.2 by excluding nodes, the links, but by showing only the processes as shown in Figure 3.6.

3.2.1 Assumptions

This section defines the key assumptions made when designing the DCTOP protocol.

AS1- Crash Tolerance

We assumed that the number of crashes that can occur in an N process cluster is bounded by $f = \left\lfloor \frac{N-1}{2} \right\rfloor$, which implies $N=2f+1$, where $\lfloor x \rfloor$ denotes the largest integer $\leq x$. The parameter f is known as the *degree of fault tolerance* as described in Raft. For example, when N is even, say $N = 4$, then $f = 1$. Also, when N is odd, say $N = 5$, then $f = 2$. Let us denote r , as $r = f + 1$. This means that when f process crashes, the majority of the processes ($\geq r$) will still be functional. The degree of fault tolerance is chosen exactly as in Raft because Raft provides a mechanism to reconfigure the system back into a ring whenever crashes do occur.

As in Raft, also we do not account for other types of failures, such as Byzantine failures [140-142]. A Byzantine failure is a situation in a distributed system where system components fail, and there is insufficiently reliable information to determine whether a component has failed, e.g., a failed process might appear to be operative to other processes, hindering achieving total order within the cluster of environments.

AS2- Asynchronous Reliable Communication

When an operative process P_i sends a message m to CN_i, P_j , then the operative process P_j eventually receives the message m . Then P_j sends this received m from P_i to its own CN_j, P_k . if P_k is operative, it will receive the message m . This continues until the message m from P_i is received by every process within the ring. We assume that message sending and forwarding take precedence over receiving messages. For all unicasts sent between processes, we assume the messages are sent over a reliable network protocol such as a TCP [143] and arrive within an unknown delay bound.

3.2.2 Ring Structure

Let $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$, be the set of processes connected in a unidirectional ring framework with variable cluster sizes. Figure 3.6 illustrates the DCTOP ring structure of our system model and simplifies Figure 3.2. We assume that processes are uniquely ordered, say, as: $P_0 < P_1 < P_2 < \dots < P_{N-2} < P_{N-1}$ (see Figure 3.6).

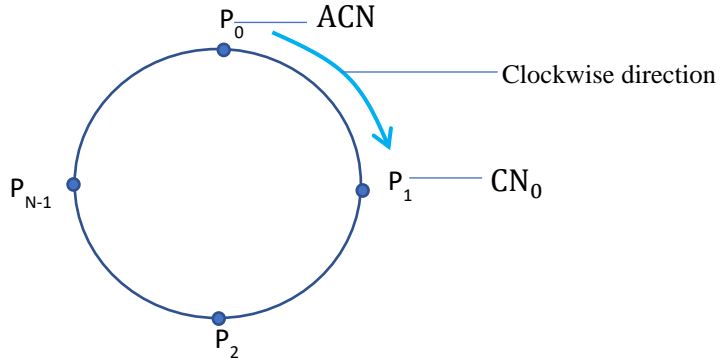


Figure 3.6 DCTOP Ring Structure

Definitions: CN , ACN , and $Hops_{i,j}$

For every process P_i , $0 \leq i \leq N - 1$, where N is the total number of processes within the cluster, as shown in Figure 3.6, we define a clockwise neighbour of P_i (CN_i) as the immediate successor of P_i , $CN_i = P_{i+1}$ or $CN_i = P_0$ if $i = N - 1$. Similarly, the anticlockwise neighbour of P_i (ACN_i) is defined as the immediate predecessor of P_i , $ACN_i = P_{i-1}$ or $ACN_i = P_{N-1}$ if $i = 0$.

As messages flow only in the clockwise direction (see Figure 3.6), P_i receives from ACN_i and sends/forwards only to CN_i . For example, P_0 can only send/forward messages to its CN_0 , that is, $CN_0 = P_1$, while P_1 can only receive messages from its ACN_1 , that is $ACN_1 = P_0$. Hence, for $N = 4$ cluster, when P_0 sends a message, its last process is ACN_0 , P_3 ; when P_1 sends a message, its last process is ACN_1 , P_0 ; when P_2 sends a message, its last process is ACN_2 , P_1 ; and when P_3 sends a message, its last process is ACN_3 , P_2 .

Additionally, we introduce the definition of $Hops_{i,j}$ which is defined as the number of hops between any two processes from P_i to P_j in the clockwise direction:

- $Hops_{i,j} = 0$, if $i = j$.
- $Hops_{i,j} = (j - i)$, if $j > i$, and
- $Hops_{i,j} = (j + N - i)$ if $j < i$.

Hops Invariant: for any P_i and P_j , $i \neq j$;

$$\text{Hops}_{i,j} + \text{Hops}_{j,i} = N$$

This reflects the fact that N hops are needed to come back to the same process after visiting every other process exactly once.

3.3 Protocol Principle

The protocol has three design aspects: (i) message sending, receiving, and forwarding, (ii) timestamp stability, and (iii) crashproofing of messages, which are described in detail one by one in this subsection.

(1) Message Sending, Receiving and Forwarding:

We use the Lamport logical clock to timestamp a message m within the ring network before m is sent. Therefore, m_ts denotes the timestamp for message m .

(2) Timestamp Stability:

A message timestamp TS , $TS \geq 0$, is said to be *stable* in a given process P_i if and only if the process P_i is guaranteed not to receive any m' , $m'_ts \leq TS$ any longer.

Observations:

- 1) A timestamp $TS' < TS$ is also stable in P_i when TS becomes stable in P_i .
- 2) We use the term “stable” to refer to the fact that once TS becomes stable in P_i , it remains stable forever. This usage corresponds to that of the “stable” property used by Chandy and Lamport [144]. Therefore, the earliest instance when a given TS becomes stable in P_i will be our interest in the later discussions.
- 3) When TS becomes stable in P_i , the process can potentially TO-deliver all received but undelivered m , $m_ts \leq TS$, because the stability of TS eliminates the possibility of P_i ever receiving any m' , $m'_ts \leq TS$, in the future.

(3) Crashproofing of Messages:

A message m is *crashproof* if m is in possession of at least $(f + 1)$ processes. Therefore, a message m is crashproof in P_i when P_i knows that m has been received by at least $(f + 1)$ processes.

3.3.1 Message Sending, Receiving and Forwarding

Every process P_i is equipped with a Lamport logical clock denoted as, LC_i , which is an integer initialized to zero at the system set-up time.

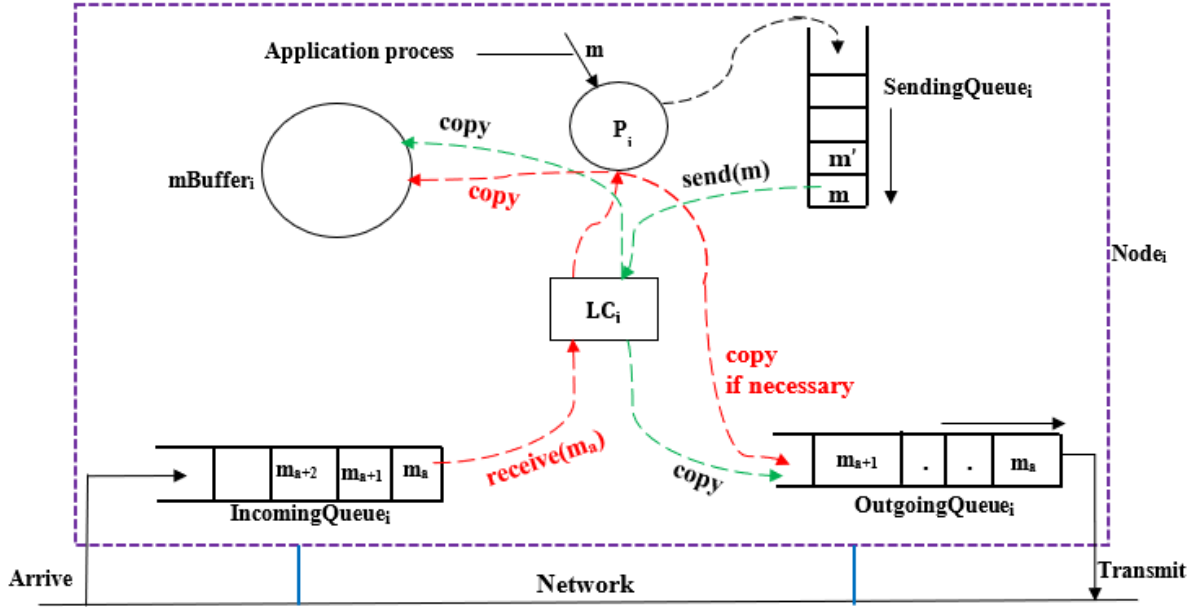


Figure 3.7 Message Sending, Receiving, Forwarding

We assume a thread called $send(m)$ which, whenever permitted by transmission control policy, dequeues m at the head of non-empty $SendingQueue_i$, timestamps it with the value of LC_i , (see Figure 3.7) as $m_{ts} = LC_i$, and increments LC_i by 1. It then enqueues the timestamped m into $OutgoingQueue_i$ for a transmission as shown in Figure 3.7. A copy of timestamped m is also deposited in a buffer called $mBuffer_i$.

Similarly, we receive a thread called $receive(m)$ which whenever permitted by transmission control policy, dequeues m from $IncomingQueue_i$, sets $LC_i = \max \{(m_{ts} + 1), LC_i\}$, and then returns m to P_i which will process m as per the algorithm. Typically, m is entered in $mBuffer_i$ and may be forwarded if necessary to CN_i by entering a copy of m with destination set to CN_i into the $OutgoingQueue_i$. The condition "if necessary" indicates that a copy of m is forwarded only when m has not completed one full cycle within the ring network as will be explained later. Consequently, once m completes one full cycle, the forwarding of m is not needed and m forwarding stops.

It is important to highlight that when P_i receives m and forwards it, the timestamp of m , m_ts , remains unchanged and also it remains in the *IncomingQueue_i*. Moreover, if two messages are received back-to-back, they will be sent in the order they were received but not necessarily back-to-back if transmission control policy dictates so. In addition, as shown in Figure 3.7, messages to be received arrive at the *IncomingQueue_i* from the network, and a copy of the received message arrives at the *mBuffer_i* while a copy of the forwarded messages appears in the *OutgoingQueue_i* according to the order they were received.

Properties

The following properties hold as deduced from the diagram in Figure 3.7:

- (i) If two messages m and m' are sent by P_i in the order: m before m' then $m_ts < m'_ts$.
- (ii) If two received messages m_a and m_{a+1} are forwarded by the process P_i . This means that the messages are forwarded in the received order and this implies that m_a was received before m_{a+1} . In some situations, a message may not be forwarded. However, in cases where m_a and m_{a+1} are both forwarded then m_a will be transmitted first, followed by m_{a+1} and this is FIFO in our system model. FIFO means that each destination receives sent or forwarded messages in the transmitted order.

Currently, our transmission control policy permits that sending a message takes precedence over receiving a message when both *SendingQueue_i* and *IncomingQueue_i* are non-empty (see Assumption AS2). So, if a process P_i has 10 messages in both *SendingQueue_i* and *IncomingQueue_i*, then it sends all the messages sequentially without consideration of the messages in the *IncomingQueue_i* not yet received. After which, P_i proceed to receive the messages that arrive in its *IncomingQueue_i* during the send operation. This policy is referred to as a *greedy sending approach*. However, this approach may have some problems with a process having a message sending priority over other processes, but we are dealing with using fairness control primitive in Section 3.7.

The DCTOP system model discussed thus far operates under the assumption of no crashes. In the event of a crash, the ring model is discarded, and the system resorts to conventional network communication. Consequently, Raft can be employed to reconfigure the system back to a ring structure.

3.3.2 Significance of Timestamp Stability

When P_i receives m , with m_{ts} it cannot TO deliver m until it knows that m_{ts} is stable. This is because TO delivery is an irreversible operation. For instance, assuming P_i receives m with $m_{ts} = 5$ and TO deliver it and later on receives m with $m_{ts} = 4$ then P_i cannot undeliver $m_{ts} = 5$. Therefore, for m to be TO delivered by P_i , then P_i must know that m_{ts} is stable. When m_{ts} is stable in P_i we will also refer to m being stable in P_i .

Deducing Stability:

Informally, when a message with $m_{ts} = TS$ has been received by every process in the ring network then $m_{ts} = TS$ becomes stable. P_i can deduce the stability of TS in two ways

- (i) By P_i receiving a message m with $m_{ts} = TS$ originating from its CN_i , that is, $m_{origin} = CN_i$, P_i can now deduce m_{ts} is stable for it, or
- (ii) By P_i receiving an indication from ACN_i over the FIFO link that informs P_i that TS is stable, that is, P_i knows that some process $P_k = ACN_{m_{origin}}$ has received the message m with m_{ts} and P_k disseminated this stability knowledge to other processes, P_i inclusive, along the ring using $\mu(m)$.

3.3.3 Crashproofing of Messages

In our system model, if f processes crash, the system can still be operative since we assumed $N=2f+1$. However, even if a single process crashes, it necessitates the removal of that process and a restructuring of the ring network. To achieve this ring restructuring, we can use the group membership protocol of Raft [3]. Raft protocols require that at most f processes can crash which is what we assumed. However, the rationale for crashproofing is that when we have at least $f + 1$ processes that have received a given message m even if f of them crash there will be at least one process that can be relied on in sending m to others and this emphasizes the importance of crashproofness in our system. Thus, if P_j receive a message m whose origin is P_i then if $Hops_{i,j}$ is equal to at least f , $Hops_{i,j} \geq f$, then P_j knows that m is crashproof and m can be delivered if m timestamp is known to be stable even if ACN_i has not receive m . This can happen because of concurrent messaging using LC. In the rest of the thesis, we excluded the consideration of crashes. This decision is based on the assumption that in the event of a crash, Raft and Viewstamp [129] group membership will be used to manage the crash, aligning with our assumption and the presence of crashproofness. These two reasons offer sufficient grounds for completely overlooking the impact of crashes in our analysis.

Delivery Requirements

Any message m can be delivered to the high-level application process by P_i if satisfies the following two requirements:

- (i) m_{ts} must be stable in P_i
- (ii) m must be crashproof in P_i , and
- (iii) Any two stable and crashproofed m , and m' are delivered in total order: m is delivered before m' iff $m_{ts} < m'_{ts}$ or $m_{ts} = m'_{ts}$ and $m_{origin} > m'_{origin}$.

During the delivery of m , if $m_{ts} = m'_{ts}$ then the messages are ordered according to the origin of the messages, usually a message from P_{N-1} are ordered before a message from P_i where $N - 1 > i$.

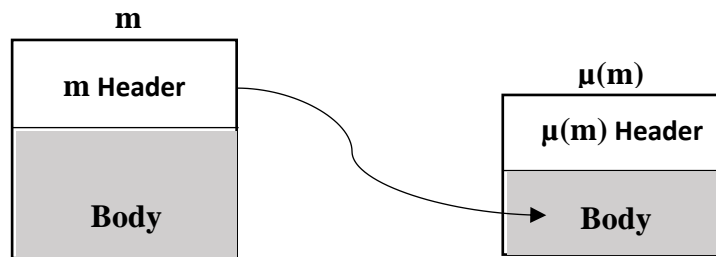
3.3.4 Data Structures

In this section, we discuss the data structures associated with each process, the message, and the μ message as used in our system simulation experiment:

Each process P_i has the following data structures:

1. **Logical clock** (LC_i): This is an integer object initialized to zero. It is used to timestamp messages.
2. **Stability clock** (SC_i): This is an integer object that holds the largest timestamp, ts , known to P_i as stable. Initially, SC_i is zero.
3. **Message Buffer** ($mBuffer_i$): This field holds the sent or received messages by P_i .
4. **Delivery Queue** (DQ_i): Messages waiting to be delivered are queued in this queue object.
5. **Garbage Collection Queue** (GCQ_i): After a message is delivered, the message is transferred to GCQ_i to be garbage collected.

We use M to denote all types of messages used by our protocol. Usually there two types of M : data message denoted by m , and an announcement or ack message that is bound to a specific data message. The latter is denoted as $\mu(m)$ when it is bound to m . $\mu(m)$ is used to announce that m has been received by all processes in Π . The relationship between m and its counterpart $\mu(m)$ is shown below.

Figure 3.8 The Relationship between m and $\mu(m)$

A message, m , consists of a header and a body, with the body containing the data application information. Every m has a corresponding μ , denoted as $\mu(m)$, which contains the information from m 's header. This is why we refer to $\mu(m)$ instead of just μ . $\mu(m)$ has m header information as its main information and does not contain its own data; therefore, the body of $\mu(m)$ is essentially m 's header (see Figure 3.8).

A message M has at least the following data structures:

1. **Message origin** (M_origin) field shows the id of the process in $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ that initiated the message multicast.
2. **Message timestamp** (M_ts) field holds the timestamp given to M by M_origin .
3. **Message destination** (M_destn) field holds the destination of M which is the CN of the process that sends/forwards M .
4. **Message flag** (M_flag) it is a Boolean field which can be true or false and is initiated to be false when M is formed.

In general, it is worth noting that while our system model incorporates fault tolerance mechanisms designed to handle up to f failures, ensuring stability and consistency in the presence of crashes. However, the current experimental setup in this work (see Section 4.2.1) does not simulate failures. Instead, it focuses on evaluating system performance under crash free conditions, which is a necessary first step in understanding baseline behaviours. While this choice limits our ability to fully validate the model's fault tolerance capabilities, we plan to extend the evaluation to include failure scenarios in future experiments.

3.4 Stepwise Design of DCTOP

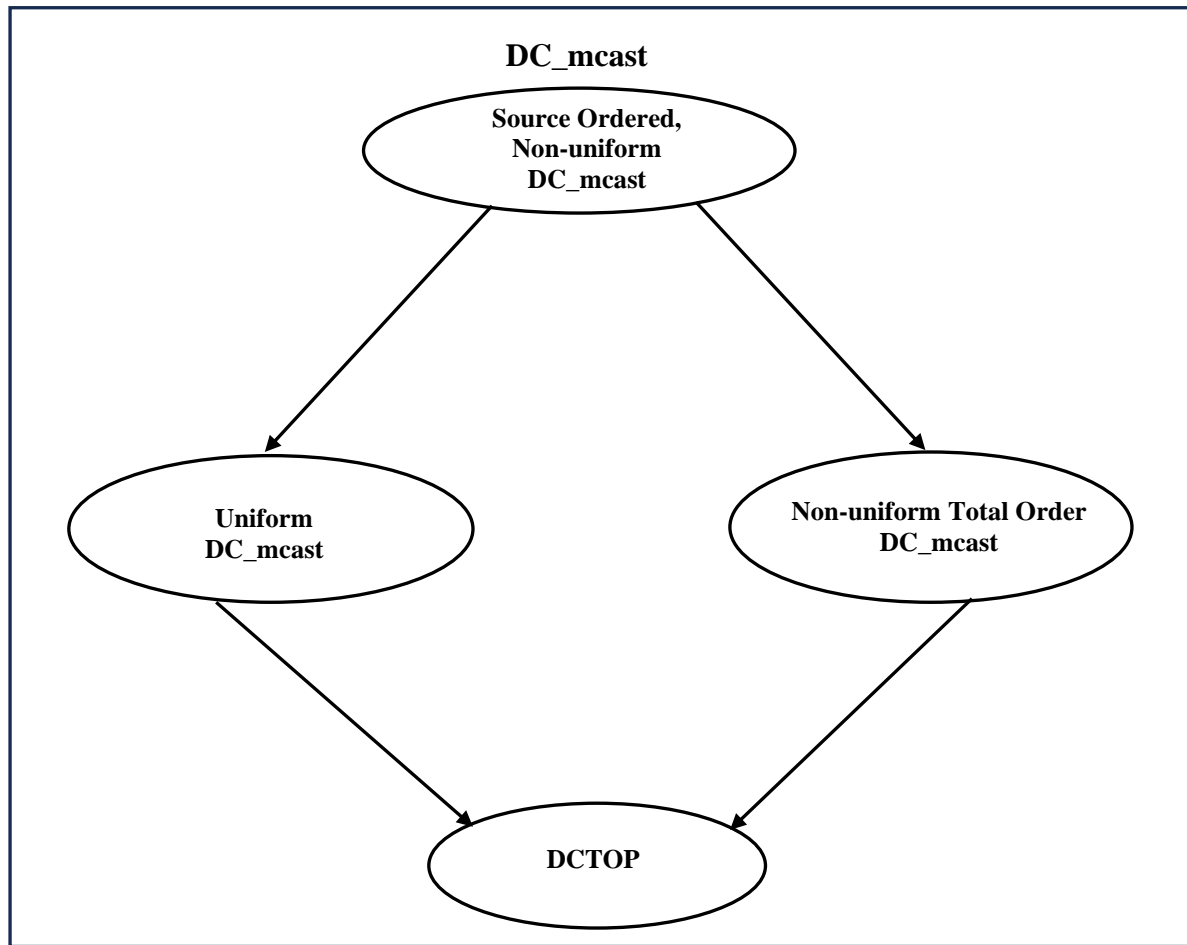


Figure 3.9 DCTOP High Level Framework

We first develop a primitive known as *source-ordered* non-uniform daisy chain multicast (DC_mcast) for sending messages from one process to all processes within a ring structure (see Section 3.2). Source order is defined as the preservation of the sequential order of messages transmitted from a single source process to destination processes across a communication network [145]. If a source, say P_i , sends two messages m and m' in that order, all processes deliver m before m' .

Secondly, we extend the non-uniform source-ordered DC_mcast to uniform source-ordered DC_mcast. Also, we develop a non-uniform total order DC_mcast. For example, in uniform DC_mcast, a process is prohibited from delivering any message out of order, even in the presence of crash processes. Conversely, non-uniform total order DC_mcast only applies to correct processes and therefore does not impose any restrictions on the behaviour of faulty processes. Finally, we combine uniform source-order DC_mcast and non-uniform total order DC_mcast to develop DCTOP- a uniform and total order multicast as shown in Figure 3.9.

3.4.1 Non-Uniform DC_mcast

DC_mcast Features:

Feature 1. In DC_mcast, only source-ordered delivery is assured; different processes can deliver messages with distinct origins in different order.

Feature 2. Delivered messages can be lost during crash recovery (non-uniform delivery).

Pseudocode for (Source Ordered, non-uniform) DC_mcast

DC_mcast has one primitive for every $P_i \in \Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ known as $DC_mcast_i(m)$, use for initiating a non-uniform multicasting of m and one thread that delivers m to the application process.

Part 1: for initiating DC_mcast

```
DC_mcasti(m)
{
    m_origin = Pi;
    m_destn = CNi
    enqueue  $m$  in SendingQueuei;
    deliver(m);
}
```

Part 2: for delivery

```
SO-Deliveryi    // Source order delivery
do {
    receive(m);
    Pj = m_origin;
    deliver(m);
    If Pj ≠ CNi then
    {
        m_destn = CNi
        enqueue  $m$  in OutgoingQueuei;
    }
    else
    {
        discard  $m$ ;
    }
}
forever
```

When a high-level application process generates a message, m and request the protocol process P_i to DC_mcast m within the cluster. P_i then invokes $DC_mcast_i(m)$ primitive. Subsequently, P_i enqueues m into the *SendingQueue_i*. It then uses send(m) thread to dequeue m at the head

of non-empty *SendingQueue_i*, sets two fields in the headers for m , $m_origin=P_i$, and $m_destn=CN_i$ for transmission and delivers it. Also, whenever a $receive(m)$ thread returns a message m and $P_j=m_origin$, it delivers it. P_i then check if $P_j \neq CN_i$. If this is true, it sets $m_destn=CN_i$ and enqueues m into the *OutgoingQueue_i*; otherwise, it discards m .

However, DC_mcast does not support uniform delivery (Feature 2). Suppose as shown in Figure 3.10, P_0 delivers its m_0 , and m_0' after sending and crashes. What if P_1 receives m_0 , delivers it and crashes before receiving m_0' . P_0 is the only process to deliver both m_0 , and m_0' , but P_2 , P_3 , and P_4 does not even know about m_0' . Therefore, DC_mcast supports only non-uniform delivery.

Now, imagine that P_0 DC_mcast m_0, m_0' and P_1 DC_mcast m_1, m_1' :

- P_0 delivers m_0, m_0' at the time of sending.
- P_1 delivers m_1, m_1' because P_1 was sending own messages at the same time with P_0 .

When P_1 receives m_0 , and m_0'

- P_1 delivers m_1, m_1', m_0 , and m_0' in that order.

When P_2 receives m_1, m_1' from P_1 and m_0, m_0' forwarded by P_1

- P_2 delivers m_1, m_1', m_0 , and m_0' .

When P_3 receives m_1, m_1', m_0 , and m_0' forwarded by P_2

- P_3 deliver s m_1, m_1', m_0 , and m_0' .

This message deliveries at P_0, P_1, P_2 , and P_3 validates Feature 1 of DC_mcast.

Another possibility is that P_4 might DC_mcast m_4 before receiving messages forwarded by P_3 .

- Then P_4 delivers m_4, m_1, m_1', m_0 , and m_0' in that order.

Now suppose that P_0 receives m_4 before receiving the messages from P_1 then:

- P_0 delivers m_0, m_0', m_4, m_1 , and m_1' because P_0 receives m_4 before m_1 , and m_1' .

Hence, in DC_mcast there is source order delivery but not total order delivery (Feature 1).

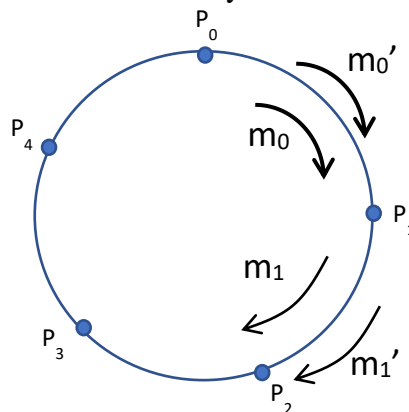


Figure 3.10 Daisy Chain Multicast Design

Consequently, both Features 1 and 2 in non-uniform DC_mcast need to be eliminated in the design of DCTOP (see Figure 3.9). We first address the second challenge due to non-uniform delivery by ensuring that no process delivers m until and unless it knows m is crashproofed: m is received by at least $(f + 1)$ processes.

3.4.2 Uniform DC_mcast

Unlike in non-uniform DC_mcast, when P_i receives m with $m_origin = CN_i$ it constructs $\mu(m)$ (see Section 3.3.4); specifically, it contains information such as m_ts , m_origin as defined in Section 3.3.4. $\mu(m)$ announces that m has completed a full cycle. For simplicity, we will say $\mu(m)$ acks m and m is acked by $\mu(m)$. P_i sends $\mu(m)$ to its CN_i .

Any P_j that receives $\mu(m)$, $\mu(m)_origin = P_k$ (say),

- P_j forwards $\mu(m)$ to CN_j if $Hops_{j,k} < f$.
- P_j stops forwarding $\mu(m)$ if $Hops_{j,k} = f$.

P_i deduces the crashproof status of m in one of two ways:

- (i) When it receives m such that $m_origin = k$ and $Hops_{i,k} \geq f$
- (ii) When it receives $\mu(m)$.

Any m that has made at least f hops has been received by at least $f + 1$ processes and receiving $\mu(m)$ indicates that m has been received by all processes.

Uniform DC_mcast Algorithm Main Points

1. When P_i uniform_mcast m , it deposits m in its $mBuffer_i$, and $mBuffer_i$ is as defined in Section 3.3.4 except that all received m are deposited in $mBuffer_i$.
2. When P_i receives m
 - If it is crashproofed, it is delivered.
 - If it is not crashproofed, it is entered in $mBuffer_i$
3. When P_i receives $\mu(m)$, it removes m from $mBuffer_i$ and delivers it.

It is important to note that the uniform_mcast protocol maintains source-ordered delivery: if P_i initiates a uniform_mcast(m) and then uniform_mcast(m')

- Every process receives m and then m' .
- $\mu(m)$ will be formed and sent before $\mu(m')$.
- Any process that receives both $\mu(m)$ and $\mu(m')$ will receive $\mu(m)$ and then $\mu(m')$.

Uniform DC_mcast Pseudocode

Uniform DC_mcast has one primitive for every $P_i \in \Pi$, $\text{Uniform_mcast}_i(m)$, for initiating a uniform multicasting of m and another thread that delivers m to the application process.

$\text{Uniform_mcast}_i(m)$ // **Part 1: for uniform multicasting**

```
{
    m_origin =  $P_i$ ;
    m_destn =  $CN_i$ 
    enqueue  $m$  in SendingQueuei;
    deposit a copy of  $m$  in mBufferi
}
```

Uniform – Delivery _{i} // **Part 2: for uniform delivery**

```
do forever {
    receive( $M$ );  $P_j = m\_origin$ ;
    If  $M = m$  then
    {
        If  $Hops_{i,j} \geq f$  then
        {
            deliver  $m$ 
        }
        else
        { deposit  $m$  in mBufferi }
        If  $P_j = CN_i$  then
        {
            discard  $m$ ;
            construct  $\mu(m)$ ;
             $\mu(m)\_destn = CN_i$ 
            enqueue  $\mu(m)$  in OutgoingQueuei
        }
        else {  $m\_destn = P_j$ ;
            enqueue  $m$  in OutgoingQueuei }
    }
    If  $M = \mu(m)$  then
    {
         $P_j = \mu(m)\_origin$ ;
        If  $Hops_{i,j} < f$  then
        {
            remove  $m$  from mBufferi;
            deliver  $m$ ;
             $\mu(m)\_destn = CN_i$ 
            enqueue  $\mu(m)$  in OutgoingQueuei
        }
        } else { discard  $\mu(m)$  }
    }
} // do forever ends
```

When a high-level application process generates a message, m and requests the protocol process P_i to uniform_mcast m within the cluster. P_i then invokes $\text{Uniform_mcast}_i(m)$ primitive. Subsequently, P_i enqueues m into the SendingQueue_i . It then uses $\text{send}(m)$ thread to dequeue m at the head of the non-empty SendingQueue_i , sets two fields in the headers for m , $m_origin = P_i$, and $m_destn = CN_i$ for transmission and deposit a copy of m in $mBuffer_i$. However, whenever a $\text{receive}(M)$ thread returns a message M in P_i there are two possibilities: M can be m or M can be $\mu(m)$. When P_i receives M , $M = m$, and suppose that $P_j = m_origin$: P_i checks if $\text{Hops}_{j,i} \geq f$. if this is true then P_i delivers m , otherwise if $\text{Hops}_{j,i} < f$, P_i enters a copy of m into $mBuffer_i$ because m is not yet crashproof. Subsequently, it checks if $P_j = CN_i$ and then P_i discards m , constructs $\mu(m)$, setting $\mu(m)_destn = CN_i$. After which P_i enqueues $\mu(m)$ into the OutgoingQueue_i ; on the other hand, if $P_j \neq CN_i$, P_i enqueues m into the OutgoingQueue_i after setting $m_destn = CN_i$. However, if $M = \mu(m)$ and $P_j = \mu(m)_origin$. Then if $\text{Hops}_{j,i} < f$, P_i removes m from $mBuffer_i$ and delivers it. It then sets $\mu(m)_origin = CN_i$ and enqueue $\mu(m)$ in OutgoingQueue_i ; otherwise, if $\text{Hops}_{j,i} > f$, P_i discard $\mu(m)$.

Consequently, we show the role of $\mu(m)$ in a uniform DC_mcast using Figure 3.11.

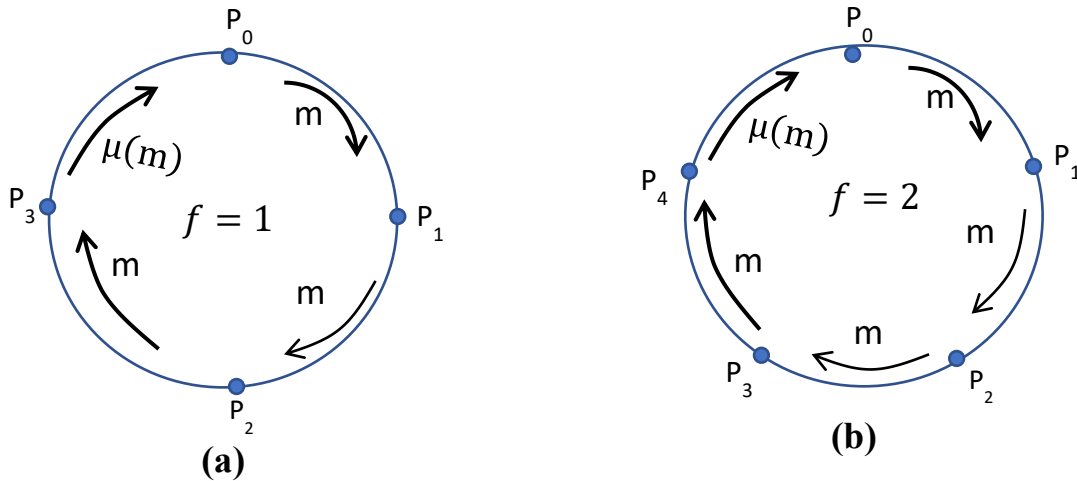


Figure 3.11 The role of $\mu(m)$ in uniform DC_mcast

As shown in Figure 3.11a and considering $f = 1$: if P_0 initiates a $\text{uniform_mcast}(m)$, it deposits a copy of m in $mBuffer_0$ and does not deliver it as m is not yet crashproof. When P_1 receives

m and because $f = 1$, it delivers m since m is crashproof: m has been received by two processes, $\text{Hops}_{0,1} = 1$. When P_2 receives m , it delivers it since m is crashproof, $\text{Hops}_{0,2} > f$. Similarly, when P_3 receives m , it delivers it and constructs $\mu(m)$. It then $\text{uniform_mcast}(\mu(m))$ to P_0 . When P_0 receives $\mu(m)$, it removes m from its mBuffer_0 and delivers it. Thus P_0 announces that every process has received m , making it crashproofed in P_0 . P_0 does not have to propagate $\mu(m)$.

Similarly, as shown in Figure 3.11b and considering $f = 2$: if P_0 initiates a $\text{uniform_mcast}(m)$, it deposits a copy of m in mBuffer_0 as m is not yet crashproof. When P_1 receives m , it deposits a copy of m in mBuffer_1 because m is not yet crashproof, $f = 2$, meaning that at least 3 processes need to receive m before m can be termed crashproofed. When P_2 receives m , it delivers it since m is crashproof, $\text{Hops}_{0,2} \geq f$. Also, when P_3 receives m , it delivers m , $\text{Hops}_{0,3} \geq f$. Finally, when P_4 receives m , it delivers it and constructs $\mu(m)$. It then $\text{uniform_mcast}(\mu(m))$ to P_0 . When P_0 receives $\mu(m)$, it removes m from its mBuffer_0 and delivers it. Also, when P_1 receives $\mu(m)$, it removes m from its mBuffer_1 and delivers it, after which P_1 discard $\mu(m)$ because every process has delivered m .

3.4.3 Non-Uniform, Total Ordered multicast

In this Section, we enhanced non-uniform DC_mcast by incorporating total order with non-uniform delivery. We removed only Feature 1 while still maintaining Feature 2 as described in Section 3.4.1.

Every P_i requires these three data structures defined as follows:

4. **SC_i** : This is as defined in Section 3.3.4. To recall SC_i maintains the largest timestamp m_ts that is known to be stable in P_i .
5. **DQ_i** : This queue contains received m entered in total order: P_i enters m in DQ_i before m' if and only if $m_ts < m'_ts$ or $m_ts = m'_ts$ and $m_origin > m'_origin$.
6. **GCQ_i** : This queue contains delivered messages in total order. When m is stable and is at the head of DQ_i : P_i enters m in GCQ_i before m' if and only if $m_ts < m'_ts$ or $m_ts = m'_ts$ and $m_origin > m'_origin$.

The protocol has three parts: a primitive for every $P_i \in \Pi$, $\text{TO_DC_mcast}_i(m)$, for initiating a non-uniform total order multicasting of m , a thread for receiving m and another thread that delivers m to the application process. These three parts are described as follows:

Part 1: Non-uniform Total Order Multicast InitializationTO_DC_mcast_i(*m*)

```

{
    m_origin = Pi;
    m_destn = CNi;
    m_flag = false;
    enqueue m in SendingQueuei;
}

```

When a high-level application process generates a message, *m* and requests the protocol process P_i to TO_DC_mcast *m* within the cluster. P_i then invokes TO_DC_mcast_i(*m*) primitive. Subsequently, P_i initialises *m* as, m_origin = P_i, m_destn = CN_i, and m_flag = false and enqueues *m* into the *SendingQueue_i*. It then uses send(*m*) thread to dequeue *m* at the head of non-empty *SendingQueue_i*, timestamp *m*, m_ts = LC_i, and transmit *m* to CN_i. The send(*m*) thread ensures that consecutive messages are formed and sent by P_i has an increasing timestamp. TO_DC_mcast_i(*m*) is similar to Uniform_mcast_i(*m*) except that it includes m_flag initially set to false.

Part 2: Processing a received M

```

{
    receive(M); Pj = M_origin;
    If M = m
    {
        m_flag = true;
        deposit copy of m in mBufferi;
        // check to forward m or not
        If Pj ≠ CNi
        {
            m_destn = CNi;
            enqueue m in OutgoingQueuei;
        }
    }
    else
    {
        SCi = max { SCi, m_ts }
        remove all m, m_ts ≤ SCi from mBufferi and enqueue them into DQi in
        total order;
        form μ(m) with
        μ(m)_origin = Pi; and
        μ(m)_destn = CNi;
        enqueue μ(m) in OutgoingQueuei;
    }
}
// end if M = m

```

```

If  $M = \mu(m)$ 
{
    // meaningful  $\mu(m)$  or not
    If ( $m \in \mu(m)$ ):  $m\_ts \leq SC_i$ )
    {
        discards  $\mu(m)$ ;
    }
    else
    {
        //  $\mu(m)$  is meaningful
         $SC_i = m\_ts$ ;
        remove all  $m$ ,  $m\_ts \leq SC_i$  from  $mBuffer_i$  and enqueue them into  $DQ_i$  in
        total order;
        // to forward  $\mu(m)$  or not?
        If  $P_j = CN_i$ 
        {
            discards  $\mu(m)$ ;
        }
        else
        {
             $\mu(m)\_destn = CN_i$ ;
            enqueue  $\mu(m)$  in  $OutgoingQueue_i$ ;
        }
    }
    } // end of meaningful  $\mu(m)$ 
} // end of  $M = \mu(m)$ 
} // end of receive( $M$ )

```

Recall that $receive(M)$ returns M to P_i after adjusting LC_i if needed. This pseudocode describes how a received M is processed by P_i and also recall that $\{m, \mu(m)\} \in M$. When P_i receives M and if $M = m$, assuming $P_j = m_origin$, P_i sets $m_flag = true$ and deposits a copy of m in $mBuffer_i$. P_i checks if m can be forwarded or not: if $P_j \neq CN_i$ then it sets $m_destn = CN_i$ and enqueues m in $OutgoingQueue_i$ otherwise if $P_j = CN_i$ then it updates SC_i as $SC_i = \max \{ SC_i, m_ts \}$, removes all m , $m_ts \leq SC_i$ from $mBuffer_i$ and enqueues them into DQ_i in total order. P_i then forms $\mu(m)$, sets two fields in the headers for $\mu(m)$, $\mu(m)_origin = P_i$, and $\mu(m)_destn = CN_i$ and enqueues $\mu(m)$ in $OutgoingQueue_i$.

On the other hand, when P_i receives M and if $M = \mu(m)$, then P_i checks if $\mu(m)$ is meaningful or not. If $m \in \mu(m)$ is such that $m_ts \leq SC_i$ then $\mu(m)$ is not meaningful and $\mu(m)$ is discarded. However, if $m \in \mu(m)$ is such that $m_ts > SC_i$ then $\mu(m)$ is meaningful, then P_i sets $SC_i = m_ts$ and removes all m , $m_ts \leq SC_i$ from $mBuffer_i$ and enqueue them into DQ_i in total order. After which, P_i checks if it can forward $\mu(m)$ or not. If $P_j = CN_i$ then P_i discard $\mu(m)$ otherwise it sets $\mu(m)_destn = CN_i$ and enqueues $\mu(m)$ in $OutgoingQueue_i$.

Part 3: TO – Delivery

```

do {
  m = head (DQi)
  while(m_flag = true) do
    {
      dequeue m from DQi
      deliver m to application process
      enter a copy of m in GCQi
    } // end while
  }
forever

```

Whenever DQ_i is non-empty, P_i dequeues *m* from the head of DQ_i and delivers *m* to application process. P_i then enters a copy of *m* into GCQ_i to represent a successful TO delivery. This action is repeated until DQ_i is empty.

Observations:

- (i) When P_i delivers *m* and the next message *m'* in DQ_i is such that *m'*_{ts} > *m*_{ts} then P_i can conclude that all messages with timestamps less than or equal to *m*_{ts} have been delivered.
- (ii) GCQ_i maintains delivered messages in their total order.

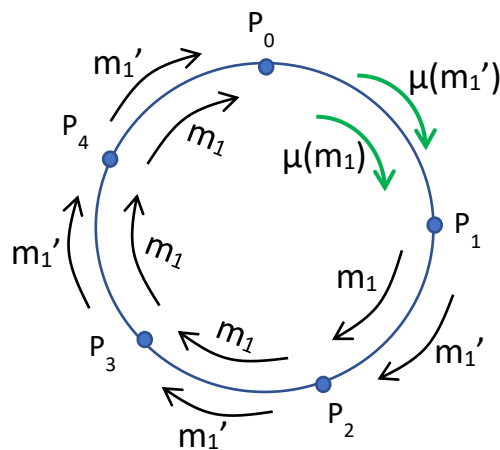


Figure 3.12 Non-uniform Total Order Multicast

Suppose as shown in Figure 3.12, P₁ TO_DC_mcast *m*₁, and *m*₁'. It is worth noting that in TO_DC_mcast when a process sends or receives *m*, it does not deliver *m* until it receives μ(*m*) or when ACN of *m*_{origin} receives *m*. Thus, when P₂ receives *m*₁, and *m*₁', it forwards *m*₁, and *m*₁' to P₃ since *m*₁_origin ≠ CN₂ and *m*₁'_origin ≠ CN₂. This forwarding of *m*₁, and *m*₁'

continues until P_0 receives them. When P_0 receives m_1 , and m_1' , it knows that both messages are stable because $m_{1_origin} = CN_0$ and $m_{1'_origin} = CN_0$.

- P_0 delivers m_1 , and m_1' in that order.

Therefore, P_0 constructs $\mu(m_1)$, and $\mu(m_1')$ and sends to P_1 . Now, suppose P_0 crashes after sending $\mu(m_1)$, $\mu(m_1')$ and what if P_1 receives $\mu(m_1)$, delivers m_1 and crashes before receiving $\mu(m_1')$. P_0 is the only process to deliver both m_1 , and m_1' , but P_2 , P_3 , and P_4 does not even know about $\mu(m_1')$. Therefore, TO_DC_mcast supports only non-uniform total order delivery.

3.5 DCTOP: Uniform and Total Order Multicast

As shown in Figure 3.9, we integrated both the uniform DC_mcast and the non-uniform total order DC_mcast to develop DCTOP including the garbage collections.

DCTOP Features:

1. In DCTOP, total ordering is assured; different processes deliver messages with distinct origin in the same order in addition to source ordering.
2. Delivered messages cannot be lost due to crashes (uniform total order delivery).

3.5.1 DCTOP Algorithm Main Points

The DCTOP algorithm main points deduced from the combination of uniform DC_mcast and non-uniform total order DC_mcast are as follows:

1. When P_i forms and sends m , it sets $m_flag = false$ before it deposits m in its $mBuffer_i$.
2. When P_i receives m and $P_j = m_origin$
 - It checks if $Hops_{j,i} \geq f$. If this is true then m is crashproofed, it does not deliver m immediately. Moreover, it sets $m_flag = true$ and deposits m in its $mBuffer_i$. if m is not crashproofed, then m_flag remains false.
 - It then checks if $P_j \neq CN_i$. if this is true, it sets $m_destn = CN_i$ and deposits m in its $OutgoingQueue_i$,
 - Otherwise, m is stable then it updates SC_i as $SC_i = \max \{ SC_i, m_ts \}$, and transfer all m , $m_ts \leq SC_i$ to DQ_i . Then, it forms $\mu(m)$, sets the two header fields, $\mu(m)_origin = P_i$, $\mu(m)_destn = CN_i$ and deposit $\mu(m)$ in $OutgoingQueue_i$.
3. When P_i receives $\mu(m)$, it knows that every process has received m .
 - If m in $\mu(m)$ does not indicate a higher stabilisation in P_i , that is, $m_ts \leq SC_i$ and $Hops_{j,i} \geq f$ then P_i ignores $\mu(m)$, else, if $Hops_{j,i} < f$, look for m in $mBuffer_i$ or DQ_i , P_i sets $m_flag = true$, $\mu(m)_destn = CN_i$ and deposit $\mu(m)$ in $OutgoingQueue_i$.

- However, if m in $\mu(m)$ indicates a higher stabilisation in P_i , that is, $m_ts > SC_i$, P_i updates SC_i as $SC_i = \max \{ SC_i, m_ts \}$, and transfer all m , $m_ts \leq SC_i$ to DQ_i .
 - If $P_j = CN_i$, P_i ignores $\mu(m)$ otherwise, it sets $\mu(m)_destn = CN_i$ and deposit $\mu(m)$ in *OutgoingQueue_i*.
4. Whenever DQ_i is non-empty, P_i dequeues m from the head of DQ_i and delivers m to application process. P_i then enters a copy of m into GCQ_i to represent a successful TO delivery. This action is repeated until DQ_i becomes empty.

It is important to note that the DCTOP maintains total order. Thus, if P_i forms and sends m and then m'

- Every process receives m and then m' .
- $\mu(m)$ will be formed and sent before $\mu(m')$.
- Any process that receives both $\mu(m)$ and $\mu(m')$ will receive $\mu(m)$ and then $\mu(m')$.

However, a message, m , may be stable but not necessarily crashproof. Therefore, we illustrate the importance of the crashproofness of a message using Figure 3.13. As shown in Figure 3.13a, when P_0 sends m_0 to its CN_0 denoted by the black arrow and assuming $LC_0 = 5$, it sets $m_0_ts = 5$. After sending m_0 , P_0 then increases its LC_0 by 1, $LC_0 > 5$. LC_0 becomes greater than the m_0_ts of the message it sent ($LC_0 > m_0_ts$), $LC_0 = 6 > 5$. When P_1 receives m_0 , it sets $LC_1 > 5$ and forwards m_0 ; when P_2 receives m_0 , it sets $LC_2 > 5$ and forwards m_0 .

Here, we consider a scenario where P_3 sends its own message, m_3 , denoted by the blue arrow before receiving m_0 , then assuming $LC_3 = 4$, $m_3_ts = 4$ and sets $LC_3 = 5 > 4$. When P_3 receives m_0 , it knows that m_0 is stable, that is, the ts of m_0 is stable. P_3 sets $LC_3 = 6 > 5$ since $m_0_ts + 1$ is greater than 5 (see Section 3.3.1). P_3 then prepares, timestamp $\mu(m_0)$ using its LC_3 and sends $\mu(m_0)$ to P_0 denoted by a short black arrow to show that the sending of m_3 happened before the sending of $\mu(m_0)$. When P_0 receives $\mu(m_0)$, it knows that every process has received m_0 then m_0 is stable and crashproof (m_0 has made at least f hops) but P_0 have two messages, m_0 and m_3 , in its $mBuffer_0$ with different TS. The messages are then ordered by their TS. So, at P_0 , m_3 is ordered before m_0 ; however, m_3 can be TO delivered because (i) it is at the head of the DQ_0 , (ii) it is stable even though it has not been received by the ACN_3 because m_3_ts is known to be stable. Remember, when any message, m_0_ts is known to be stable, then any message, $m_3_ts \leq m_0_ts$ is automatically known to be stable. (iii) it is crashproof because $f = 1$ (for an $N = 4$ process cluster) indicating that m_3 have been received

by 2 processes. Additionally, m_0 will be TO delivered also since it is stable and crashproof and is at the head of the DQ_0 after the TO delivery of m_3 .

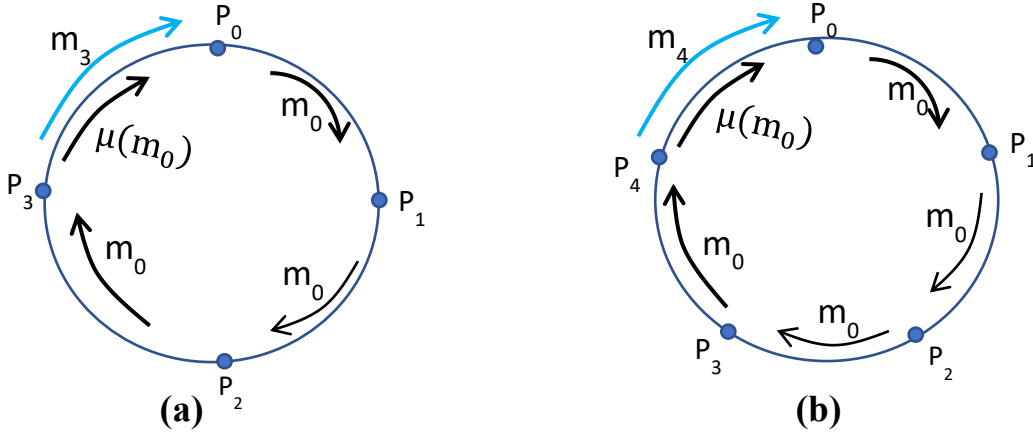


Figure 3.13 Checking the crashproofness of a message

Furthermore, imagine $f = 2$ for a system of $N = 5$ processes as shown in Figure 3.13b. When P_0 sends m_0 to its CN_0 denoted by the black arrow and assuming $LC_0 = 7$, it sets $m_{0_ts} = 7$. After sending m_0 , P_0 then increases its logical clock by 1, $LC_0 > 7$. LC_0 becomes greater than the m_{0_ts} of the message it sent ($LC_0 > m_{0_ts}$), $LC_0 = 8 > 7$. When P_1 receives m_0 , it sets $LC_1 > 7$ and forwards m_0 ; when P_2 receives m_0 , it sets $LC_2 > 7$ and forwards m_0 ; when P_3 receives m_0 , it sets $LC_3 > 7$ and forwards m_0 .

Here, we consider a scenario where P_4 sends its own message, m_4 , denoted by the blue arrow before receiving m_0 , then assuming $LC_4 = 7$, $m_{4_ts} = 7$ and sets $LC_4 = 8 > 7$. When P_4 receives m_0 , it knows that m_0 is stable, that is, the ts of m_0 is stable. LC_4 remains unchanged since $m_{0_ts} + 1$ is equal to 8 (see Section 3.3.1). P_4 then prepares, timestamp $\mu(m_0)$ using its LC_4 and sends $\mu(m_0)$ to P_0 denoted by a short black arrow to show that the sending of m_4 happened before the sending of $\mu(m_0)$. When P_0 receives $\mu(m_0)$, it knows that m_0 is stable and crashproof but P_0 have two messages, m_0 and m_4 , in its $mBuffer_0$ with equal TS . The messages are ordered by their TS but if the TS are equal, then messages from P_4 takes priority over messages from P_0 . So, at P_0 , m_4 is ordered before m_0 . However, P_0 cannot TO deliver m_4 even though it is at the head of the DQ_0 and stable but it is not crashproof.

It is crucial to note that m_4 needs to be received by at least $(f + 1)$ distinct processes before it becomes crashproof (see Section 3.3.3), that is, m_4 needs to be received by at least 3 processes (for an $N = 5$ process cluster) to become crashproof, in this case, it is only received by 2 processes, P_4 and P_0 . Conversely, m_0 remains undeliverable despite its stability and crashproofness due to m_4 at the head of the DQ_0 in P_0 lacking crashproofness though stable.

3.5.2 DCTOP Pseudocodes

This section mirrors the discussion in Section 3.4.3, with the inclusion of uniform delivery. Similar to TO_DC_mcast, DCTOP also has three parts: a primitive for every $P_i \in \Pi$, $DC_TOP_i(m)$, for initiating a uniform total order multicasting of m , a thread for receiving m and another thread that delivers m to the application process.

Part 1: Uniform Total Order Multicast Initialization

```

DC_TOPi(m)
{
    m_origin = Pi;
    m_destn = CNi;
    m_flag = false;
    enqueue  $m$  in SendingQueuei;
}

```

$DC_TOP_i(m)$ operates similarly to the description provided in part 1 of $TO_DC_mcast_i(m)$.

Part 2: Processing a received M

```

{
    receive(M); Pj = M_origin;
    If ( $M = m$ )
    {
        //Check the crashproof status of  $m$ 
        If ( $Hops_{j,i} \geq f$ ) then
        {
            m_flag = true;
        }
        deposit copy of  $m$  in mBufferi;
        // check to forward  $m$  or not
        If  $P_j \neq CN_i$ 
        {
            m_destn = CNi;
            enqueue  $m$  in OutgoingQueuei ;}
        else
        {
            SCi = max { SCi, m_ts}
            remove all  $m$ ,  $m\_ts \leq SC_i$  from mBufferi and enqueue them into DQi in
            total order;
            form  $\mu(m)$  with
             $\mu(m)\_origin = P_i$ ; and
             $\mu(m)\_destn = CN_i$ ;
            enqueue  $\mu(m)$  in OutgoingQueuei ;}
        } // end of if  $M = m$ 
    }
}

```

```

If  $M = \mu(m)$ 
{ // meaningful  $\mu(m)$  or not
  If ( $m$  in  $\mu(m)$ :  $m\_ts \leq SC_i$  and  $Hops_{i,j} \geq f$ )
    { discards  $\mu(m)$ ; }
  else { //  $\mu(m)$  is meaningful
    If ( $Hops_{i,j} < f$ )
      { look for  $m$  in  $mBuffer_i$  or  $DQ_i$ 
        Set  $m\_flag = true$ ;
         $\mu(m)\_destn = CN_i$ ;
        enqueue  $\mu(m)$  in the OutgoingQueuei
      } // end of meaningful  $\mu(m)$  and  $Hops_{i,j} < f$ 
    If ( $m$  in  $\mu(m)$  :  $m\_ts > SC_i$ )
      {  $SC_i = m\_ts$ ;
        remove all  $m$ ,  $m\_ts \leq SC_i$  from  $mBuffer_i$  and enqueue them into  $DQ_i$  in
        total order;
      } // to forward  $\mu(m)$  or not?
      If  $P_j = CN_i$ 
        { discards  $\mu(m)$ ; }
      else {
         $\mu(m)\_destn = CN_i$ ;
        enqueue  $\mu(m)$  in OutgoingQueuei
      } // end of meaningful  $\mu(m)$  and  $m\_ts > SC_i$ 
    } // end of meaningful  $\mu(m)$ 
  } // end  $M = \mu(m)$ 
} // receive( $M$ )

```

When P_i receives M and if $M = m$, assuming $P_j = m_origin$, P_i checks the crashproof status of m . If $Hops_{j,i} \geq f$ then m is crashproof, P_i sets $m_flag = true$ and deposits a copy of m in $mBuffer_i$. P_i checks if m can be forwarded or not. If $P_j \neq CN_i$ then it sets $m_destn = CN_i$ and enqueues m in *OutgoingQueue_i* otherwise if $P_j = CN_i$ then it updates SC_i as $SC_i = \max \{ SC_i, m_ts \}$, removes all m , $m_ts \leq SC_i$ from $mBuffer_i$ and enqueues them into DQ_i in total order. P_i then forms $\mu(m)$, sets two fields in the headers for $\mu(m)$, $\mu(m)_origin = P_i$, and $\mu(m)_destn = CN_i$ and enqueues $\mu(m)$ in *OutgoingQueue_i*.

On the other hand, when P_i receives M and if $M = \mu(m)$, then P_i checks if $\mu(m)$ is meaningful or not. If m in $\mu(m)$ is such that $m_ts \leq SC_i$ and $Hops_{j,i} \geq f$ then $\mu(m)$ is not meaningful and $\mu(m)$ is discarded, otherwise $\mu(m)$ is meaningful. If $Hops_{j,i} < f$, P_i searches for m in $mBuffer_i$ or DQ_i , sets $m_flag = true$, $\mu(m)_destn = CN_i$ and enqueues $\mu(m)$ in *OutgoingQueue_i*. However, if m in $\mu(m)$ is such that $m_ts > SC_i$, then P_i sets $SC_i = m_ts$ and removes all m , $m_ts \leq SC_i$ from $mBuffer_i$ and enqueue them into DQ_i in total order. After which, P_i checks if it can

forward $\mu(m)$ or not. If $P_j = CN_i$ then P_i discard $\mu(m)$, otherwise it sets $\mu(m)_{_destn} = CN_i$ and enqueues $\mu(m)$ in *OutgoingQueue_i*.

Part 3: TO – Delivery

Part 3 of the DCTOP pseudocode is structured similarly to part 3 of the non-uniform total order DC_mcast (refer to Section 3.4.3)

3.5.3 Exemplifying DCTOP Pseudocode Implementation

In this section, we illustrate the implementation of DCTOP pseudocode using a single multicast as shown in Figure 3.14. This is a trivial case as one message is considered here but in the next example, we considered the sending of concurrent messages. When P_0 sends m , denoted by the black arrow labelled 1, it deposits a copy of m into its *mBuffer₀* and *OutgoingQueue₀*.

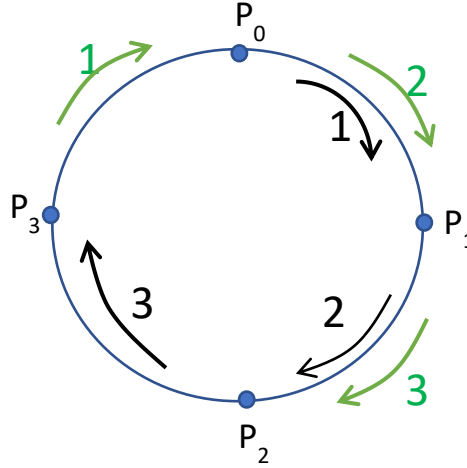


Figure 3.14 Illustrating DCTOP Pseudocodes

When P_1 receives m , it knows that m is crashproof since $Hops_{0,1} = 1$ and $f = 1$ meaning that 2 processes have received m , sets $m_flag = true$ and deposit a copy of m into its *mBuffer₁*. P_1 then deposit m into its *OutgoingQueue₁* since $m_origin \neq CN_1$. When P_2 receives m , it repeats the actions of P_1 . Finally, when P_3 receives m , it knows that m is stable, it updates SC_3 as $SC_3 = \max \{ SC_3, m_ts \}$, transfers m into *DQ₃*. It then forms $\mu(m)$ denoted by the green arrow labelled 1, timestamps it with its LC_3 , sets the two header fields as $\mu(m)_{_origin} = P_3$, $\mu(m)_{_destn} = CN_3$ and enqueues $\mu(m)$ in *OutgoingQueue₃*. P_3 can then attempt TO delivery if m is at the head of *DQ₃* and this is represented by transferring m from *DQ₃* to *GCQ₃*.

However, when P_0 receives $\mu(m)$, if m in $\mu(m)$ is such that $m_ts > SC_0$, then P_0 sets $SC_0 = m_ts$ and removes m , $m_ts \leq SC_0$ from *mBuffer₀* and enqueue it into *DQ₀*. Then P_0 delivers m .

If $\mu(m)_{\text{origin}} \neq \text{CN}_0$, it deposits $\mu(m)$ into its OutgoingQueue_0 . P_1 delivers m following the same action as P_0 when it receives $\mu(m)$. Moreover, when P_2 the ACN of $\mu(m)_{\text{origin}}$ receives $\mu(m)$, it delivers m after following the same action as P_0 . P_2 then discards $\mu(m)$ since $\mu(m)_{\text{origin}} = \text{CN}_2$. Therefore, the latency of m takes at most $2(N - 1)$ hops for a single multicast within the ring structure; $(N - 1)$ hops for m and $(N - 1)$ hops for $\mu(m)$.

Suppose as illustrated in Figure 3.14, we consider a scenario where P_3 potentially sends m_3 just before P_3 receives m , with a timestamp either greater or smaller than m_{ts} . If $m_3_{\text{ts}} \leq m_{\text{ts}}$ then no $\mu(m_3)$ will be generated for m_3 . This is because when the ACN of m_3_{origin} receives m_3 , it knows that m_3_{ts} is already stable. However, if $m_3_{\text{ts}} > m_{\text{ts}}$ then $\mu(m_3)$ will indeed be generated for m_3 when the ACN of m_3_{origin} receives m_3 to communicate m_3 stability.

Concurrent Multicasts with Equal Timestamp

We have established that a latency of $2(N - 1)$ assumes that there is only one multicast within the unidirectional ring framework, but that is not the case for a concurrent multicast. Here, we present concurrent multicast including the dissemination of the stability of the concurrent messages to other processes.

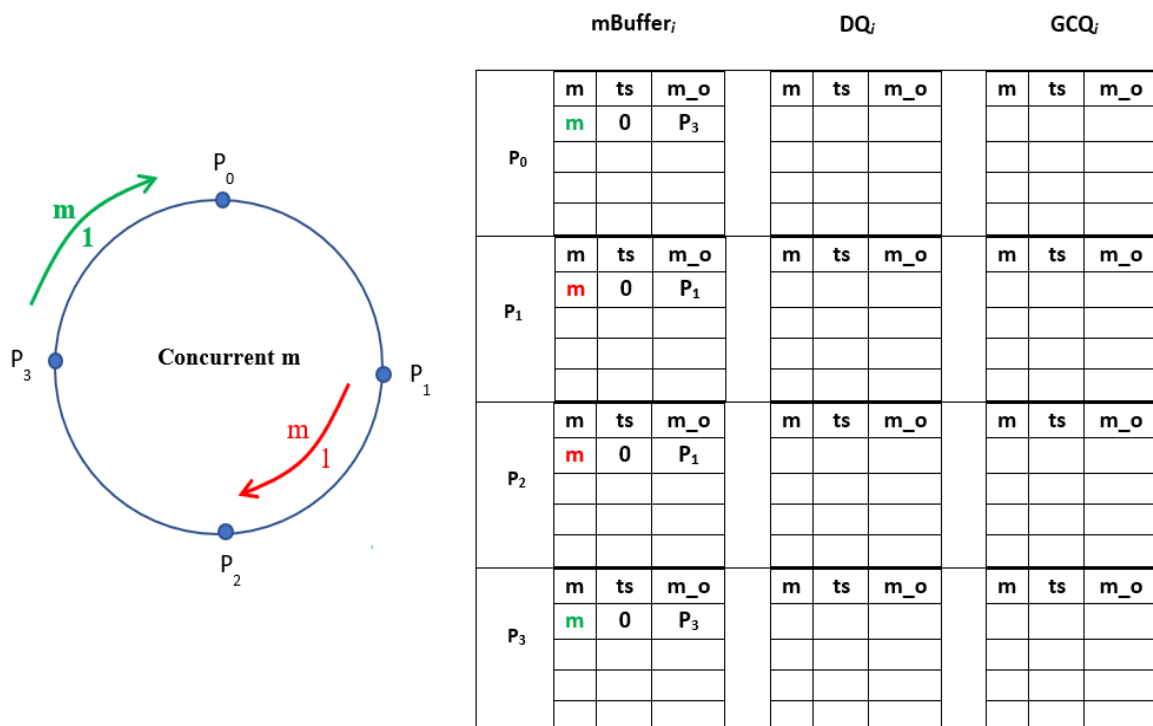


Figure 3.15 Concurrent Multicast from P_1 and P_3

In Figure 3.15, both P_3 and P_1 multicast messages, m and m concurrently with the equal TS, $m_ts=0$ and $m_ts=0$. Following the transmission, P_3 deposits m in its $mBuffer_3$ while P_1 also deposits m in its $mBuffer_1$. Subsequently, P_0 and P_2 received the messages simultaneously after one hop. However, P_0 deposits m in its $mBuffer_0$, while P_2 also deposits m in its $mBuffer_2$.

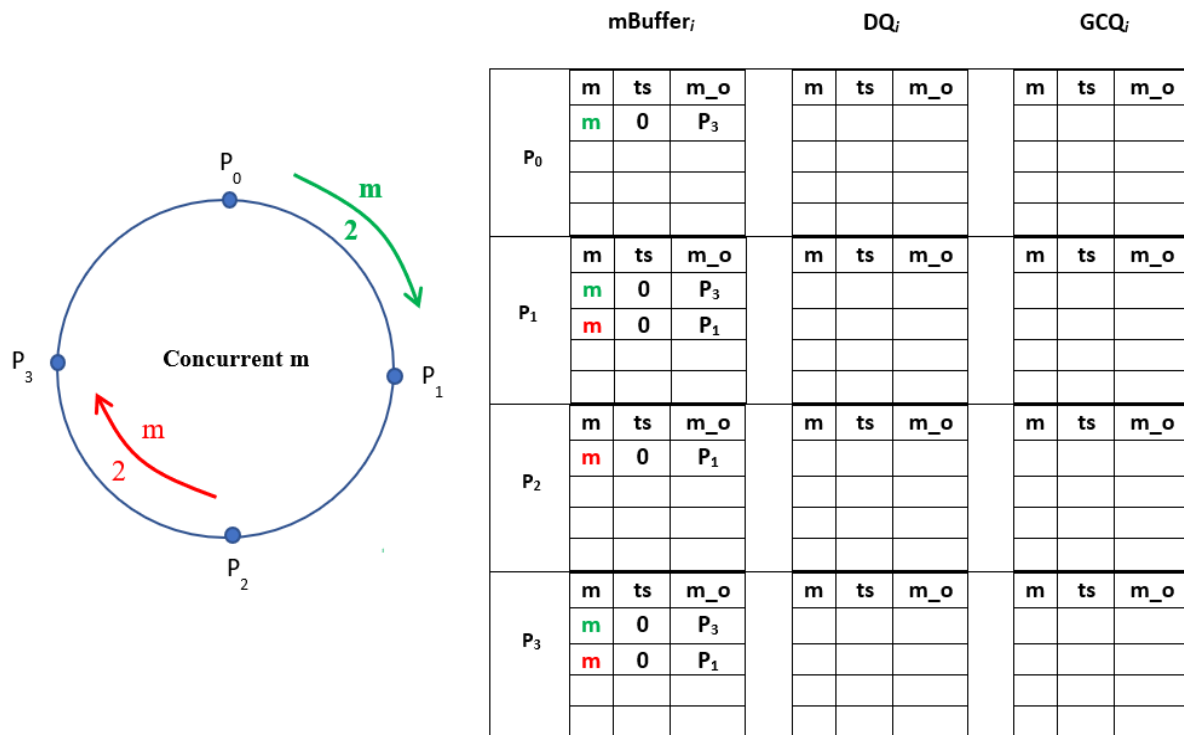


Figure 3.16 P_0 and P_2 Forwards Concurrent Messages from P_1 and P_3 to CN_0 and CN_2

As depicted in Figure 3.16, P_0 forwards a copy of the green message, m , to P_1 whereas P_2 forwards a copy of the red message, m to P_3 . P_3 received the red message, m , and at the same time P_1 receive the green message, m . Moreover, following the receive operation, P_3 deposits m in its $mBuffer_3$ while P_1 also deposits m in its $mBuffer_1$. Thus, P_3 and P_1 received the messages simultaneously after two hops and because both messages have equal TS, then m is ordered before m in both $mBuffer_3$ and $mBuffer_1$ since $P_3 > P_1$. Additionally, m is crashproof since $Hops_{1,3} = 2 > f=1$ and m is also crashproof since $Hops_{3,1} = 2 > f=1$.

Therefore, in Figure 3.17, P_3 forwards m to its CN_3 and P_0 also forwards m to its CN_0 . When P_0 receives m it knows that m is stable, updates its SC_0 as explained earlier, transfers all m , $m_ts \leq SC_0$ from $mBuffer_0$ to DQ_0 . Similarly, when P_2 receives m it knows that m is stable, updates its SC_2 , transfers all m , $m_ts \leq SC_2$ from $mBuffer_2$ to DQ_2 . Therefore, after 3 hops, both P_0 and P_2 delivers m and m to application process simultaneously.

The migration of \bar{m} and \bar{m} to DQ_2 and DQ_0 and finally into GCQ_2 and GCQ_0 represents the successful TO delivery of \bar{m} and \bar{m} , respectively (see Figure 3.17).

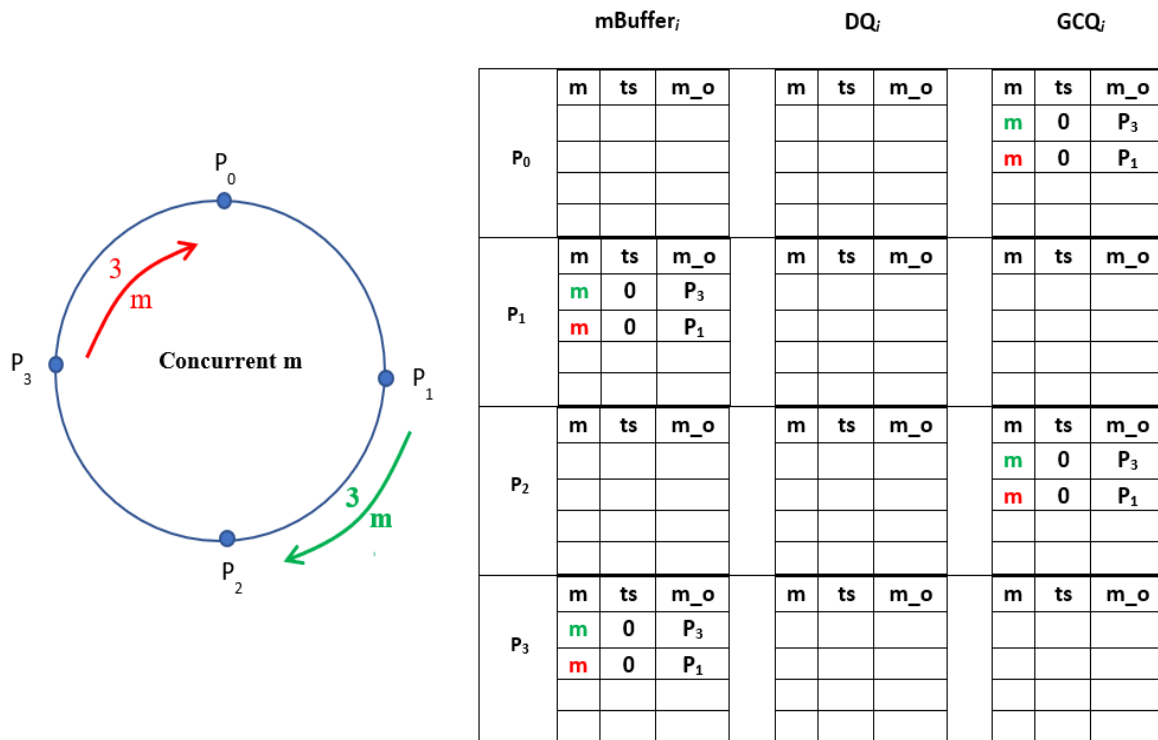
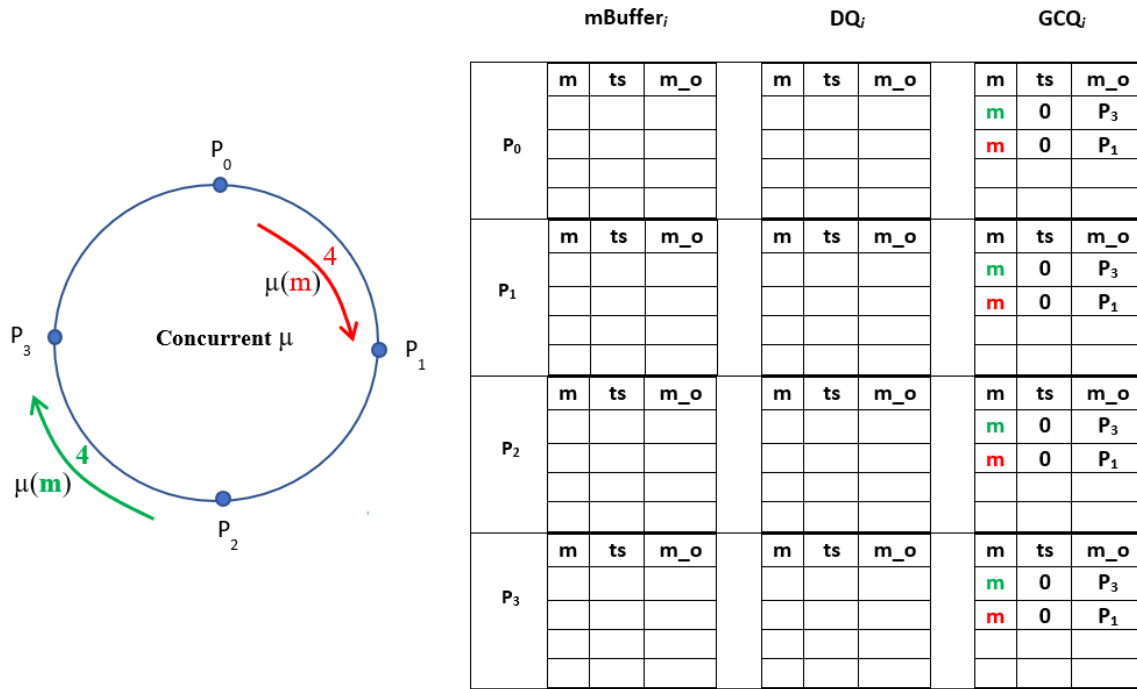
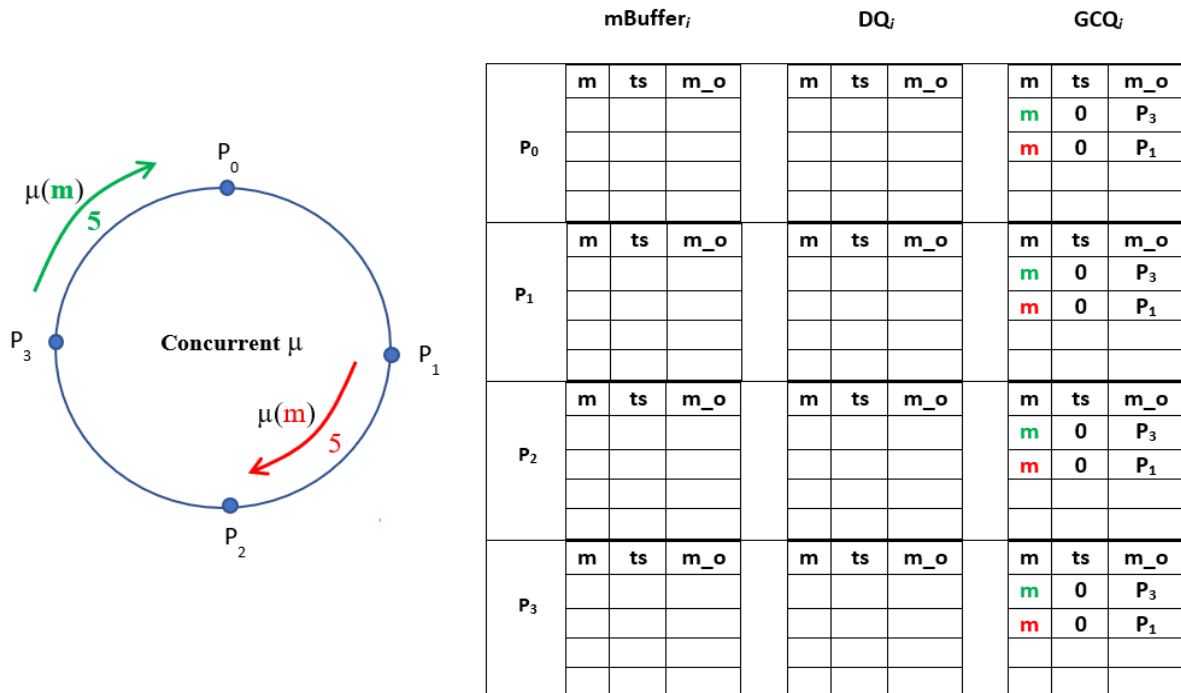


Figure 3.17 P_1 and P_3 Forward Concurrent Messages to CN_1 and CN_3

In Figure 3.18, P_0 forms $\mu(\bar{m})$ and sends it to its clockwise neighbour, CN_0 while P_2 concurrently forms $\mu(\bar{m})$ and sends it to its CN_2 . When P_1 receive $\mu(\bar{m})$, it determines that message \bar{m} has been received by all processes in the cluster. P_1 then updates its stability clock, SC_1 , and transfers all messages m , $m_ts \leq SC_1$ from $mBuffer_1$ to DQ_1 . Similarly, when P_3 receive $\mu(\bar{m})$, it confirms that message \bar{m} has been received by all processes, updates its updates its SC_3 , and transfers all m , $m_ts \leq SC_1$ from $mBuffer_3$ to DQ_3 . Therefore, after four communication hops, both P_3 and P_1 delivers \bar{m} and \bar{m} to application process simultaneously. The migration of \bar{m} and \bar{m} to DQ_3 and DQ_1 and finally into GCQ_3 and GCQ_1 represents the successful TO delivery of \bar{m} and \bar{m} , respectively. This ensures that m has been processed in the same sequence by all relevant processes, maintaining consistency across the cluster.

Figure 3.18 P₀ and P₂ generate and send $\mu(\text{red})$ and $\mu(\text{green})$ Concurrently to CN₀ and CN₂

Then P₃ forwards $\mu(\text{green})$ to P₀ while P₁ forwards $\mu(\text{red})$ to P₂ concurrently as shown in Figure 3.19. When P₀ and P₂ receives the $\mu(\text{green})$ and $\mu(\text{red})$ messages both processes, P₀ and P₂, proceed to discard the $\mu(\text{green})$ and $\mu(\text{red})$ since the associated messages, m and m does not show a higher stabilisation at both P₀ and P₂ and had already been TO delivered (see Section 3.5.1).

Figure 3.19 P₁ and P₃ Forward $\mu(\text{red})$ and $\mu(\text{green})$ Concurrently to CN₁ and CN₃

Finally, P_2 forwards $\mu(\mathbf{m})$ to P_3 while P_0 forwards $\mu(\mathbf{m})$ to P_1 concurrently as shown in Figure 3.20. When P_3 and P_1 receives $\mu(\mathbf{m})$ and $\mu(\mathbf{m})$ both processes, P_3 and P_1 , proceed to discard $\mu(\mathbf{m})$ and $\mu(\mathbf{m})$ since the associated messages, \mathbf{m} and \mathbf{m} does not show a higher stabilisation at both P_3 and P_1 and had already been TO delivered by P_3 and P_1

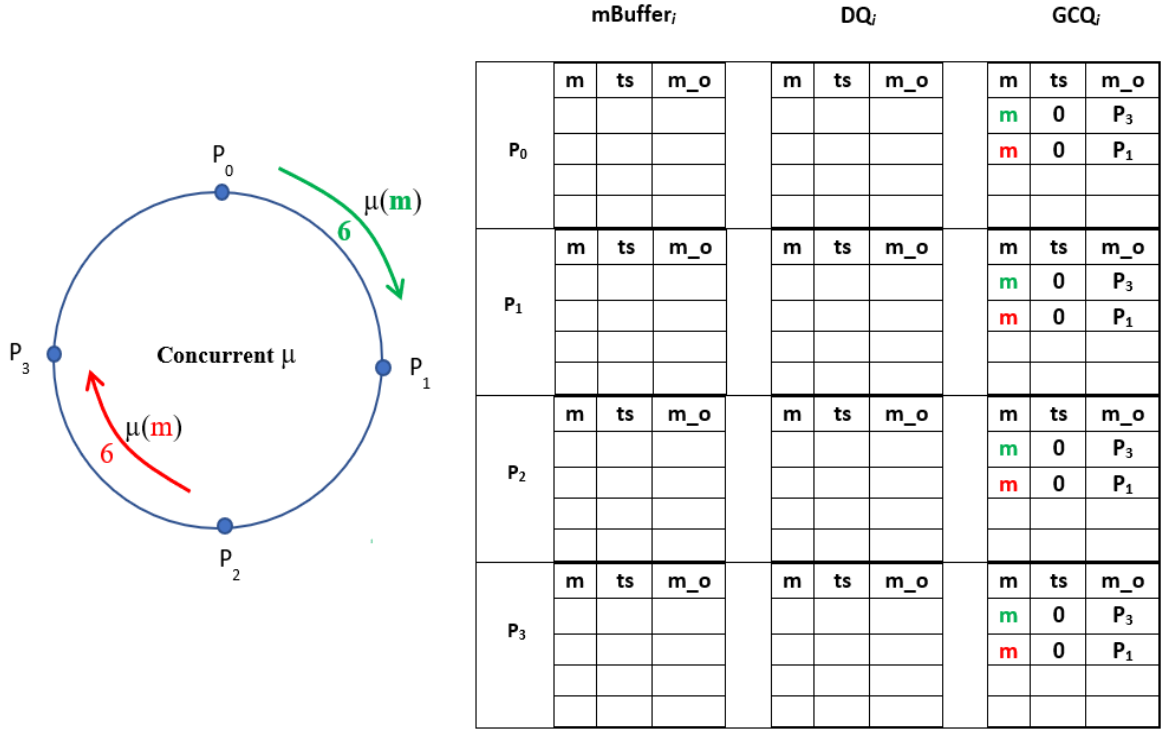


Figure 3.20 P_0 and P_2 Forward $\mu(\mathbf{m})$ and $\mu(\mathbf{m})$ Concurrently to CN_0 and CN_2

Therefore, with concurrency, P_3 and P_1 TO deliver messages, \mathbf{m} and \mathbf{m} , respectively after $[2(N - 1) - 2]$ hops, while P_2 and P_0 TO deliver messages, \mathbf{m} and \mathbf{m} , respectively, after $[2(N - 1) - 1 - 2]$ hops. We noted that if two concurrent sources are apart by k hops, latency is also reduced by k hops. However, this is a unique case observed with a single concurrent multicast, and cannot make a generalization, as we do not have control of the distribution when multiple messages are sent concurrently during simulations.

3.5.4 Garbage Collection

A process P_i can garbage collect a message m that it delivered, that is irreversibly discarded m , only when it knows that every process P_j has also delivered that m . Thus, at the heart of the garbage collection algorithm is the task of collecting this global knowledge on message delivery. To gather this knowledge, each process maintains some additional data structures and variables and also uses a special message denoted as Δ message, to share the local delivery-related information with other processes. We will explain these additional entities below:

Delivery Counter (DC)

Every process P_i maintains DC_i which indicates the timestamp upto which it has locally completed TO-delivery. More precisely, P_i has TO-delivered all m with $m_ts \leq DC_i$; i.e., it will never TO-deliver any m with $m_ts \leq DC_i$ and will TO-deliver in future only those m with $m_ts > DC_i$. Since messages are TO-delivered in the non-decreasing order of their timestamps, DC_i will only increase, and will never decrease, with time. DC_i will be updated during the TO-delivery() whose modified pseudocode is given below.

Part 3: TO – Delivery

```

do {
    m = head (DQi)
    while (m_flag == true) do
        {
            dequeue  $m$  from DQi
            deliver  $m$  to application process
            enter a copy of  $m$  in GCQi
            if (head (DQi) is empty or  $m\_ts < \text{head} (DQ_i)$ )
                then  $DC_i = m\_ts$ 
        } // end while
    }
forever

```

Note that when m_ts of a just TO-delivered m is less than the timestamp of the next message in a non-empty DQ_i or if TO-delivery of m leads DQ_i to be empty, then it indicates that all m_ts have been TO-delivered and DC_i can now be set to m_ts .

Garbage Collection Target Timestamp

In our garbage collection protocol, processes do not disseminate every increase in their DC_i but only when DC_i reaches a target timestamp value, called *Garbage Collection Target*, and is denoted as GCT. Every process P_i has the variable GCT_i which is initialised to zero at the start. As garbage collection of delivered messages progresses, the system-wide target GCT also needs to be increased. We assume that GCT is incremented by a fixed value, called the target increment, and denoted as TI, which is known to all processes.

We also assume that only one process will apply the increment TI on GCT on behalf of all other processes at any given time. Without loss of generality, we assign P_0 to be the process to apply the first increment, thus setting GCT₀ from 0 to $(0 + TI)$.

Delta Messages

They are used to exchange information about local TO-delivery among processes. It has two fields:

Δ .GCT: The current target value.

Δ .Setter: The identity of the process that sets the target Δ .GCT

Every process P_i observes the following three rules on handling a Δ -message- two rules before sending Δ and one after receiving it:

Rule 1:

P_i cannot send Δ message (to CN_i) unless it has earlier received Δ message (from ACN_i).

The only exception to this rule is when P_0 initiates the circulation of the Δ message with first non-zero Δ .GCT = TI.

Rule 2:

P_i cannot send Δ message (to CN_i) until Δ .GCT \leq DC_i .

Thus, a received Δ may have to wait for an arbitrary amount of time in P_i until DC_i equals or exceeds the target specified in Δ . As a result, whenever P_i sends Δ , it removes the old timestamp (in the received Δ) and gives the outgoing Δ a fresh timestamp. Note that this rule also allows an outgoing Δ to be piggybacked (without a fresh timestamp) onto any m or $\mu(m)$ that is being sent/forwarded at that time, thus saving message overhead.

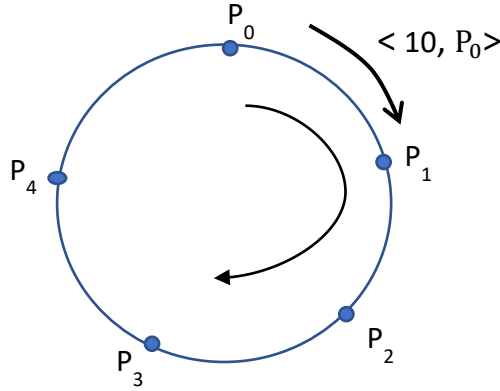
Rule 3:

When P_i receives Δ message, it garbage collects all m in GCQ_i with $m_{ts} \leq GCT_i$ and then sets $GCT_i = \Delta$.GCT.

As we will explain shortly, Δ .GCT within a received Δ will always be larger than GCT_i by TI; also, arrival of Δ indicates to P_i that all processes have delivered upto at least GCT_i .

Garbage Collection Sub-protocol:

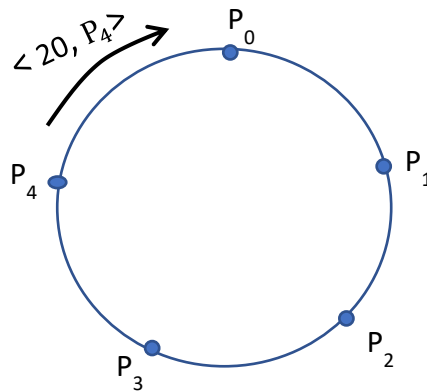
We will first informally explain the sub-protocol with an example and then provide the pseudocode for any process P_i . Assume, for illustration, $TI = 10$ and $N = 5$. Initially every P_i , $0 \leq i \leq 4$, has set $GCT_i = 0$ and knows $TI = 10$.

Figure 3.21 Initiating Circulation of Δ Message

Process P_0 initiates Δ circulation by forming $\Delta.GCT = GCT_0 + TI = 10$, and $\Delta.Setter = P_0$. Let us for simplicity represent this Δ message as $\langle 10, P_0 \rangle$ as depicted in Figure 3.21. $\langle 10, P_0 \rangle$ is now sent to $CN_0 = P_1$ after $DC_0 \geq \Delta.GCT = 10$ (see Rule 2).

When P_i , $0 \leq i \leq 3$, receives $\langle 10, P_0 \rangle$ from ACN_i , it garbage collects upto $GCT_i = 0$ (which is a null operation in the first circulation of Δ) as per Rule 3. P_i then sets $GCT_i = \Delta.GCT = 10$ and waits for $DC_i \geq GCT_i = 10$. Once the latter becomes true, it sends (or piggybacks) $\langle 10, P_0 \rangle$ to CN_i , except when $P_i = P_4$ because $CN_4 = \Delta.Setter = P_0$.

The process P_4 , which is the ACN of $\Delta.Setter$, now knows that all processes (including itself) has delivered upto $GCT_4 = 10$ and therefore garbage collects upto $GCT_4 = 10$. It then sets $GCT_4 = 10 + TI = 20$ and sends $\langle 20, P_4 \rangle$ to $CN_4 = P_0$ after its $DC_4 \geq 20$ becomes true (Rule 2 again). This initiates the second round of Δ circulation as depicted in Figure 3.22.

Figure 3.22 Second Round of Δ Circulation

Note that the value of $\Delta.Setter$ remains unchanged (as with $\Delta.GCT$) during circulation until the circulation ends (at P_4 for $\langle 10, P_0 \rangle$) so that the last process can initiate the next circulation.

When P_i , $0 \leq i \leq 3$, receives $\langle 20, P_4 \rangle$:

- It garbage collects upto $GCT_i = 10$ (Rule 3)
- It sets $GCT_i = \Delta.GCT = 20$
- It waits until $DC_i \geq GCT_i = 20$
- If $P_i \neq P_3$, P_i forwards $\langle 20, P_4 \rangle$ to CN_i
- If $P_i = P_3$ then P_3 garbage collects upto $GCT_3 = 20$, sets $GCT_3 = 20 + TI = 30$, forms Δ as $\Delta.GCT = GCT_3 = 30$, and $\Delta.Setter = P_3$ and waits until $DC_3 \geq GCT_3$ before initiating the third round of Δ circulation of $\langle 30, P_3 \rangle$. At that time all processes could have garbage collected at least upto 10.

Similarly, the third circulation (of $\langle 30, P_3 \rangle$) will terminate at P_2 which will initiate the fourth circulation (of $\langle 40, P_2 \rangle$) at which time all processes would have garbage collected upto at least 20. As this cycle of Δ circulation continues, processes progress in garbage collecting TO-delivered messages.

Pseudocode

The pseudocode assumes a thread called *Garbage-Collect(Δ)* that is invoked whenever a Δ message is received. Hence the pseudocode in part 2 of Uniform – Delivery_i is extended as follows:

```

do {
    receive M;
    if M = m then
        {
            as before
        }
    if M =  $\mu(m)$  then
        {
            as before
        }
    if M =  $\Delta$  then
        {
            invoke Garbage-collect( $\Delta$ )
        }
}
forever

```

The pseudocode for Garbage-collect(Δ) with input as the received Δ is given below for P_i

Garbage- Collect (Δ)

```

{
    discard all m in GCQi such that mts ≤ GCTi;
    GCTi =  $\Delta$ .GCT;
    GCT – Setteri =  $\Delta$ .Setter;
    discard ( $\Delta$ ) // copy contents of received  $\Delta$  and discard it
    wait until DCi ≥ GCTi;
    if CNi ≠ GCT – Setteri then
    {
        form  $\Delta$  such that
         $\Delta$ .GCT = GCTi
         $\Delta$ .Setter = GCT – Setteri
         $\Delta$ .destn = CNi
        Send ( $\Delta$ );
    }
else
{
    discard all m in GCQi such that mts ≤ GCTi;
    GCTi = GCTi + TI; // set new GCT;
                        // can be optimised
    form  $\Delta$  such that
     $\Delta$ .GCT = GCTi ;
     $\Delta$ .Setter = Pi;
     $\Delta$ .destn = CNi;
    wait until DCi ≥ GCTi;
    Send ( $\Delta$ ); // new circulation round initiated
}
}

```

Note that the else part (CN_i = Δ .Setter) ensures that new circulation round is initiated after DC_i ≥ GCT_i

Optimisations:

As stated earlier, Δ can be piggybacked along any outgoing m or $\mu(m)$ that is being currently set. We assumed that TI is a fixed integer known to all processes and Δ .GCT is always incremented by TI by Δ .Setter. In fact, Δ .Setter can choose any value that suits the rate of TO-delivery which in turn depends on the rate of messages being sent for TO-delivery by the application processes. So, we can fix an adaptive setting of new Δ .GCT by changing

$$\begin{aligned}
 & \text{GCT}_i = \text{GCT}_i + \text{TI} \text{ into} \\
 & \text{GCT}_i = \max\{\text{GCT}_i + \text{TI}, \text{DC}_i\}.
 \end{aligned}$$

If the rate of TO-delivery is large then DC_i will be chosen as the larger of the two and more TO-delivered messages will be garbage collected in each round.

3.5.5 DCTOP Membership Changes

Membership of the group of processes executing DCTOP can change due to

- (i) A crashed member being removed from the ring and/or
- (ii) A former member recovering and being included in the ring.

Let G represent the group of DCTOP processes executing the protocol at any given time. G is initially $\Pi = \{P_0, P_1, P_2, \dots, P_{N-1}\}$ and $G \subseteq \Pi$ is true at all times.

Figure 3.23 depicts an example of DCTOP membership changes by assuming $N = 5$

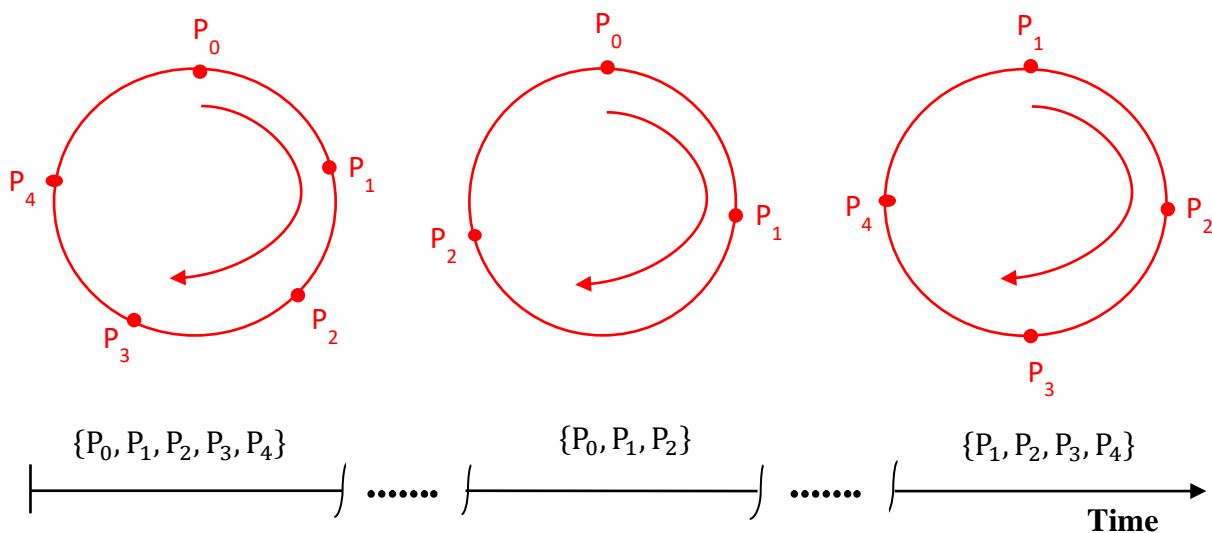


Figure 3.23 DCTOP Membership Changes Overtime - An Example

Referring to Figure 3.23, G starts with being $G = \Pi = \{P_0, P_1, P_2, P_3, P_4\}$ where at most two processes can crash. It is assumed that P_3 and P_4 both crash at the same time. G now becomes $\{P_0, P_1, P_2\}$ in which at most one process can crash. Figure 3.23 assumes that P_0 crashes at a later time while both P_3 and P_4 have recovered. The new G now becomes $\{P_1, P_2, P_3, P_4\}$ in which at most one can crash.

Recall that our DCTOP design assumes the use of a membership service, similar to that of RAFT [3] or Viewstamped replication [130]. More precisely, the local membership is assumed to send an interrupt to the local DCTOP process, say, P_i when a membership change is imminent.

On receiving the interrupt P_i completes processing of any message it has already started processing and then suspends all DCTOP activities and waits for new G to be formed: sending

of m or $\mu(m)$ (by enqueueing into SendingQueue_i), receiving of m or $\mu(m)$ (from IncomingQueue_i) and delivering of m (from DQ_i) are all suspended.

In this sub-section, we will address how P_i resumes DCTOP operations in the new G after the latter is constituted by the membership service. It should be emphasised that between the moment DCTOP activities are suspended in old G and the moment they are resumed in new G , processes communicate directly with each other, i.e., the unidirectional ring communication structure is also suspended. First, we define some notations.

Definitions of Notations

The old group version that *immediately precedes* G is denoted as G_{prev} . Referring to Figure 3.23, G_{prev} is empty (i.e., non-existent) when $G = \{P_0, P_1, P_2, P_3, P_4\}$; $G_{\text{prev}} = \Pi$ when $G = \{P_0, P_1, P_2\}$ and $G_{\text{prev}} = \{P_0, P_1, P_2\}$ when $G = \{P_1, P_2, P_3, P_4\}$.

$$G_{\text{prev}} = \text{immediate predecessor of } G$$

Survivors

We define $\text{Survivors}(G)$ as the set of processes of G_{prev} that remain as members of G as well.

$$\text{Survivors}(G) = G_{\text{prev}} \cap G$$

Joiners

$\text{Joiners}(G)$ are defined as incoming members of G that were not in G_{prev} but ‘joined’ G after completing recovery from an earlier crash.

$$\text{Joiners}(G) = G - \text{Survivors}(G)$$

Referring to Figure 3.23, when $G = \{P_1, P_2, P_3, P_4\}$, $\text{Joiners}(G) = \{P_3, P_4\}$, and $\text{Survivors}(G) = \{P_1, P_2\}$. Note that at the time of system initialisation, i.e., at time = 0,

- $G = \Pi = \{P_0, P_1, P_2, P_3, P_4\}$
- $\text{Survivors}(G) = G \cap \{\} = \{\}$
- $\text{Joiners}(G) = G$

That is every process is a joiner and no process is a survivor.

At any time $t > 0$,

$$|\text{Survivors}(G)| \geq \left\lceil \frac{|G_{\text{prev}}|+1}{2} \right\rceil$$

$\text{Survivors}(G)$ is a majority subset of G_{prev} (see Assumption AS1). That is, if P_k in G_{prev} TO-delivers m and does not survive into G_k and if P_i in G does not have m , then at least one process

in Survivors(G), say P_S , must have that m in $mBuffer_S$, DQ_S or GCQ_S by the requirement of crashproof for TO-delivery.

Last TO-Delivered Message

Even though DCTOP processes of G are TO-delivering messages in identical order, they may not be TO-delivering the same message at the same time. When they receive an interrupt from a local membership service, they may have TO-delivered different sequences of messages to their respective local application process.

For any $P_i \in G$ that receives an interrupt, let $Last_i$ denote the last m that P_i TO-delivered just before it suspends the DCTOP activities. Let $P_k \in G$ be another process that also received the membership interrupt, $k \neq i$. Since $k \neq i$, $Last_i$ and $Last_k$ are related in one of three ways:

- Both are the same: $Last_i \equiv Last_k$;
- $Last_i$ is totally ordered before $Last_k$ denoted as $Last_i \ll Last_k$: $Last_{i_ts} < Last_{k_ts}$
OR $(Last_{i_origin} > Last_{k_origin} \text{ AND } Last_{i_ts} = Last_{k_ts})$.
- $Last_k \ll Last_i$

Earliest and Latest Last TO-Delivered Messages

For Survivors(G), we define $Last_E$ and $Last_L$ as the earliest and the latest among the last messages delivered by processes of Survivors(G) when the latter are arranged in total order.

More precisely, for all $P_S \in Survivors(G)$:

- $Last_S \equiv Last_E$ or $Last_E \ll Last_S$
- $Last_S = Last_L$ or $Last_S \ll Last_L$

In other words, every surviving process $P_S \in Survivors(G)$ has TO-delivered all m , $m \ll Last_E$ and no P_S has TO-delivered any m , $m \gg Last_L$.

Finally, the surviving process P_S whose $Last_S \equiv Last_L$ is called the “Lead” because it has delivered the latest m compared to other processes in Survivors(G). This also means that the application process to which Lead-Survivor TO-delivers, is most advanced in its processing and has the latest application service state.

STEPS FOR RESUMING DCTOP

These steps serve two objectives:

Objective 1:

Survivors(G), the processes that were in G_{prev} and migrated to G , must TO-deliver all messages sent in G_{prev} .

Accomplishing this objective must address all cases that a crash may give rise to. Assume that $G_{prev} = \{P_0, P_1, P_2\}$ as in Figure 3.25. Assume also that P_2 sends m to P_0 just before membership change and P_0 crashes before forwarding m to P_1 . After $G = \{P_1, P_2, P_3, P_4\}$ is constituted, P_2 must transit m to P_1 so both can TO-deliver m before resuming DCTOP in G .

Consider another example to illustrate a different crash-related complication. Suppose that P_0 in G_{prev} forms and sends m to P_1 and then sends $\mu(m')$ with $m'_{ts} \geq m_{ts}$ to P_1 , before crashing. P_1 TO-delivers m and receives membership interrupt; it suspends all activities before m can leave the $OutgoingQueue_1$ for transmission. So, P_2 now needs to receive m which is only with P_1 (in its GCQ_1). So, before resuming DCTOP in G , P_1 must retrieve a copy of m from GCQ_1 and send it to P_2 so that the latter can also deliver m sent in G_{prev} .

Objective 2:

Any process P_j in Joiners(G) will have its application process state updated to the state instance where all Survivors(G) will reach when they TO-deliver all m sent in G_{prev} .

It is important to note that the joiner P_j , when it crashed earlier, will be out of synchrony with Survivors(G) as it was out of action for a while. So, its state must be updated. To accomplish this objective, we make use of Lead-Survivor which has the latest application state.

The Lead-Survivor in Survivors(G) sends to P_j :

- (i) Its current checkpoint C
- (ii) Its TO_Queue , denoted as TO_Queue_{LS} , which it will construct in accomplishing objective 1.

P_j will update its state using C and TO-deliver every m in TO_Queue_{LS} . Once TO-delivered messages are processed the stable P_j will be in synchrony with Survivors of G . Now, P_j is ready to resume DCTOP in G .

Steps executed by every P_i in Survivors(G):

The following six steps are executed by every $P_i \in \text{Survivors}(G)$. For convenience, another Survivor process (that is not P_i) is denoted as P_s .

Step 1

```

{
  multicast( $Last_{i\_ts}$ ,  $Last_{i\_origin}$ ) to every other Survivor  $P_s$ 
  compute  $Last_E$ ,  $Last_L$ , LEAD-Survivor after receiving ( $Last_{s\_ts}$ ,  $Last_{s\_origin}$ ) from
  every other  $P_s$ 
}

```

In Step 1, Survivors of G exchange information about the last message they TO-delivered and other useful information is derived once the message exchange is complete among all Survivors of G .

Step 2

// Send all messages that need to be sent

```

{
  (a) every  $m$  in  $SendingQueue_i$  is timestamped and sent to every other  $P_s$ ;
      // Sending Queue cleared
  (b) for every other  $P_s \in Survivors(G)$ 
      do
      {
        Send to  $P_s$  every  $m$  in  $mBuffer_i$ ;
        TO_Queue $_i$  or GCQ $_i$  such that  $m \gg Last_s$ ;
        // any missing message of  $P_i$  is now sent to  $P_s$ 
      (c) Send “Finished $_i$ ” to all  $P_s$  in Survivor( $G$ )
      }
}

```

In Step 2(c), P_i by sending “Finished $_i$ ” informs other P_s that it has completed the sending step.

Step 3

// Receive all messages that need to be received

```

{
  (a) Repeat
      {
        receive( $m$ )
        If ( $m$  is in  $mBuffer_i$  or TO_Queue $_i$  or GCQ $_i$ ) then
          discard( $m$ ) // duplicate
        else { Store  $m$  in  $mBuffer_i$  }
      }
      Until Finished $_s$  is received from every other  $P_s$ ;
  (b) Send “Ready $_i$ ” to every other  $P_s \in Survivor(G)$ ;
}

```

In Step 3(a), P_i stores all non-duplicate messages in $mBuffer_i$; receiving Finished $_s$ from all other P_s indicates that all messages to be received have been received by P_i .

Step 4

```
// Stabilise mBufferi and build TO_Queuei
{
  (a) Wait until Readys received from every other Ps;
  (b) Repeat
    {
      remove m in Total order from mBufferi;
      enqueue m in TO_Queuei
    }
  Until mBufferi has no m
}
```

The condition in Step 4(a) ensures that P_i knows that every Survivor P_s has received what it sent to P_s. That is, all *m* in mBuffer_i are stable and can be totally ordered as shown in Step 4(b).

Step 5

```
// executed by Lead-Survivor if there are Joiners
{
  If ( Pi = Lead-Survivor AND Joiners(G) ≠ { } ) then
    {
      Compute checkpoint C;
      Send Invite (C, TO_Queuei) to every Pj in NewComer(G);
    }
}
```

State Checkpoint and TO_Queue of the lead-Survivor is sent to all P_j ∈ NewComer(G) so that P_j can “catch-up” with Survivors(G).

Step 6

```
// Resume TO-Delivery in Gprev
{
  (a) Repeat
    {
      dequeue(m);
      deliver(m) to application process
      enqueue(m) in GCQi
    }
  Until TO – Queuei is empty;
  (b) Send “Completedi” to every other Pk in G;
}
```

Note that in Step 6(b) a completed_i message is sent to every process in G to indicate that P_i has completed TO-delivering in Gprev.

Step 7*// initialise in (new) G*

```

{
  wait until completedk received from every other  $P_k \in G$ ;
  initialise DCTOP variables
    //  $LC_i = 0, SC_i = 0$ 
  empty buffer and Queues
    //  $mBuffer_i = \text{empty buffer}$ 
    //  $GCQ_i = \text{empty queue}$ 
  resume DCTOP in G
}

```

The following three steps are executed by every $P_j \in \text{Joiners}(G)$ which will meet objective 2.

Step 1J*// receive checkpoint for state update*

```

{
  wait until Invite(C, Q) received from some  $P_i \in G$ ;
  implement checkpoint C;
  TO – Queuej = Q;
}

```

Step 2J and Step 3J are exactly the same as steps 6 and 7 for a Survivor, with P_i /subscript i replaced by P_j /subscript j.

Moreover, the explicit differences between DCTOP and LCR, including the different assumptions regarding failures are stated as follows: (i) DCTOP utilizes a Lamport logical clock for message timestamping. The Lamport clock is simpler and requires only a single integer, making it more lightweight while LCR uses a vector clock for message vector timestamping. The vector timestamp involves multiple integers (one for each process), which increases information overhead. (ii) DCTOP uses a dynamically determine last process for ordering concurrent messages while LCR relies on a globally fixed last process for ordering concurrent messages. (iii) DCTOP assumes a failure threshold of $f = (N-1)/2$, which implies $N=2f+1$. This assumption enables DCTOP to efficiently determine the crashproofness of a message early, thereby reducing the total order (TO) message delivery latency. By relaxing the constraints on fault tolerance, the protocol ensures faster decision-making while still maintaining reliability and consistency in distributed systems. In contrast, LCR operates under a stricter failure threshold of $f=N-1$, requiring all processes to receive a message before it can be delivered. While this assumption enhances fault tolerance, it comes at the cost of increased latency. Finally, the pseudocodes of the DCTOP protocol, along with the procedures for handling membership changes, are summarised in Figure 3.24, Figure 3.25, and Figure 3.26.

Pseudocode 1a: Message multicast and the approaches executed by any process P_i

```

1. Procedure initialization (initial_view for each  $P_i$ )
2.    $mBuffer_i \leftarrow \emptyset$  {holds incoming messages}
3.    $DQ_i \leftarrow \emptyset$  {stores totally ordered messages}
4.    $GCQ_i \leftarrow \emptyset$  {stores garbage-collected messages}
5.    $LC_i \leftarrow \{0, \dots, 0\}$  {local logical clock}
6.    $SC_i \leftarrow \{0, \dots, 0\}$  {stability clock}
7.    $SendingQueue_i \leftarrow \emptyset$  {outgoing message queue}
8.    $Group\ G \leftarrow initial\_G$  {set of initial group members}

9. Procedure utoMulticast (M) at  $P_i$ 
10.  a. Initialize (M):
11.     $M\_flag = false$ 
12.    Assign timestamp  $M\_ts = LC_i$ .
13.    Enqueue M in  $SendingQueue_i$ 
14.  b. Multicast M reliably to all  $P_j \in G$  {multicast a message}
15.    Store a copy of m in  $mBuffer_i$ 
16.    Increment  $LC_i$  after sending m

17. Upon Receive (M) do
18.   If (M = m) then
19.     Update  $LC_i = \max(LC_i, m\_ts + 1)$  {update local logical clock}
20.     If  $Hops_{j,i} \geq f$ , mark  $m\_flag = true$  {message is crash-proof}
21.     Store m in  $mBuffer_i$ .
22.     Forwarding Decision:
23.     If  $P_j \neq CN_i$ :
24.       Set  $m\_destn = CN_i$ .
25.       Enqueue m in  $OutgoingQueue_i$  {forward the message}
26.     Else:
27.       Update  $SC = \max(SC_i, m\_ts)$ .
28.       Mark all m,  $m\_ts \leq SC_i$  as stable in  $mBuffer_i$  {m is stable}
29.       Move these messages to  $DQ_i$  in total order
30.       Form  $\mu(m)$  with  $\mu(m)\_destn = CN_i$ 
31.       Enqueue  $\mu(m)$  in  $OutgoingQueue_i$  {forward the  $\mu(m)$ }
32.   If (M =  $\mu(m)$ ) then
33.     Check if  $\mu(m)$  is meaningful or not:
34.     If  $\mu(m)$  contains m where  $m\_ts \leq SC_i$  and  $Hops_{i,j} \geq f$ , discard  $\mu(m)$ .
35.     Otherwise:
36.       If  $Hops_{i,j} < f$ 
37.         Search for m in  $mBuffer_i$  or  $DQueue_i$ 
38.         Mark  $m\_flag = true$  {message is crashproof}
39.          $\mu(m)\_destn = CN_i$ 
40.         Enqueue  $\mu(m)$  in  $OutgoingQueue_i$  {forward the  $\mu(m)$ }
41.       If  $m\_ts > SC_i$ 
42.         Update  $SC_i = m\_ts$ 
43.         Stabilize all m,  $m\_ts \leq SC_i$  in  $mBuffer_i$  {m is stable}
44.         Move these messages to  $DQ_i$  in total order
45.         If  $P_j = CN_i$ , discard  $\mu(m)$ .
46.         Otherwise, forward  $\mu(m)$  to  $CN_i$ 

```

Figure 3.24 Pseudocode of the DCTOP Protocol

Pseudocode 1b: Uniform Total Order Delivery and Garbage Collection at P_i

```

47. Procedure utoDeliver(m)
48.   Repeat until  $DQ_i$  is empty
49.     Dequeue  $m = \text{head}(DQ_i)$ 
50.     If  $m\_flag = \text{true}$ :
51.       Deliver  $m$  to the application process.           {deliver a message}
52.       Add  $m$  to  $GCQ_i$ .
53.       If  $DQ_i$  is empty or  $(m\_ts < \text{head}(DQ_i))$ :
54.         Update  $DC_i = m\_ts$ .

55. Procedure Garbage Collect ( $\Delta$ )
56.   If Message  $\Delta$  is Received then
57.     Invoke Garbage_Collect( $\Delta$ ):
58.       Discard all  $m \in GCQ_i$  where  $m\_ts \leq GCT_i$            {garbage collect a message}
59.       Update  $GCT_i = \Delta.GCT$ .
60.       Set  $GCT\_Setter_i = \Delta.Setter$ 
61.       discard ( $\Delta$ ) // copy contents of received  $\Delta$  and discard it
62.       Wait until  $DC_i \geq GCT_i$ .
63.   If  $CN_i \neq GCT\_Setter_i$  then
64.     Form  $\Delta$  with
65.        $\Delta.GCT = GCT_i$ 
66.        $\Delta.Setter = GCT\_Setter_i$ 
67.        $\Delta.destn = CN_i$ 
68.     Send ( $\Delta$ )
69.   Else
70.     Discard all  $m \in GCQ_i$  with  $m\_ts \leq GCT_i$            {garbage collect a message}
71.     Increment  $GCT_i = GCT_i + TI$                            {set new GCT}
72.     form  $\Delta$  with
73.        $\Delta.GCT = GCT_i$ 
74.        $\Delta.Setter = P_i$ 
75.        $\Delta.destn = CN_i$ 
76.       wait until  $DC_i \geq GCT_i$ 
77.     Send ( $\Delta$ )                                           {new circulation round initiated}

```

Figure 3.25 Continuation of the Pseudocode of the DCTOP Protocol.

Pseudocode 2: Membership Change Steps Executed by any P_i in Survivor(G)

Upon Membership Change ($G' \leftarrow \text{new group}$)

1. **Determine Lead Survivor**
 - P_i multicasts ($\text{Last}_i\text{-ts}$, $\text{Last}_i\text{-origin}$) to all other Survivors $P_s \in G$.
 - After receiving ($\text{Last}_s\text{-ts}$, $\text{Last}_s\text{-origin}$) from all P_s , compute:
 - i. Last_E , Last_L , and LEAD-Survivor.
2. **Send Pending Messages**
 - (a) For each message $m \in \text{SendingQueue}_i$, timestamp and send m to all $P_s \in G$.
 - Clear SendingQueue_i .
 - (b) For each Survivor P_s , send any missing messages from $m\text{Buffer}_i$, TO_Queue_i , or GCQ_i such that $m \gg \text{Last}_s$.
 - (c) Send Finished_i to all $P_s \in G$.
3. **Receive All Messages**
 - (a) Repeat the following until Finished_s is received from every $P_s \in G$:
 - Receive m .
 - If $m \in \{m\text{Buffer}_i, \text{TO_Queue}_i, \text{GCQ}_i\}$: discard m (duplicate).
 - Otherwise, store m in $m\text{Buffer}_i$.
 - (b) Send Ready_i to all $P_s \in G$.
4. **Stabilize Buffers and Build Total Order Queue**
 - (a) Wait until Ready_s is received from every $P_s \in G$.
 - (b) Repeat the following until $m\text{Buffer}_i$ is empty:
 - Remove m from $m\text{Buffer}_i$ in total order.
 - Enqueue m into TO_Queue_i .
5. **Handle Joiners (Executed by Lead Survivor)**
 If $P_i = \text{LEAD-Survivor}$ and $\text{Joiners}(G') \neq \emptyset$:
 - (a) Compute checkpoint C .
 - (b) Send Invite (C , TO_Queue_i) to each $P_j \in \text{NewComer}(G')$.
6. **Resume Delivery in Previous Group G_{prev}**
 - (a) Repeat until TO_Queue_i is empty:
 - Dequeue m .
 - Deliver m to the application process.
 - Enqueue m into GCQ_i .
 - (b) Send Completed_i to all $P_k \in G$.
7. **Initialize for New Group G'**
 - Wait until Completed_k is received from all $P_k \in G'$.
 - Initialize DCTOP variables.
 - Clear buffers and queues.
 - Resume DCTOP in G' .

For Joiner Process P_j :

8. **Receive Checkpoint for Update**
 - Wait until Invite (C , Q) is received from a $P_i \in G$.
 - Apply checkpoint C .
 - Set $\text{TO_Queue}_j \leftarrow Q$.
 9. **Follow Steps 6 and 7 for Survivors**
 - Replace P_i references with P_j .
-

Figure 3.26 Pseudocode of the DCTOP Membership Changes

3.5.6 DCTOP Proof of Correctness

Lemma 1 (VALIDITY). *If any correct process P_i utoMulticasts a message m , then it eventually utoDelivers m .*

PROOF: Let P_i be a correct process and let m_i be a message sent by P_i . This message is added to $mBuffer_i$ (Line 15 of Figure 3.24). There are two cases to consider:

Case 1: Presence of membership change

If there is a membership change, P_i will be in $Survivor(G)$ since P_i is a correct process. Consequently, the membership changes steps ensure that P_i will deliver all messages stored in its $mBuffer_i$, TO_Queue_i or GCQ_i including m_i (Lines 6 (a, and b) of Figure 3.26). Thus, P_i utoDelivers message m_i that it sent.

Case 2: No membership changes.

When there is no membership change, all the processes within the DCTOP system including the m_i_origin will eventually deliver m_i after setting m_i stable (Line 28 of Figure 3.24). This happens because when P_i timestamps, sets $m_i_flag=false$ and sends m_i to its CN_i , it deposits a copy of m_i to its $mBuffer_i$ and sets $LC_i > m_i_ts$ afterwards. The message is forwarded along the ring network until the ACN_i receives m_i . Any process that receives m_i deposits a copy of it into its $mBuffer$ and sets $LC > m_i_ts$. It also checks if $Hops_{ij} \geq f$, then m_i is crashproof and it sets $m_i_flag=true$. The ACN_i sets m_i stable (Lines 28 of Figure 3.24) and crashproof (Line 20 of Figure 3.24) at ACN_i , transfers m_i to DQ and then it attempts utoDeliver m_i (Lines 47 to 54 of Figure 3.25) if m_i is at the head of DQ. ACN_i generates, timestamp $\mu(m_i)$ using its LC and then sends it to its own CN. Similarly, $\mu(m_i)$ is forwarded along the ring (Lines 31 and 40 of Figure 3.24) until the ACN of $\mu(m_i)_origin$ receives $\mu(m_i)$. When any process receives $\mu(m_i)$ and $Hops_{ij} < f$, it knows that m_i is stable and crashproof but if $Hops_{ij} \geq f$, then m_i is only stable because m_i is already known to be crashproof since at least $f+1$ processes had already received m_i . Any process that receives $\mu(m_i)$ transfers m_i from $mBuffer$ to DQ and then attempts to utoDeliver m_i if m_i is at the head of DQ. Suppose P_k sends m_k before receiving m_i , $i < k$. Consequently, ACN_i will receive m_k before it receives m_i and thus before sending $\mu(m_i)$ for m_i . As each process forwards messages in the order in which it receives them, we know that P_i will necessarily receive m_k before receiving $\mu(m_i)$ for message m_i .

- (a) If $m_i_ts = m_k_ts$, then P_i orders m_k before m_i in $mBuffer_i$ since $i < k$ (we assumed that when messages have equal timestamps, message from a higher origin is ordered before message from a lower origin.). When P_i receives $\mu(m_i)$ for message m_i it transfers both

messages to DQ and can *utoDeliver* both messages, m_k before m_i , because TS is already known to be stable because of TS equality.

- (b) If $m_i_ts < m_k_ts$ then P_i orders m_i before m_k in $mBuffer_i$. When P_i receives $\mu(m_i)$ for message m_i it transfers both messages to DQ and can *utoDeliver* m_i only since it is stable and at the head of DQ. P_i will eventually *utoDeliver* m_k when it receives $\mu(m_k)$ for m_k since it is now at the head of DQ after m_i delivery.
- (c) Option (a) or (b) is applicable in any other processes within the DCTOP system since there are no membership changes. Thus, if any correct process P_i sends a message m_i , then it eventually delivers m_i .

Note that if $f+1$ processes receive a message m , then m is crashproof. During concurrent multicasts, the timestamp (TS) can stabilize quickly, allowing m to be delivered even before the ACN of m_origin receives it. This outcome is due to the relaxation in our crash failure assumption.

Lemma 2 (INTEGRITY). *For any message m , any process P_k *utoDelivers* m at most once, and only if m was previously *utoMulticast* by some process P_i .*

PROOF. Our crash failure assumption ensures that no false message is ever *utoDelivered* by a process. Thus, only messages that have been *utoMulticast* are *utoDelivered*. Moreover, each process maintains a logical clock LC, which is updated to ensure that every message is delivered only once. The sending rule ensures that messages are sent with an increasing timestamp by any process P_i , and the receive rule ensures that the LC of the receiving process is updated after receiving a message. This means that no process can send any two messages with equal timestamps. Hence, if there is no membership change, Lines 16 and 19 of Figure 3.24 guarantee that no message is processed twice by process P_k . In the case of a membership change, Line 3(a) of Figure 3.26 ensures that process P_k does not deliver messages twice. Additionally, Lines 7 of Figure 3.26 ensure that P_k 's variables such as logical and stability clock are set to zero, and the buffer and queues are emptied after a membership change. This is done because processes had already delivered all the messages of the old group discarding message duplicates (Line 3(a) of Figure 3.26) to the application process and no messages in the old group will be delivered in the new group. Thus, after a membership change, the new group is started as if it were a new DCTOP operation. The new group might contain messages with the same timestamp as those in the old group, but these messages are distinct from those in the old group. Since timestamps are primarily used to maintain message order and delivery,

they do not hold significant meaning for the application process itself. This strict condition is to ensure that messages already delivered during the membership change procedure are not delivered again in the future.

Lemma 3 (UNIFORM AGREEMENT). *If any process P_j utoDelivers any message m in the current G , then every correct process P_k in the current G eventually utoDelivers m .*

PROOF. Let m_i be a message sent by process P_i and let P_j be a process that delivered m_i in the current G .

Case 1: P_j delivered m_i during a membership change.

This means that P_j had m_i in its $mBuffer_i$, TO_Queue_i or GCQ_i before executing line 6(a and b) of Figure 3.26. Since all correct processes exchange their $mBuffer_i$, TO_Queue_i and GCQ_i during the membership change procedure, we are sure that all correct processes that did not deliver m_i before the membership change will have it in their $mBuffer_i$, TO_Queue_i or GCQ_i before executing procedures 1 to 7 of Figure 3.26. Consequently, all correct processes in the new G will deliver m_i .

Case 2: P_j delivered m_i in the absence of a membership change.

The protocol ensures that m_i does a complete cycle around the ring before being delivered by P_j : indeed, P_j can only deliver m_i after it knows that m_i is crashproof and stable, which either happens when it is the ACN_i in the ring or when it receives $\mu(m_i)$ for message m_i . Remember that processes transfer messages from their $mBuffer$ to DQ when the messages become stable. Consequently, all processes stored m_i in their DQ before P_j delivered it. If a membership change occurs after P_j delivered m_i and before all other correct processes delivered it, the protocol ensures that all $Survivor(G)$ that did not yet deliver m_i will do it (Lines 6 (a, and b) of Figure 3.26). If there is no membership change after P_j delivered m_i and before all other processes delivered it, the protocol ensures that $\mu(m_i)$ for m_i will be forwarded around the ring, which will cause all processes to set m_i to stable and crashproof. Remember, when any process receives $\mu(m_i)$ and $Hops_{ij} < f$, it knows that m_i is stable and crashproof but if $Hops_{ij} \geq f$, then m_i is only stable because m_i is already known to be crashproof since at least $f+1$ processes had already received m_i . Each correct process will thus be able to deliver m_i as soon as m_i is at the head of DQ (Line 51 of Figure 3.25). The protocol ensures that m_i will become first eventually.

The reasons are the following: (1) the number of messages that are before m_i in DQ of every process P_k is strictly decreasing, and (2) all messages that are before m_i in DQ of a correct process P_k will become crashproof and stable eventually. The first reason is a consequence of the fact that once a process P_k sets message m_i to crashproof and stable, it can no longer receive any message m such that $m < m_i$. Indeed, a process P_l can only produce a message $m_l < m_i$ before receiving m_i . As each process forwards messages in the order in which it received them, we are sure that the process that will produce $\mu(m_i)$ for m_i will have first received m_l . Consequently, every process setting m_i to crashproof and stable will have first received m_l . The second reason is a consequence of the fact that for every message m that is utoMulticast in the system, the protocol ensures that $\mu(m)$ will be forwarded around the ring (Lines 31 and 40 of Figure 3.24), implying that all correct processes will mark the message as crashproof and stable. Consequently, all correct processes will eventually deliver m_i .

Lemma 4 (TOTAL ORDER). For any two messages m and m' if any process P_i utoDelivers m without having delivered m' , then no process P_j utoDelivers m' before m .

Suppose that P_i deduces stability of TS, $TS \geq 0$, for the first time by (i) above at, say, time t , that is, by receiving m , $m_ts = TS$ and $m_origin = CN_i$, at time t . P_i cannot have any m' , $m'_ts \leq TS$ in its IncomingQueue _{i} at time t nor will ever have m' at any time after t .

Proof (By Contradiction)

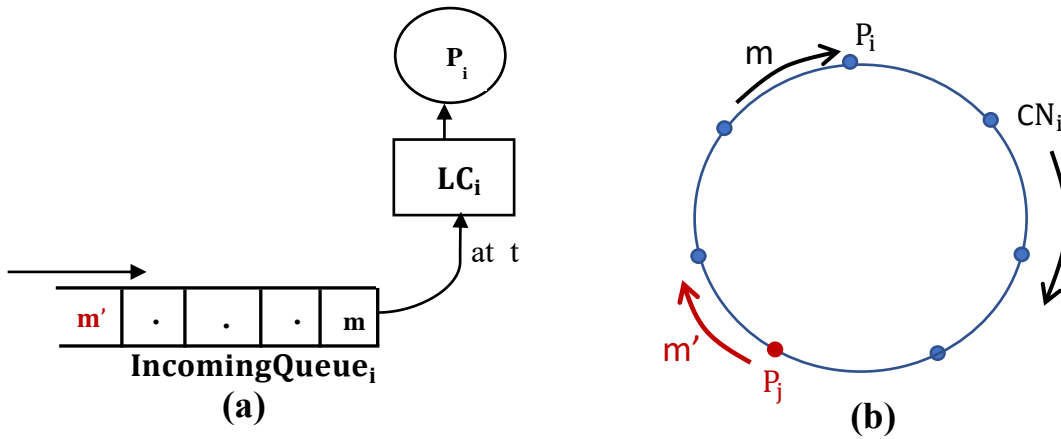


Figure 3.27 Example Contradicting Lemma 1

Assume, contrary to Lemma, that P_i is to receive m' , $m'_ts \leq TS$, after t as shown in Figure 3.27a.

Case 1:

Let $m_origin = m'_origin = P_j$. So, imagine that P_j is the same as CN_i , $P_j \equiv CN_i$, as shown in Figure 3.27b. Given that $m'_ts \leq TS = m_ts$, $m'_ts < m_ts$ must be true when $m_origin = m'_origin$. So, P_j must have sent m' first and then m .

Note that

- (a) The link between any pair of consecutive processes in the ring maintains FIFO, and
- (b) Processes $P_{j+1}, P_{j+2}, \dots, P_{i-1}$ forward messages in the order they received those messages.

Therefore, it is not possible for P_i to receive m' after it received m , that is, after t . So, case 1 cannot exist.

Case 2:

Imagine that m_origin is from CN_i and m'_origin is from P_j , $m_origin = CN_i \neq m'_origin = P_j$, as shown in Figure 3.27b. Since P_i is the last process to receive m in the system, P_j must have received m before t ; since $m'_ts \leq m_ts$, P_j could not have sent m' after receiving m . So, the only possibility for $m'_ts \leq m_ts$ to hold is: P_j must form and send m' before it is received and forwarded m .

For the cases of (a) and (b) in case 1, P_i must receive m' before m . Therefore, the assumption made contrary to Lemma 1 cannot be true. Thus, Lemma 1 is proved.

Theorem 1

If a process first deduces that TS is stable, say at time t , then it will never receive any m' , $m'_ts \leq TS$, after t .

Proof

Suppose that a process, say, P_i first deduces the stability of TS by receiving m , $m_ts = TS$ and $m_origin = CN_i$. By Lemma 1, the theorem is true for P_i .

Note that since P_i is the ACN of m_origin , all processes in the ring must have received m . Therefore, LCs of all processes (including P_i) must read a value larger than TS. That is, no process will ever generate a m' , $m'_ts \leq TS$, after P_i received m .

Suppose that P_i does not forward m that it received with $m_ts = TS$ and $m_origin = CN_i$; instead, P_i informs its clockwise neighbours of the stability of TS by generating and sending a special message denoted as $\mu(m)$, where m in $\mu(m)$ denotes the message whose reception led P_i to construct $\mu(m)$ to enable the deduction of TS stability.

Let a process, say, P_j , $j \neq i$, receive this $\mu(m)$ at time t and thereby learns for the first time that $TS = m_ts$ had become stable using the second means of timestamp stability deduction described earlier. We will now prove the theorem for P_j by showing that P_j will not receive any m' , $m'_ts \leq TS$ after it received $\mu(m)$ at time t .

As shown in Figure 3.28, let P_j be the h^{th} clockwise neighbour of P_i , $1 \leq h \leq N - 1$. We will do the proof by induction on h .

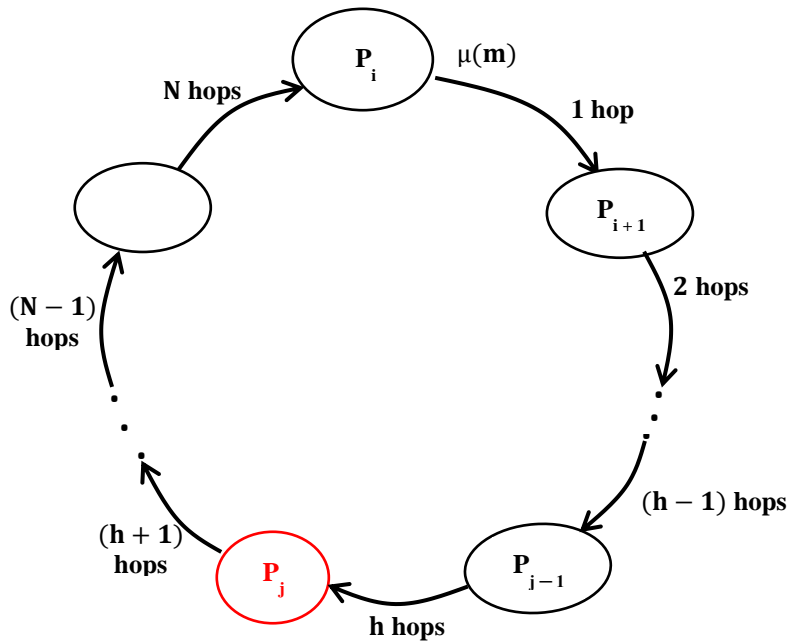


Figure 3.28 P_j deduces stability of m_ts by receiving $\mu(m)$

Part 1 (Induction base):

Let $h = 1$. That is, $P_j \equiv P_{i+1}$ in Figure 3.28. P_j cannot receive any m' , $m'_ts \leq m_ts$, after t due to the following two reasons:

- (a) P_i does not forward m to P_j ; also, P_i neither generates and send nor receives and forwards any m' , $m'_{ts} \leq m_{ts}$ after it deduced stability of $m_{ts} = TS$ by receiving m , with $m_{origin} = CN_i$; and,
- (b) The messages transmitted by P_i reach $P_j \equiv P_{i+1}$ in the order of transmission.

So, P_j cannot receive any $m'_{ts} \leq TS$ after it received $\mu(m)$ at t where $m_{ts} = TS$. Let us form the above statement as a statement for $h = 1$.

Note that the statement for $h = 1$ holds irrespective of the duration of the interval between P_i deducing the stability of TS and subsequently sending $\mu(m)$. That is, P_i could send or forward any number of messages in that interval.

Part 2 (Induction):

Let $h \geq 2$ and the statement for $(h - 1)$, $1 \leq h \leq N - 2$, be true: P_{j-1} (see Figure 3.28) cannot receive m' , $m'_{ts} \leq TS$, after it received $\mu(m)$ and deduced stability of TS for the first time. Since the statement for $(h - 1)$ is true, P_{j-1} cannot (receive and) forward any m' , $m'_{ts} \leq TS$ after it received $\mu(m)$ and deduced TS stability for the first time.

Moreover, $LC_{j-1} > m_{ts} = TS$, when P_{j-1} received m which happened before P_i received m and sent $\mu(m)$. That is, P_{j-1} cannot generate and send m' , $m'_{ts} \leq TS$, after it received $\mu(m)$. Therefore P_j will not receive any m' , $m'_{ts} \leq TS$, after it receives $\mu(m)$ and thereby deduced the stability of TS . This is the statement for h which holds true.

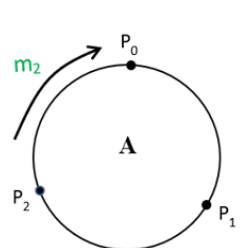
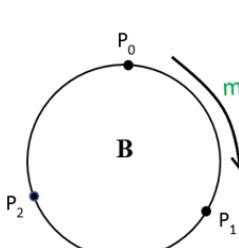
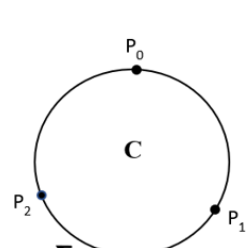
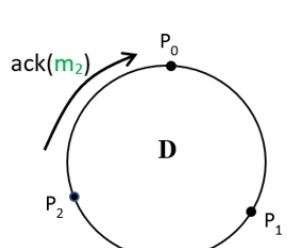
Thus, statement for $(h - 1)$ being true \Rightarrow statement for h being true. Therefore, the statement is true for all h , $1 \leq h \leq N - 1$. Hence, the Theorem is proved.

3.6 DCTOP Operations

In this section, to simplify our explanation, we demonstrated the functionality of DCTOP using a cluster of $N = 3$ processes. We examined scenarios involving a single multicast, and two concurrently sent multicast with equal timestamps at the start of the DCTOP protocol.

3.6.1 Single Multicast

Table 3-1 Single Multicast in DCTOP using $N=3$

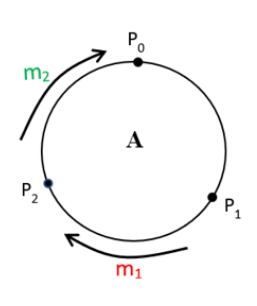
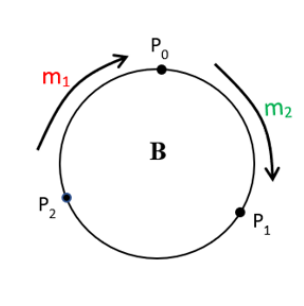
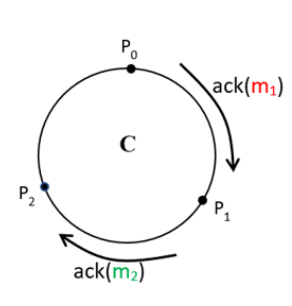
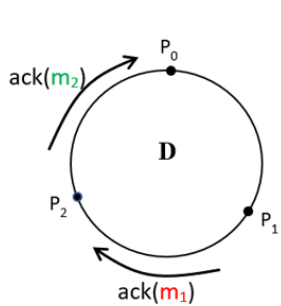
																											
<p>After 1 hop</p> <table><tr><td>P₀ LC₀= [1]</td><td>(m₂, 0, P₂)</td></tr><tr><td>P₁ LC₁= [0]</td><td></td></tr><tr><td>P₂ LC₂= [1]</td><td>(m₂, 0, P₂)</td></tr></table>	P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)	P ₁ LC ₁ = [0]		P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂)	<p>After 2 hops</p> <table><tr><td>P₀ LC₀= [1]</td><td>(m₂, 0, P₂)</td></tr><tr><td>P₁ LC₁= [1]</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>P₂ LC₂= [1]</td><td>(m₂, 0, P₂)</td></tr></table>	P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)	P ₁ LC ₁ = [1]	(m ₂ , 0, P ₂ , stable)	P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂)	<p>After 3 hops</p> <table><tr><td>P₀ LC₀= [1]</td><td>(m₂, 0, P₂)</td></tr><tr><td>P₁ LC₁= [1]</td><td></td></tr><tr><td>P₂ LC₂= [1]</td><td>(m₂, 0, P₂, stable)</td></tr></table>	P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)	P ₁ LC ₁ = [1]		P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂ , stable)	<p>After 4 hops</p> <table><tr><td>P₀ LC₀= [1]</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>P₁ LC₁= [1]</td><td></td></tr><tr><td>P₂ LC₂= [1]</td><td></td></tr></table>	P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂ , stable)	P ₁ LC ₁ = [1]		P ₂ LC ₂ = [1]	
P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)																										
P ₁ LC ₁ = [0]																											
P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂)																										
P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)																										
P ₁ LC ₁ = [1]	(m ₂ , 0, P ₂ , stable)																										
P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂)																										
P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂)																										
P ₁ LC ₁ = [1]																											
P ₂ LC ₂ = [1]	(m ₂ , 0, P ₂ , stable)																										
P ₀ LC ₀ = [1]	(m ₂ , 0, P ₂ , stable)																										
P ₁ LC ₁ = [1]																											
P ₂ LC ₂ = [1]																											

From Table 3-1, the process P_2 before sending m_2 to its CN_2 , P_2 , timestamp m_2 with its LC_2 , $m_2_{ts} = 0$, and after sending m_2 , it sets $LC_2 = 1 > 0$. Process P_2 retains a copy of the transmitted message, m_2 , in its $mBuffer_2$. Process P_0 receives the message and stores it in its $mBuffer_0$, deduces crashproof of m_2 but not its stability and sets $LC_0 = 1 > 0$. When P_1 receives m_2 , then m_2 has made a complete cycle at P_1 making m_2 crashproof and stable and is TO delivered by P_1 . Also, after receiving m_2 , P_1 sets $LC_1 = 1 > 0$ and $SC_0 = 0$.

Subsequently, P_1 prepares and sends $\mu(m_2)$ to P_2 with $Hops_{12} = 1$ since $f = 1$ for $N = 3$. P_2 receives $\mu(m_2)$, sets $SC_2 = 0$ after the third hop. P_2 deduces crashproof while it also deduces stability of m_2 and allowing for its TO delivery by the process P_2 . Thereafter, P_2 forwards a copy of $\mu(m_2)$ to P_0 since $\mu(m_2)$ indicated a higher stabilisation at P_2 ($\mu(m_2)_{ts} > SC_0$). After receiving the $\mu(m_2)$ by P_0 in the fourth hop, P_0 knows that m_2 is stable, sets $SC_0 = 0$ and also TO deliver m_2 . Process P_0 stops the forwarding of $\mu(m_2)$ since $\mu(m_2)$ originated from its CN_0 . Hence, for a single multicast, m_2 , using $N = 3$, DCTOP has the earliest TO delivery after 2 hops while the latest TO delivery occurs after 4 hops.

3.6.2 Concurrent Multicast

Table 3-2 DCTOP Operation with Concurrent Unicast

																																																			
After 1 hop	After 2 hops	After 3 hops	After 4 hops																																																
<table><tr><td>P₀</td><td>(m₂, 0, P₂)</td></tr><tr><td>LC₀= [1]</td><td></td></tr><tr><td>P₁</td><td>(m₁, 0, P₁)</td></tr><tr><td>LC₁= [1]</td><td></td></tr><tr><td>P₂</td><td>(m₂, 0, P₂)</td></tr><tr><td>LC₂= [1]</td><td></td></tr></table>	P ₀	(m ₂ , 0, P ₂)	LC ₀ = [1]		P ₁	(m ₁ , 0, P ₁)	LC ₁ = [1]		P ₂	(m ₂ , 0, P ₂)	LC ₂ = [1]		<table><tr><td>P₀</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>LC₀= [1]</td><td>(m₁, 0, P₁, stable)</td></tr><tr><td>P₁</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>LC₁= [1]</td><td>(m₁, 0, P₁, stable)</td></tr><tr><td>P₂</td><td>(m₂, 0, P₂)</td></tr><tr><td>LC₂= [1]</td><td>(m₁, 0, P₁)</td></tr></table>	P ₀	(m ₂ , 0, P ₂ , stable)	LC ₀ = [1]	(m ₁ , 0, P ₁ , stable)	P ₁	(m ₂ , 0, P ₂ , stable)	LC ₁ = [1]	(m ₁ , 0, P ₁ , stable)	P ₂	(m ₂ , 0, P ₂)	LC ₂ = [1]	(m ₁ , 0, P ₁)	<table><tr><td>P₀</td><td></td></tr><tr><td>LC₀= [1]</td><td></td></tr><tr><td>P₁</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>LC₁= [1]</td><td>(m₁, 0, P₁, stable)</td></tr><tr><td>P₂</td><td>(m₂, 0, P₂, stable)</td></tr><tr><td>LC₂= [1]</td><td>(m₁, 0, P₁, stable)</td></tr></table>	P ₀		LC ₀ = [1]		P ₁	(m ₂ , 0, P ₂ , stable)	LC ₁ = [1]	(m ₁ , 0, P ₁ , stable)	P ₂	(m ₂ , 0, P ₂ , stable)	LC ₂ = [1]	(m ₁ , 0, P ₁ , stable)	<table><tr><td>P₀</td><td></td></tr><tr><td>LC₀= [1]</td><td></td></tr><tr><td>P₁</td><td></td></tr><tr><td>LC₁= [1]</td><td></td></tr><tr><td>P₂</td><td></td></tr><tr><td>LC₂= [1]</td><td></td></tr></table>	P ₀		LC ₀ = [1]		P ₁		LC ₁ = [1]		P ₂		LC ₂ = [1]	
P ₀	(m ₂ , 0, P ₂)																																																		
LC ₀ = [1]																																																			
P ₁	(m ₁ , 0, P ₁)																																																		
LC ₁ = [1]																																																			
P ₂	(m ₂ , 0, P ₂)																																																		
LC ₂ = [1]																																																			
P ₀	(m ₂ , 0, P ₂ , stable)																																																		
LC ₀ = [1]	(m ₁ , 0, P ₁ , stable)																																																		
P ₁	(m ₂ , 0, P ₂ , stable)																																																		
LC ₁ = [1]	(m ₁ , 0, P ₁ , stable)																																																		
P ₂	(m ₂ , 0, P ₂)																																																		
LC ₂ = [1]	(m ₁ , 0, P ₁)																																																		
P ₀																																																			
LC ₀ = [1]																																																			
P ₁	(m ₂ , 0, P ₂ , stable)																																																		
LC ₁ = [1]	(m ₁ , 0, P ₁ , stable)																																																		
P ₂	(m ₂ , 0, P ₂ , stable)																																																		
LC ₂ = [1]	(m ₁ , 0, P ₁ , stable)																																																		
P ₀																																																			
LC ₀ = [1]																																																			
P ₁																																																			
LC ₁ = [1]																																																			
P ₂																																																			
LC ₂ = [1]																																																			

As illustrated in Table 3-2, the processes P_2 and P_1 before sending m_2 and m_1 concurrently to their CN_2 and CN_1 . P_2 , timestamp m_2 with its LC_2 , $m_{2_ts} = 0$, and after sending m_2 , it sets $LC_2 = 1 > 0$. Also, P_1 , timestamp m_1 with its LC_1 , $m_{1_ts} = 0$, and after sending m_1 , it sets $LC_1 = 1 > 0$. After 1 hop, P_0 received the m_2 message from P_2 while P_2 also received the m_1 message from P_1 . P_2 forwards a copy of m_1 to P_0 and P_0 forwards a copy of m_2 to P_1 concurrently. After 2 hops P_0 receives m_1 and P_1 receives m_2 . When P_0 receives m_1 , it knows that m_1 is crashproof ($r = f + 1$ distinct processes have received m_1 , $f = 1$ for $N = 3$ cluster) and stable (P_0 is the ACN_0 of m_1 origin). Also m_2 is crashproof (since $r = f + 1$ distinct processes have received m_2 , $f = 1$ for $N = 3$ cluster) and stable since $m_{1_ts} = m_{2_ts}$. Similarly, When P_1 receives m_2 , it knows that m_2 is crashproof ($r = f + 1$ distinct processes m_2 have receive and $f = 1$ for $N = 3$ cluster) and stable. Also m_1 is stable since $m_{1_ts} = m_{2_ts}$ but P_1 does not know m_1 is crashproof, it can only know by receiving $\mu(m_1)$. Hence, P_0 TO deliver m_1 and m_2 after 2 hops. Subsequently, after the third hop, P_1 and P_2 TO deliver m_1 and m_2 simultaneously after receiving the $\mu(m_1)$ and $\mu(m_2)$. P_2 knows that m_1 is also stable (m_{1_ts} is already known to be stable due to timestamp equality) and crashproof since m_1 has been received by at least $f + 1$ distinct processes (in this case, $f = 1$ see assumption AS1). In the fourth hop, when P_2 and P_1 receive $\mu(m_1)$ and $\mu(m_2)$ they both discard it. Hence, for two concurrent messages, m_1 and m_2 using $N = 3$, DCTOP has the earliest TO delivery after 2 hops while the latest TO delivery occurs after 3 hops.

3.6.3 Performance Benefits of DCTOP

Our simulation results show that these unique attributes (i) the Lamport logical clock and, (ii) the dynamically determined last process to order concurrent messages, guarantee high throughput and improved latency despite the number of processes sending concurrent messages, unlike the LCR. Latency is further improved when a single concurrent multicast message occurs with an equal timestamp between two processes, as illustrated in Sections 3.5.3 and 3.6.2. This particular case cannot be applied to multiple concurrent multicast messages since we cannot deterministically control the distribution during the simulation.

3.7 Fairness-Controlled Environment

We defined *fairness* as every process P_i having an equal chance of sending own messages and having those sent messages eventually delivered by all processes within the system. Every process ensures messages from the ACN are forwarded in the order they were received before sending their own message. Therefore, no process has priority over another during the sending of messages. If P_i sends 10 messages and manages to get 10 messages delivered, then every other process should be able to do the same thing in that time frame. The discussion in the preceding sections did not consider the fairness aspects of the process sending its own messages within the DCTOP environment cluster. The DCTOP discussed thus far used a greedy sending approach, so each process could send more messages than the other processes, thus appearing to have priority of message multicast over others.

In a ring structure, a process P_i send its own messages to the CN_i and also forwards messages from ACN_i whose origins are from other processes. Thus, seeking a balance, where every process has an equal opportunity to send their own messages is fairness. We modified FSR [1] fairness primitive and adopted the modification into our system design to bring fairness into our system design.

The advantageous difference between DCTOP fairness control primitive to FSR fairness is that FSR fairness considers only a case where the incoming queue is not empty and the process P_i has a message to send. So, before sending its own message m , P_i forwards messages whose origin is not yet in its forward list. However, unlike FSR fairness control (see Section 2.5.2) we consider that a process P_i can only send one message if (i) the incoming queue is empty (ii) the incoming queue is not empty and it had either forwarded exactly one message originating from every other process or after it had forwarded a message originating from some P_j and the message currently at the head of the incoming buffer is another message originating also from

the same P_i . It is worth noting that the forwarding of messages happens according to the order it was received. Section 3.7.2 presents several illustrative scenarios of DCTOP fairness control primitives.

3.7.1 Fairness Performance Model

In this section, we discuss the DCTOP fairness mechanism: for a given round k , any process P_i either sends its own message to the CN_i or forwards messages from its ACN_i to the CN_i . A round (see Section 2.7.1) is defined as follows: for any round k , every process P_i sends at most one message, m , to its CN_i and also receives at most one message, m , from its ACN_i in the same round. Every process P_i has an incoming queue which contains the list of all messages P_i received from the ACN_i which was sent by other processes, and a *SendingQueue_i*. The *SendingQueue_i* consist of the messages generated by the process P_i waiting to be transmitted to other processes. When the *SendingQueue_i* is empty, the process P_i forwards every message one at time in its incoming queue but whenever the *SendingQueue_i* is not empty, a rule is required to coordinate the sending and forwarding of messages to achieve fairness.

Fairness Control Rules

Suppose that process P_i has one or more message(s) to send stored in its *SendingQueue_i*, it follows these rules before sending each message in its *SendingQueue_i* to the CN_i : process P_i sends exactly one message in *SendingQueue_i* to the CN_i if

- (1) the *IncomingQueue_i* is empty, or
- (2) the *IncomingQueue_i* is not empty and either
 - (2.1) P_i had forwarded exactly one message originating from every other process or
 - (2.2) the message at the head of the *IncomingQueue_i* originates from a process whose message the process P_i had already forwarded.

To implement these rules and verify rules 2.1 and 2.2, we introduced a data structure called *forwardlist*. The *forwardlist_i* at any time consists of the list of the origins of the messages that process P_i forwarded ever since it last sent its own message. Obviously by definition, as soon as the process P_i sends a message, the *forwardlist_i* is empty. Therefore, if the process P_i forwards a message that originates from the process P_{i-1} , $i > 0$, which was initially in its *IncomingQueue_i*, then process P_i will contain the process P_{i-1} in its forward list, and whenever it sends a message the process P_{i-1} will be deleted from the *forwardlist_i*.

3.7.2 Fairness Control Scenarios

This section discusses three cases we drew up to illustrate how DCTOP handles fairness control during the simulation of our system design using $N = 4$ process cluster and P_0 as the process under consideration. The arrow sign denotes the order of message arrival at both the *SendingQueue*₀ and *IncomingQueue*₀ and the origin of the forwarded message's arrival at the *forwardlist*₀ according to the designated arrow. It is worth noting that messages in the *SendingQueue*₀ are not timestamps. However, just before sending any message from the *SendingQueue*₀, it is timestamped and sent, after which the LC_0 is updated. Similarly, we assumed that messages in the *IncomingQueue*₀ have already been received and LC_0 updated but not yet forwarded.

Case 1: Incoming Queue is empty.

Here, process P_0 has two messages, m_0 , and m'_0 stored in the *SendingQueue*₀ waiting to be transmitted and at the same time the *IncomingQueue*₀ is empty. Hence, rule 1 is applicable, as shown in Figure 3.29a. Therefore, P_0 then send m_0 immediately to the CN_0 after timestamping m_0 and after which it updates the LC_0 . The message disappears from the *SendingQueue*₀ suggesting that P_0 has successfully sent m_0 , as illustrated in Figure 3.29b. Subsequently, the process P_0 checks *IncomingQueue*₀ to determine if it is empty. If *IncomingQueue*₀ is not empty, the process P_0 checks which rule is applicable; however, since it is still empty, the process P_0 proceeds to send m'_0 , as shown in Figure 3.29c. For illustration purposes, we marked the forwarded received message as green while the received message but not forwarded as red for all cases considered.

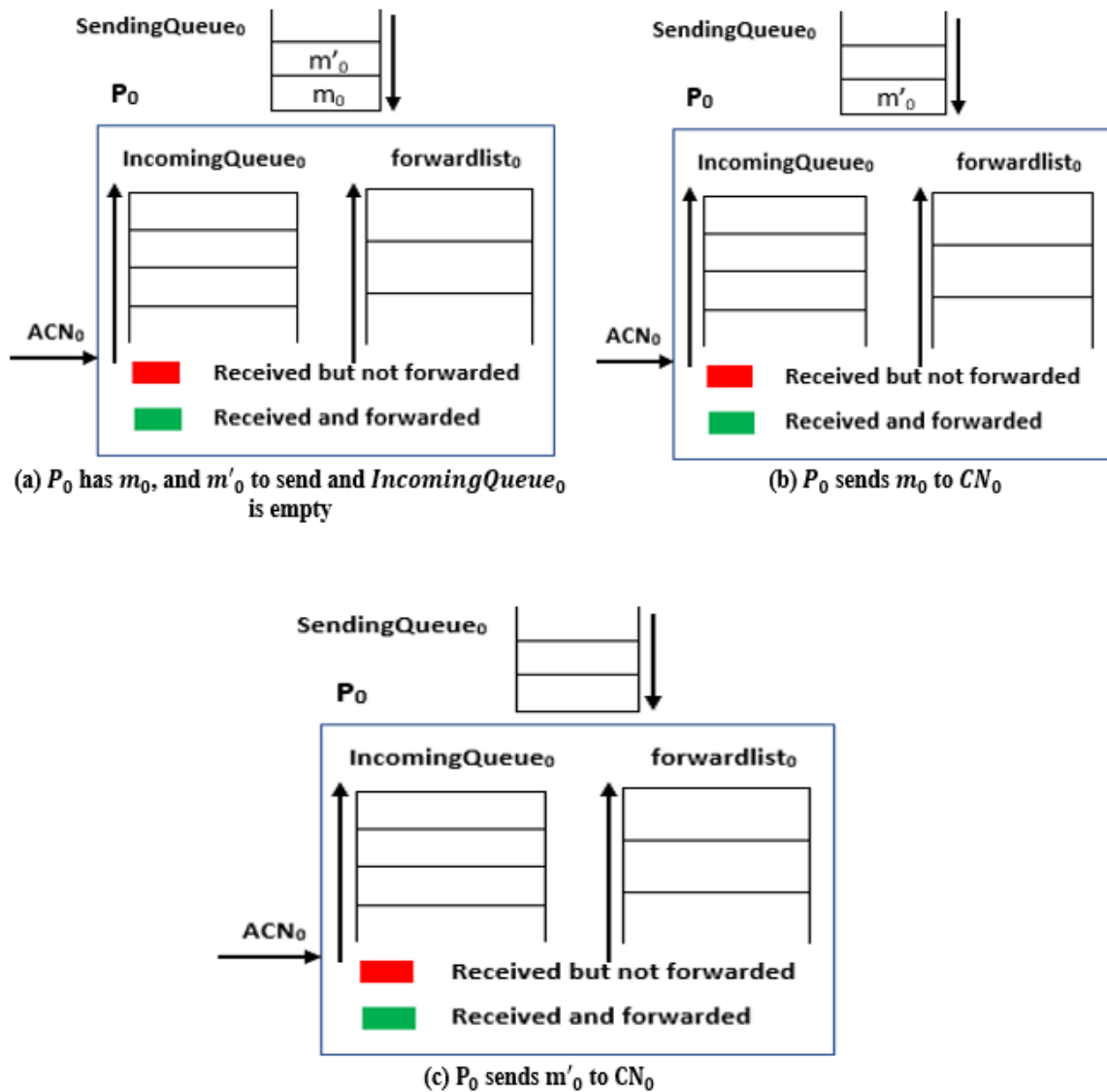


Figure 3.29 Incoming Queue is empty.

Case 2: Incoming Queue is not empty.

Here, process P_0 has one own message, m_0 , stored in the $SendingQueue_0$ waiting to be sent and at the same time the $IncomingQueue_0$ is not empty, and rule 2 is applicable, as shown in Figure 3.30a. This means that P_0 have messages it has received and updated the LC_0 but not forwarded. The process P_0 forwards exactly one message from every other process before sending its own message. Thus, the process P_0 forwards m_1 to the CN_0 , as shown in Figure 3.30b. It then checks the $IncomingQueue_0$ again, finding it still not empty and rule 2 remains applicable, leading to the forwarding of m_2 as shown in Figure 3.30c. The process P_0 repeats this step, forwarding m_3 (Figure 3.30d), and upon checking $IncomingQueue_0$ again, finding it empty, it applies rule 2.1. P_0 then sends its message and resets the $forwardlist_0$, as shown

in Figure 3.30e. The message, m_0 , is removed from the $SendingQueue_0$ suggesting a successful sending of messages.

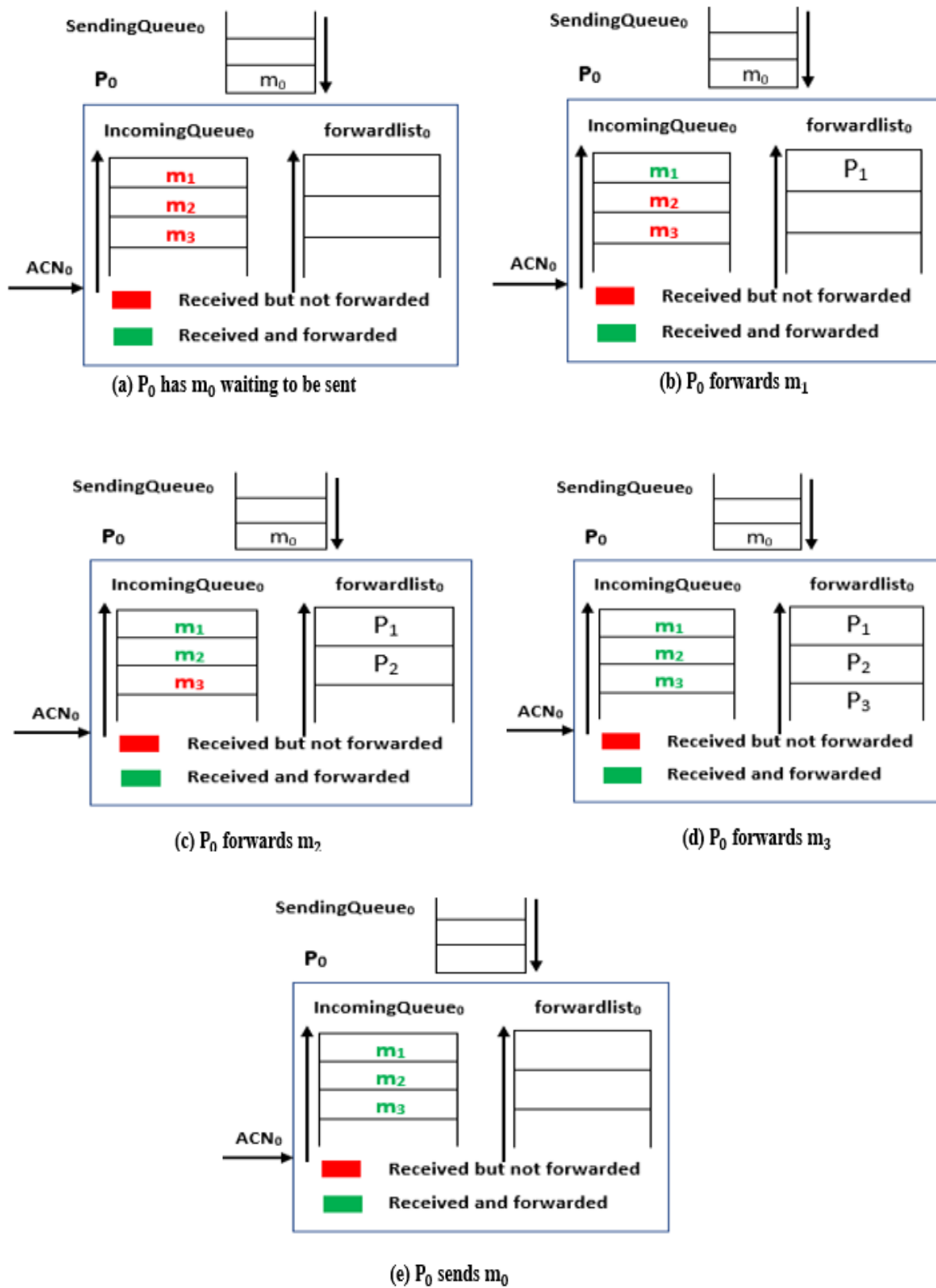
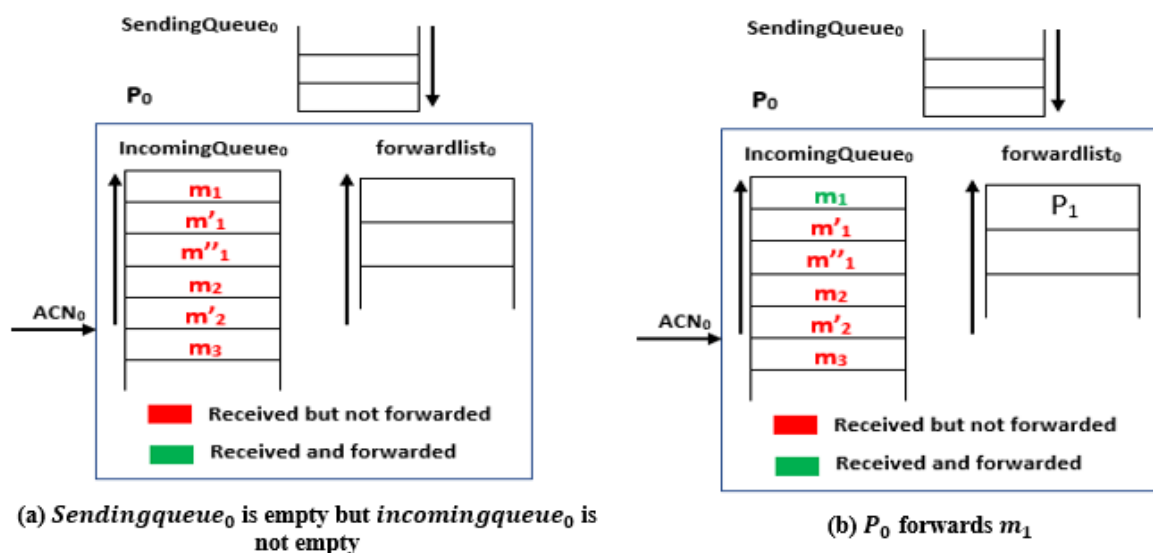


Figure 3.30 Incoming Queue is not empty.

Case 3: Sending Queue is empty.

In this scenario, we explore a situation where the *SendingQueue*₀ is empty but the *IncomingQueue*₀ is not empty as illustrated in Figure 3.31a. In such cases, when no rule is applicable, the process P_0 proceeds to forward the messages to the CN_0 one at a time.

Accordingly, the process P_0 forwards m_1 which is at the head of the *IncomingQueue*₀ to the CN_0 , as shown in Figure 3.31b. Subsequently, P_0 checks if the *SendingQueue*₀ is still empty, finding it empty, no rule is applicable, leading to the forwarding of m'_1 (Figure 3.31c). The process repeats this step and because it is still empty no rule is applicable still, then it forwards m''_1 as depicted in Figure 3.31d. Again, it checks the status of *SendingQueue*₀ and because it is still empty, it forwards m_2 as shown in Figure 3.31e. Subsequently, it checks the *SendingQueue*₀ and determined that it is no longer empty, but the message gets stuck as neither rule 2.1 nor rule 2.2 is applicable. Consequently, the process P_0 forwards m'_2 as shown in Figure 3.31f. The process P_0 cannot send its own message presently as the message at the head of the *IncomingQueue*₀, m_3 , did not originate from a process whose message the process P_0 had already forwarded, that is rule 2.2 is not applicable. Hence, the process P_0 forwards m_3 as shown in Figure 3.31g. Finally, with rule 1 applicable, the process P_0 sends its own message and resets the *forwardlist*₀ as shown in Figure 3.31h.



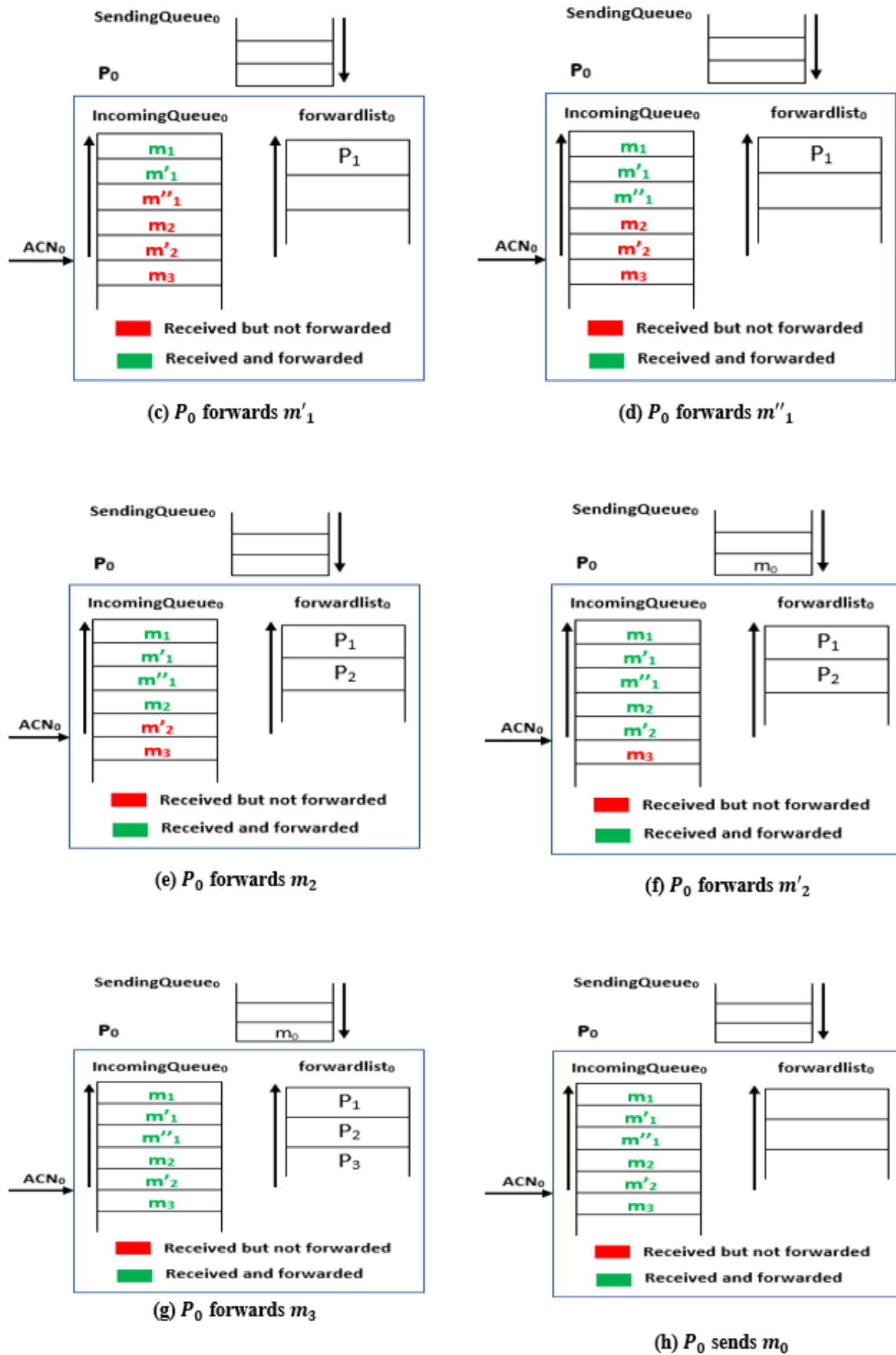


Figure 3.31 The Sending Queue is empty at some point.

3.8 Summary

This chapter presented DCTOP, an alternative ring-based leaderless order protocol that uses two primitives- uniform daisy chain multicast and non-uniform, total order multicast, to create an efficient multicast and uniform total order delivery solution. We presented the details of our new protocol's system design and the total ordering requirements before presenting the total order delivery and the deliverability requirements of the two primitives. However, the key difference between DCTOP and LCR is as follows: (i) DCTOP employs a logical clock for timestamping messages, which is simple and lightweight, requiring only a single integer value to represent time. This reduces the complexity and overhead involved in tracking the order of messages across the system. In contrast, LCR relies on a vector clock for doing vector timestamps on the messages. This vector timestamp uses a vector of integers, one for each process, to maintain causal relationships. Also, vector timestamp can lead to higher overhead as the number of processes increases since each message must include vector timestamps before sending. (ii) DCTOP introduces a flexible and adaptive approach to ordering concurrent messages. It uses a dynamically determined process replica to serve as the ordering node, which allows for more efficient handling of concurrency. This dynamic selection helps avoid bottlenecks and adapts to varying system conditions. On the other hand, LCR adopts a rigid approach by relying on a globally fixed "last process" to order all concurrent messages. While simple, this fixed last process can become a single point of contention and may not rapidly sequence multiple concurrent messages as load increases. (iii) DCTOP assumes $N=2f+1$ while LCR assumes $N=f+1$. The more relaxed fault tolerance assumption in DCTOP allows for lower latency by enabling message delivery once crashproofness is achieved, even if the ACN of the message origin has not yet declared the message stable. This is possible because message timestamps can stabilize quickly during concurrent transmissions. In contrast, LCR's stricter assumption requires that a message be received by all processes before it can be delivered, which can significantly increase latency. Finally, we presented the modified FSR fairness-controlled primitive specifically for use within our new protocol and the LCR during the simulation of both protocols. The next chapter presents a comprehensive performance evaluation of DCTOP and LCR using (i) without fairness primitive (greedy sending approach) and (ii) fairness control primitives.

Chapter 4

Comparative Performance Evaluation of Leaderless Protocols

This chapter provides an extensive performance evaluation of the essential concepts of leaderless protocols introduced in this thesis (see Section 2.7.1 and Chapter 3). The evaluation focuses on two key issues:

- i. A performance comparison between DCTOP and LCR using a greedy sending approach (without fairness control primitive) see Section 3.3.1.
- ii. A performance comparison between DCTOP and LCR protocols using fairness control primitive as described in Section 3.7.

We organised the remainder of this chapter as follows: the first section details the motivation for conducting a performance comparison between the DCTOP and the LCR protocols as described in Section 4.1. The second presents the details of the simulation and evaluation of (i) DCTOP and LCR under variable group size, N , using a greedy sending approach (without fairness primitives), as described in Section 4.2. Section 4.3 introduced a simulation that evaluates (ii) DCTOP and LCR using fairness control primitives under varying group sizes. We maintained in both simulations the same message arrival rate λ , using Poisson distribution, message transmission delay μ , exponentially distributed, and the same simulation durations.

4.1 Motivation

The primary justification for comparing the performance evaluation of DCTOP and LCR protocols is that, in certain exceptional scenarios, LCR might perform better than DCTOP or both could have a comparable performance, while in alternative situations, DCTOP could demonstrate superior performance compared to LCR. Therefore, we utilized the diagrams in Table 2-3 and Table 2-4 to illustrate the simplified workings of the LCR protocol while Table 3-1 and Table 3-2 show the simplified workings of DCTOP within a process cluster of $N = 3$

under some edge cases (a single and concurrent multicast). Notably, DCTOP exhibits a comparable operational behaviour to LCR when considering a single multicast operation as evidenced from Table 2-3 and Table 3-1. Therefore, for the $N = 3$ process cluster and using a single multicast, both protocols exhibit comparable operational behaviour, with the earliest TO delivery occurring after two hops and the latest TO delivery occurring after four hops. Moreover, for two concurrent messages, m_1 and m_2 as illustrated in Table 2-4 and Table 3-2 LCR has the earliest TO delivery after 2 hops while the latest TO delivery occurs after 4 hops. Conversely, DCTOP has the earliest TO delivery after 2 hops (more messages TO delivered in DCTOP compared to LCR) while the latest TO delivery occurs after 3 hops which is better compared to LCR.

However, these examples, as shown in Table 2-3, Table 2-4, Table 3-1 and Table 3-2, are insufficient to assert that DCTOP is better than LCR since our interest lies in the average maximum latencies incurred when a process performs TO delivery for concurrent message multicasts. This underscores the motivation for conducting a comprehensive performance comparison of DCTOP and LCR through simulations. In addition, it is challenging to theoretically and analytically articulate every scenario that occurs during concurrent message multicasts within the comparisons of DCTOP and LCR operations; hence the need for discrete event simulation-based evaluations.

4.2 Greedy Sending Approach (Without Fairness Primitives)

This section describes the simulation and performance comparison of LCR and DCTOP using the greedy sending approach. We hypothesise that DCTOP approaches improve the LCR latency performance if DCTOP offers lower average maximum latencies compared to LCR for the considered cluster group sizes. Thus, we developed an experiment that performs discrete event simulations [146] for both ring-based ordering protocols to test our hypothesis. This experiment simulates the state changes and underlying communication phases the LCR and DCTOP require to deliver concurrent message multicasts in total order without utilising the fairness primitive. The use of without fairness primitive means that a process might have priority over other processes during message multicast. The comparison of DCTOP and LCR focuses on the average maximum latency required to perform total order delivery of concurrent message multicasts, and the average maximum throughput which is at the heart of the research report in this thesis. In our simulation, it can be inferred that DCTOP's performance enhances the LCR if it can demonstrably decrease latency and possibly produce similar throughput since

both are ring-based leaderless protocols, which are known to offer maximum attainable throughput. Therefore, we assume that our hypothesis is correct if our simulation results reveal that DCTOP consistently outperforms LCR in terms of latency. We evaluated the effectiveness of both protocols in a uniform setting by utilising the same simulation structure and workloads across the protocols being evaluated.

4.2.1 Simulation

Assumptions:

- (i) **Inter-Transmission Times:** The time between successive message transmissions is assumed to be exponentially distributed with a mean of 30 milliseconds. This reflects the memoryless property of the exponential distribution, suitable for modelling independent transmission events.
- (ii) **Inter-Transmission Delay:** The delay between the end of one message transmission and the start of the next is assumed to be exponentially distributed with a mean of 3 milliseconds. This approach ensures that delays are modelled realistically, capturing the inherent randomness of network delays, ensuring realistic and stochastic behaviour in simulations.
- (iii) **Node Reliability:** We assumed that process replicas have 100% uptime since we did not consider crash failure scenarios during the simulation.
- (iv) **Message Loss:** We assumed that there is no message loss. This means that every message sent between processes is successfully delivered without any failures.
- (v) **System Size:** Simulations were conducted with a variable process replica size. E.g., 4, 5, 7, and 9 processes.
- (vi) **Arrival Rate:** The arrival rate of messages is assumed to follow a Poisson arrival rate with an average rate of 40 messages per second. This models the randomness and variability typically observed in real-world systems.
- (vii) **Simulation Duration:** The simulation runs between a total of 40000 seconds and 1000000 seconds. This duration is chosen to ensure that the system reaches a steady state and that sufficient data points are collected for a 95% confidence interval analysis. The long duration also ensures that at least one million (1000k) messages to 25 million (25000k) messages are sent by each process and delivered by all processes.

The actual number of processes that can crash, f (Assumption AS1), in the DCTOP, has been excluded from this simulation since we only evaluated the performance of the two techniques

in a crash-free situation. Our rationale for removing f was that the crash-tolerance provisions described in DCTOP are the only possible solution for ensuring that the protocol can continue to execute even if f processes crashed during a multicast. Other total order solutions are not required to use this additional communication phase. Furthermore, in our simulation, we are comparing DCTOP to the LCR protocol, which has never simulated any method to deal with a crashed process; hence, eliminating f from the DCTOP protocol allows for a more accurate comparison of the two protocols.

Detailed Description of Simulation Design and Implementation

We performed a discrete event simulation [147] of DCTOP and LCR protocols using the Java (OpenJDK-17, Java version 17.02) framework with variable group sizes, $N = 4, 5, 7$, and 9 processes. All processes in the simulation utilised commodity PCs of 3.00GHz 11th Gen Intel(R) Core (TM) i7-1185G7 CPU and 16GB of RAM.

Simulation Workflow: Our simulation design is illustrated in the diagram located in Appendix A1. This simulation is repeated 10 times and is explained in detail as follows below:

Initialization:

1. Create N processes, each with a unique ID.
2. Set up the ring by linking each process to its CN.
3. Initialize LC and SC to 0.
4. Create empty buffers and queues //mBuffer = empty buffer // DQ and GCQ = empty queues.
5. Generate N send events ordered by event time and enqueue into the event queue.
6. Pick up the event (Send or Receive) from the head of the event queue

Events:

1. **Send:** A process sends a message (M) to the CN in the ring
2. **Receive:** A process receives a message (M) from the ACN in the ring

Event Handlers:

Send Event:

1. If the picked-up event is a send event then create a message with the senders LC together with the data structure explained earlier. Thus, for a send event, any process P_i

can generate, timestamps and send a message (M) to its CN_i (the message sender deposits a copy into the receiving buffer) while storing a copy of the message in its $mBuffer_i$, after which it increases the value of its logical clock by 1. Note that M can be m or $\mu(m)$.

2. P_i also creates a Receive Event for the message it sends to the CN_i , and the Next Send Event for subsequent message sending. It then updates the Event Queue (EQ) with the receive and next send events.

Receive Event:

1. The received message is stored in the $mBuffer_i$ of the receiving process and the LC is updated as specified in the pseudocodes. Who then forwards a copy of the message to its CN_i if its CN is not the origin of the message. Note that if the process P_i wants to send its own message m' after receiving a message m from ACN_i , it will timestamp m' using the updated logical clock to reflect that the received m happened before the sending of m' .
2. When a message is forwarded, then the forwarding process also creates a receive event and enqueues it to the event queue for the receiving process.
3. Once M has been received by all processes, the ACN of M origin knows that M is stable and attempts to deliver M in total order. It then communicates this stability to other processes by sending $M=\mu(m)$ in DCTOP or $ack(m)$ in LCR.
3. When all processes have received and delivered $multicast(m)$ as per the delivery requirements of the DCTOP and LCR protocols, the $multicast(m)$ is considered to be complete.
4. After a total order delivery is attempted, a new send event is created and enqueued into the event queue and this continues until the simulation duration terminates.

Performance Metrics

The performance of the two approaches is measured based on the average maximum latency and average maximum throughput.

Latency: In LCR and DCTOP, latency is calculated as the time elapsed between a process's initial multicast m to its clockwise neighbour (CN) within the cluster and all members of the m destination delivering the m to the high-level application process. For example, let t_0 and t_l be the time when P_0 sends a message to its CN_0 and the time when the ACN (ACN_0) delivers

that message in total order respectively. Then $t_l - t_0$ defines the maximum latency delivery for that message. The average of 1000k to 25000k messages of such maximum latencies was computed, and the experiment was repeated 10 times for a confidence interval of 95%.

Throughput: For both approaches, we computed the throughput as the average number of total order messages delivered (aNoMD) by any process during the simulation time calculated, like latencies, with a 95% confidence interval. Furthermore, we report latency and throughput improvements offered by the proposed protocol, DCTOP, over LCR and were measured as follows: Let U and U_v be metrics for LCR and DCTOP approaches, respectively; improvement in latency (L) is $\frac{L-L_v}{L}$ and that in throughput (T) is $\frac{T_v-T}{T}$. Thus, a positive value implies that the proposed protocol is better.

Note: We isolated all our experiments to avoid unintended consequences from running multiple experiments concurrently on the same cluster. However, we spread out the execution of each experiment over roughly the same amount of time to maintain a consistent load on the ring-based network across all of our experiments.

4.2.2 Evaluation

This section focuses on the performance evaluation of LCR and DCTOP under different group sizes of N . We considered when $N = 4, 5, 7$, and 9 for all performance evaluations.

Latency

Figure 4.1 shows the average maximum latency comparison between DCTOP and the LCR protocols for all the group sizes of N considered. Each plot on the graph is an average of ten crash-free experiments, with each process completing sending messages of varying numbers ranging from 1000k to 25000k messages for a given simulation time. Thus, across all ten experiments, each process in the DCTOP and LCR protocols sends and receives an average of 1000k to 25000k messages delivered in total order according to FIFO. Here, we considered only when every process is sending messages concurrently. Starting with the latency shown in Figure 4.1a. The figure indicates that DCTOP has lower latencies than LCR for all messages sent during the simulation, with an 11% average maximum latency improvement. This can be attributed to the presence of the Lamport logical clock for message timestamping, the dynamically determined last process for each sender for ordering concurrent messages, and the crash failure assumption relaxation which enables crashproof messages to be delivered even before they are declared stable by the ACN of message origin since timestamp stability can

happen quickly during concurrent message multicast. When considering the average maximum latency for group size $N = 5, 7, 9$, we observe a similar trend between DCTOP and LCR as we did between DCTOP and LCR when $N = 4$. What is different is the varying percentage improvement of average maximum latency: for $N = 5$, DCTOP offered a 7% average maximum latency improvement; for $N = 7$, DCTOP showed a 6% average maximum latency improvement; and for $N = 9$, DCTOP offered an 8% average maximum latency improvement. The results obtained from the simulation demonstrate that in terms of average maximum latency, DCTOP outperforms LCR irrespective of the group sizes, though with differing degrees of improvement, with a group size of 4 producing the highest latency improvement, 11%. The consistent trend suggests that the advantages provided by DCTOP, such as the message timestamping mechanisms, unique last process for each sender, and a relaxation of the crash failure assumption, continue to contribute to lower latencies as the system scales.

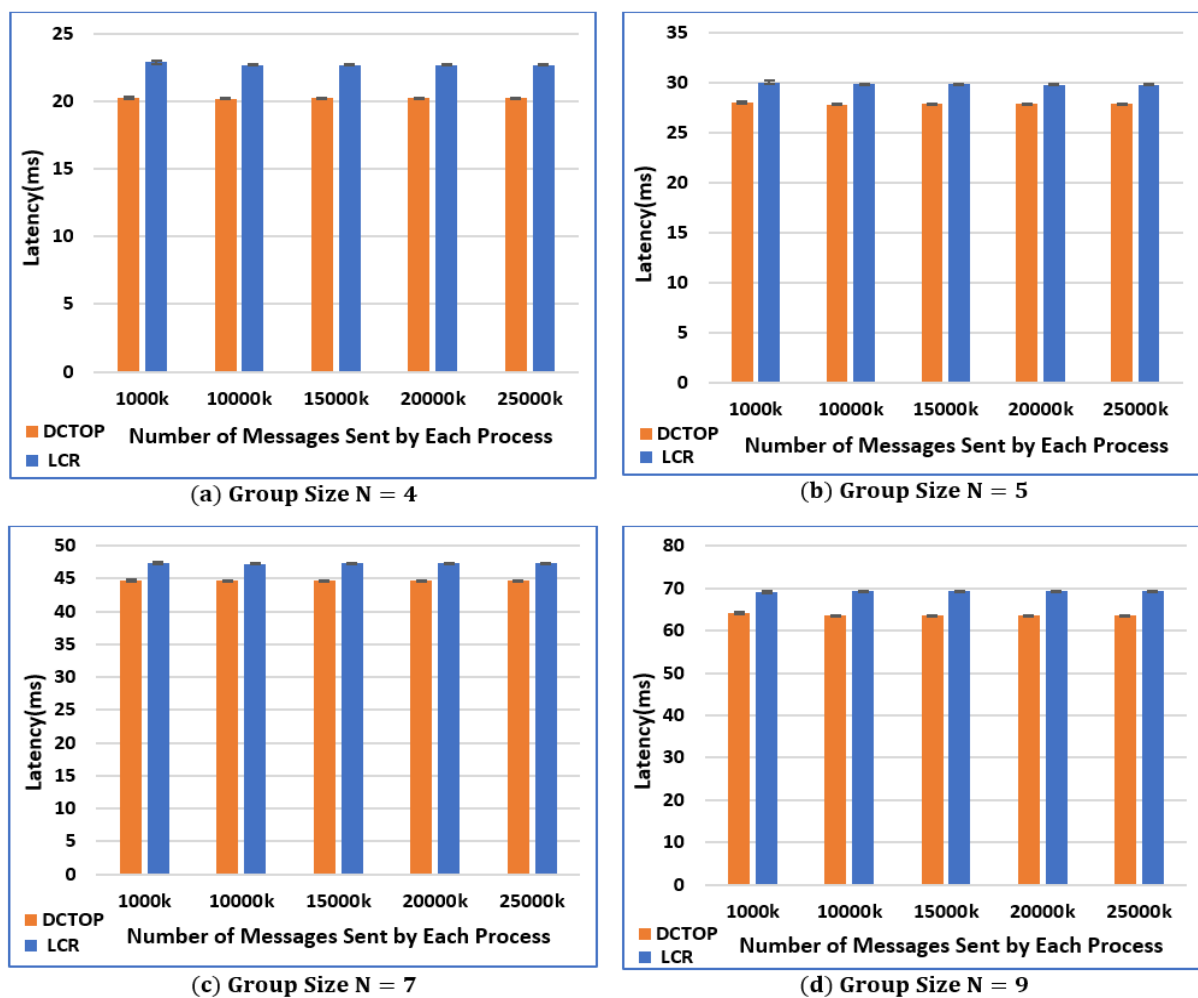


Figure 4.1 Latency Comparison without Fairness Primitive

Throughput

Figure 4.2 depicts the throughput results of our simulation for all N and the varying number of messages transmitted during the simulation between 1000k and 25000k messages.

DCTOP and LCR have almost comparable throughput across all message counts transmitted during the simulation, as shown in Figure 4.2a. This indicates that the throughput capabilities of DCTOP and LCR are similar. The high throughput observed here can be explained by the efficiency of ring-based leaderless total order protocols. Protocols like LCR [2], and findings from DCTOP demonstrate that organizing processes in a ring leads to optimal use of resources, resulting in the highest feasible throughput for message delivery.

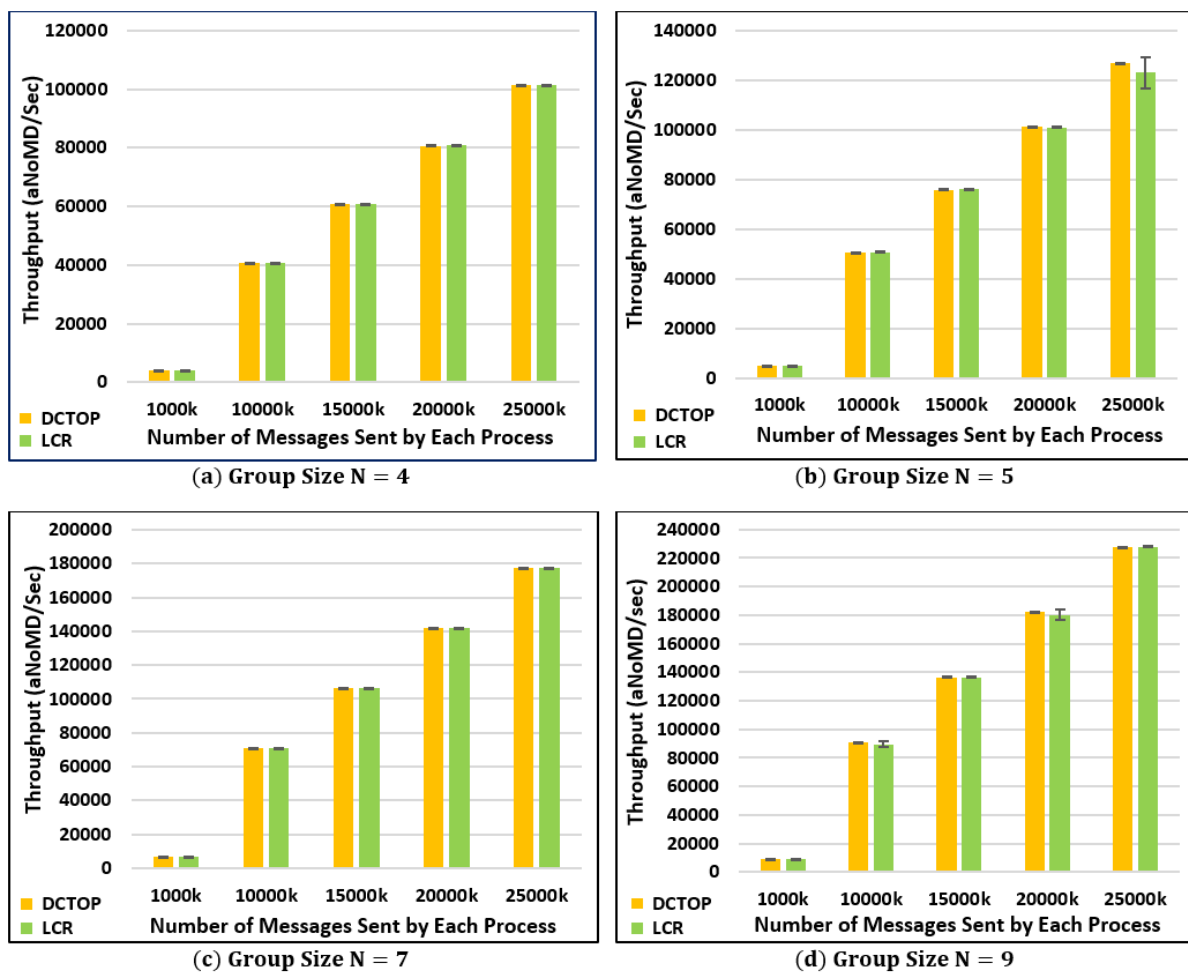


Figure 4.2 Throughput Similarity without Fairness Primitive

When considering the average maximum throughput for group sizes $N = 5, 7$, and 9 , a similar trend is observed between DCTOP and LCR as was observed when $N = 4$ (see Figure 4.2b, Figure 4.2c and Figure 4.2d). This suggests that the relationship between the two protocols remains consistent as the group size increases.

However, the average maximum throughput values slightly vary for each group size. The results show that DCTOP and LCR perform comparably similarly in terms of throughput, notwithstanding the number of messages sent during the simulation time. This implies that both protocols can process messages effectively and oversee the additional workload with a slight change in throughput. However, the evaluation presented in this thesis is unbiased because LCR and DCTOP are built out of an identical code base and utilize the same performance evaluation setup and hardware attributes.

4.2.3 Summary

Without using fairness primitive in our simulation, the results clearly show a latency improvement over the well-known LCR protocol under its original use of a vector clock for message vector timestamping and a notion of the fixed “last” process to order concurrent messages. We deduced three critical findings from the performance comparisons: firstly, the Lamport logical clock used for message timestamping does bring performance benefits for all values of N considered during the simulations; secondly, our unique last process mechanism is an effective alternative to a globally fixed last process as evidenced in the LCR protocol because DCTOP improved latency in all the scenarios we considered. Thirdly, the relaxation of the crash failure assumption also helps to improve latency as message delivery happens quickly even before they are declared stable by the ACN of the message origin. We also observe that as the group size increases, the throughput increases, as shown in Figure 4.2. Thus, adding more processes into the group, on the one hand, improves throughput, but on the other hand, it consumes more compute resources. Hence, the trade-off between increased throughput and computing resource consumption should be considered when scaling the system.

4.3 With Fairness Primitive

In the previous section, we tested the performance of DCTOP and LCR by varying the number of processes in the group size N and the number of messages transmitted. The result showed that DCTOP consistently improved latency for all N considered. This led us to conclude that DCTOP is a desirable alternative to LCR based on the number of messages transmitted, different replication degrees N and latency improvements. We observed that without using a fairness control primitive, some processes within the group may have message-sending priority over other processes, as earlier stated.

In this section, we included a fairness control primitive in our simulation to ensure that all group processes have an equal chance of transmitting messages. This will help to resolve any imbalances in message-sending priorities. Thus, by adding fairness control, we aim to provide an equal opportunity for all processes to transmit messages and then evaluate the usefulness of DCTOP and LCR under this new restriction. We measured the impact of fairness control on the number of messages sent, different replication degrees (N), and overall performance improvements.

The performance of DCTOP and LCR with fairness primitives is compared to their performance without fairness control as part of the evaluation process. We conducted the simulation experiments by varying the number of processes in the group (N) and the number of messages transmitted using the same assumption and simulation detail as stated in Section 4.2.1. The only difference between the fairness control simulations and the simulation in Section 4.2.1 is that before any process can perform a send or receive event, it must observe the fairness control rules as stated in Section 3.7.1. This will allow us to observe how fairness control impacts the performance of both protocols across different scenarios. Based on the simulation results, we can establish if introducing fairness primitives affects the performance of DCTOP and LCR and observe any performance differences between the two protocols under fairness-controlled conditions. This evaluation will bring insight into the effectiveness and applicability of DCTOP and LCR in settings where fairness control is critical.

4.3.1 Evaluation

This section primarily investigates the performance evaluation of the LCR and DCTOP protocols incorporated with fairness primitive under variable group sizes of N . The evaluations were performed for group sizes of 4, 5, 7, and 9 to compare the performance of both approaches.

Latency

Figure 4.3 compares the average maximum latency of the DCTOP and LCR protocols across various group sizes (N). The graph depicts the average of ten simulations in which variable messages, ranging from 1000k to 25000k messages, were transmitted during a predetermined simulation duration. The DCTOP and LCR approaches guarantee that each process receives messages following the FIFO principle to maintain total order in all ten simulations. The comparison is based on steady-state data from the simulations while ensuring

that every process simultaneously transmits messages. The simulation results illustrated in Figure 4.3a show that we examine both approaches within the same simulation period; the DCTOP consistently achieves lower latencies than LCR for all messages transmitted.

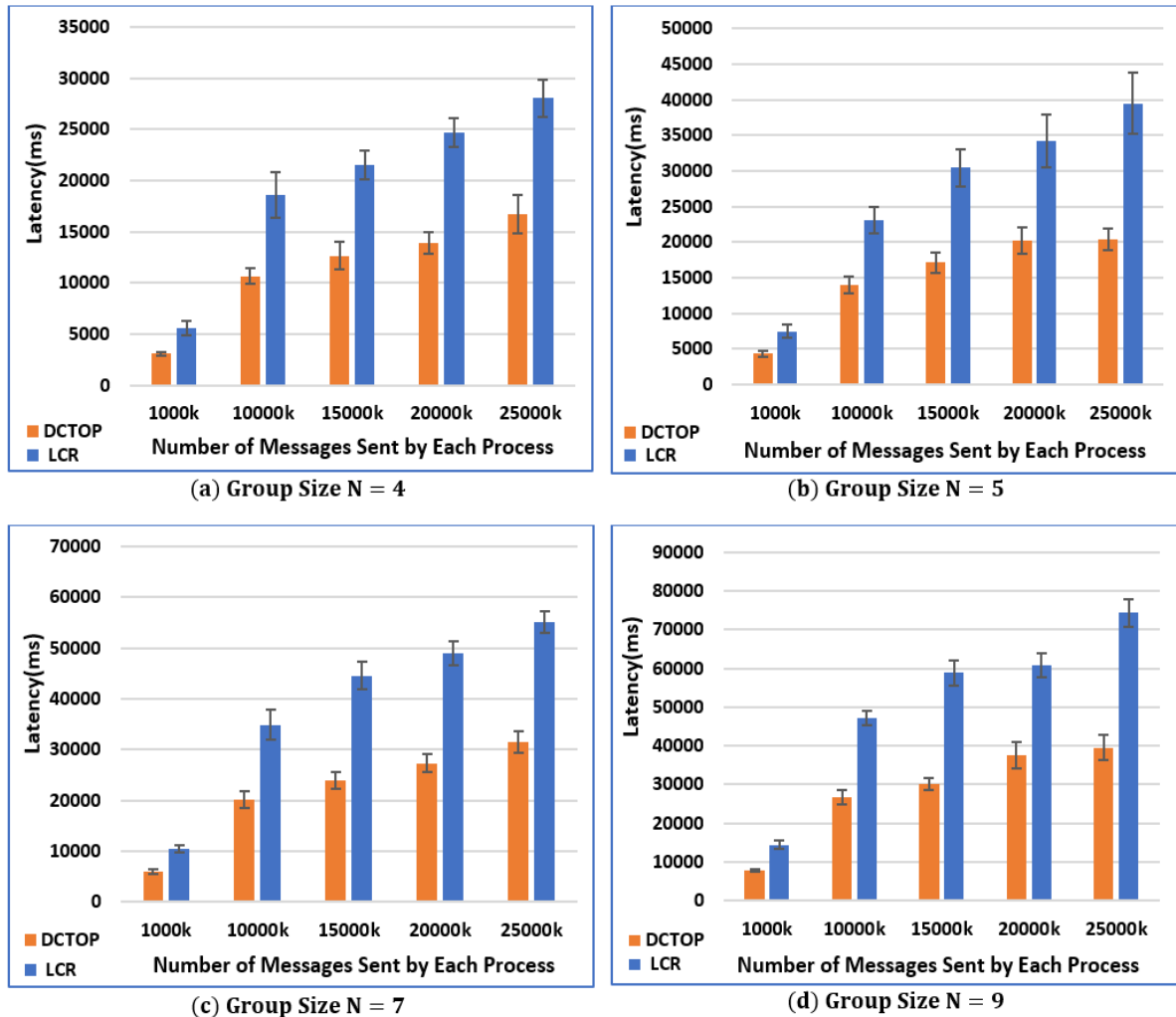


Figure 4.3 Latency Comparison with Fairness Primitive

When $N = 4$, DCTOP shows an average improvement in latency of 42.6%. This improvement can be due to the efficient sequencing of concurrent messages using Lamport logical clocks for message timestamping, incorporating a distinct last process for each sender and a relaxation of the crash failure assumption which allows message delivery to happen quickly thereby reducing latency. Additionally, DCTOP and LCR display a comparable latency pattern when group sizes $N = 5, 7$, and 9 are considered, although DCTOP consistently outperforms LCR regarding latency. In particular, the latency improvement is 42.9% when $N = 5$, 43.6% when $N = 7$, and 44.5% when $N = 9$.

Throughput

Figure 4.4 shows the throughput results of our fairness-controlled environment simulation for all N and the varying number of messages transmitted during the simulation. DCTOP and LCR have almost similar throughput across all message counts transmitted during the simulation, as shown in Figure 4.4a. This indicates that the throughput capabilities of DCTOP and LCR are comparable. This observation proves that ring-based protocols, such as LCR, have been shown to provide the highest attainable throughput.

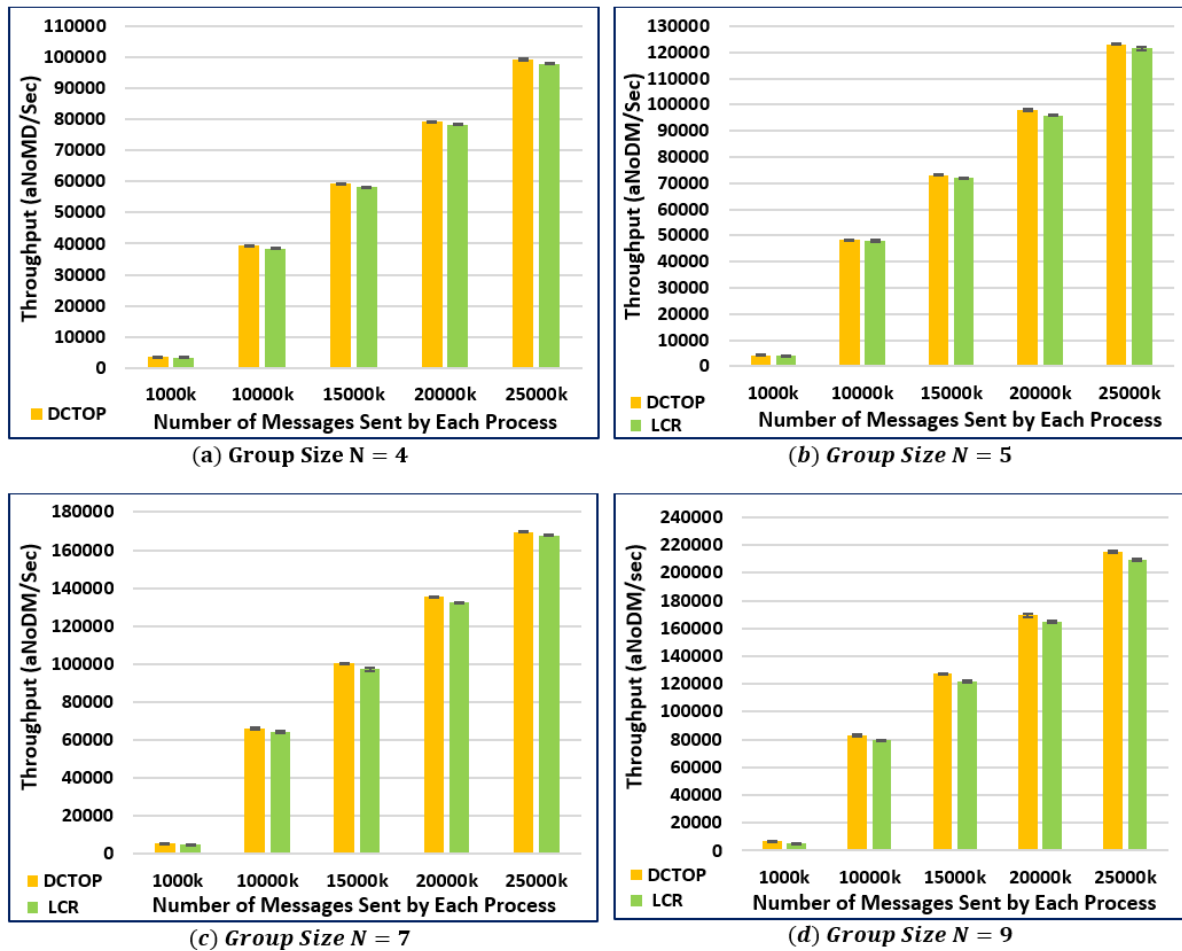


Figure 4.4 Throughput Similarities with Fairness Primitive

Interestingly, the two methods also yielded approximately equal average system throughput, as illustrated in Figure 4.4b, Figure 4.4c, and Figure 4.4d, with DCTOP having a relative improvement of 6.6% or less over LCR. This suggests that the relationship between the two protocols remains consistent as the system scales. However, the average maximum throughput values varied for each group size.

Overall, the results indicate that ring-based leaderless order protocols, DCTOP and LCR, perform similarly in terms of throughput, regardless of the number of messages transmitted during the simulation time and the different approaches employed in such ring-based protocols. This implies that both protocols execute concurrent message transmission efficiently and can oversee the additional workloads with negligible change in throughput.

4.3.2 Fairness vs Without Fairness Control Primitives

In this section, we presented latency and throughput in the form of performance improvements in a simplified manner. Table 4-1(a) and Table 4-1(b) show latency and throughput improvements in the absence of a fairness control primitive. In general, it is worth noting that the performance of DCTOP outperforms that of LCR for all the number of transmitted messages, as shown in Table 4-1(a). DCTOP offers a constant latency improvement for all the group sizes considered in the simulation, although the improvement decreases as the group size increases while the system reaches a steady state. This can result from the increased number of process replicas. As the group size increases, the number of hops a message must make also increases, thereby increasing latency which declines latency performance. However, there is no significant throughput improvement, as observed in Table 4-1b. This implies that DCTOP and LCR offer equivalent throughputs for all the transmitted messages regardless of the differences in latency improvements.

Table 4-1 Performance improvement of DCTOP over LCR in the absence of fairness control primitive

N #Messages	4	5	7	9
1000k	11%	7%	5%	7%
10000k	11%	7%	5%	8%
15000k	11%	7%	6%	8%
20000k	11%	6%	6%	8%
25000k	11%	7%	6%	8%

(a) Latency Improvement

N #Messages	4	5	7	9
1000k	1%	0%	1%	1%
10000k	0%	0%	0%	2%
15000k	0%	0%	0%	0%
20000k	0%	0%	0%	1%
25000k	0%	3%	0%	0%

(a) Throughput Improvement

Table 4-2 Performance improvement of DCTOP over LCR in the presence of fairness control primitive

N #Messages	4	5	7	9
1000k	45%	42%	42%	45%
10000k	42%	39%	42%	43%
15000k	41%	44%	46%	49%
20000k	44%	41%	44%	38%
25000k	41%	48%	43%	47%

(a) Latency Improvement

N #Messages	4	5	7	9
1000k	9%	4%	16%	18%
10000k	3%	1%	3%	4%
15000k	2%	2%	3%	5%
20000k	1%	2%	2%	3%
25000k	1%	1%	1%	3%

(a) Throughput Improvement

Table 4-2(a) and Table 4-2(b) present latency and throughput improvements in the presence of a fairness control primitive. Here, we observed a significant latency improvement for all the groups and the number of messages transmitted, as shown in Table 4-2a. For instance, when $N=4$, the latency improvements were between 41% to 45% as against 11% latency improvement in the absence of fairness; when $N=5$, the latency improvements were between 39% to 48% as against 7% latency improvement in the absence of fairness; when $N=7$, the latency improvements were between 42% to 46% as against 6% latency improvement in the absence of fairness; when $N=9$, the latency improvements were between 38% to 49% as against 8% latency improvement in the absence of fairness. In addition, DCTOP shows constant and consistent latency improvements as the system reaches a steady state for all N . Furthermore, there is a slight significant throughput improvement, as observed in Table 4-2b, compared to that obtained in the absence of fairness primitive. This significant latency and slight throughput improvement observed in DCTOP compared to LCR can be attributed to the supplementary approach of migrating a stable message from $mBuffer_i$ into the delivery queue (DQ_i) in DCTOP. The immediate migration of stable messages to a dedicated delivery queue in DCTOP ensures that these messages are delivered as soon as they are at the head of DQ_i , minimizing delivery delays. In contrast, LCR's approach of using a single pending list can lead

to stable messages being delayed unnecessarily if preceding messages in the list are still unstable. On the other hand, while the throughput improvement in DCTOP is slight, it is primarily due to the optimized handling of stable messages. However, this improvement is not as pronounced as the reduction in latency because the primary bottleneck for throughput lies in network and processing speeds, which both protocols must contend with. Overall, DCTOP's design enhances message delivery efficiency by reducing latency through a more structured approach to handling message stability, even though the throughput gains are relatively modest.

4.3.3 Summary

The inclusion of fairness control primitive into our simulation for both protocols resulted in significant performance improvement in terms of latency and a slight improvement in throughput. The latency in DCTOP consistently did better than the LCR latency for all the messages transmitted using variable numbers of N . This consistent performance improvement consolidates the claim that DCTOP is a viable alternative to LCR, and this can be the result of using the Lamport logical clock for message timestamping, the dynamically determined last process mechanism for ordering concurrent messages, which is an efficient alternative to a globally fixed last process as evidenced in the LCR protocol, and the relaxation of the crash failure assumption. Interestingly, we observe that DCTOP has a relative throughput improvement of 6.6% or less over LCR. This implies that in terms of throughput, the relationship between the two approaches remains stable as the group size grows. However, the average maximum throughput's specific values may differ depending on the group size.

When running both LCR and DCTOP protocols using a greedy sending approach, the results show no considerable improvement in throughput (about 0%). However, there is a relative latency improvement, with DCTOP improving latency by about 5% to 11%. It is important to note that, while the fairness control primitives aim to provide each process in the group an equal opportunity to transmit messages, they adversely affect throughput and increase latencies in both protocols. Notably, DCTOP performs significantly better in this regard. However, the trade-off between fairness and performance must be carefully considered when choosing between these approaches.

While the provided discrete event simulation for DCTOP offers interesting improvements over LCR, it comes with some limitations that should be acknowledged: (i) The simulation assumes that events occur independently, and the future state depends only on the current state (Markov property). This simplifies the modelling but does not capture the complex dependencies and correlations present in real-world systems. (ii) The current implementation does not model any

process or communication failures. In real distributed systems, processes can crash, and network partitions can occur. Handling such failures is crucial for a robust total order protocol.

(iii) The simulation uses a small number of processes (e.g., $N=4, 5, 7$, and 9). While small-scale distributed systems might seem to work seamlessly, real-world systems with a much larger number of processes often uncover issues related to scalability, performance, and robustness that were not evident in smaller setups. It is crucial to design and test systems with these considerations in mind.

Chapter 5

Tackling Leader Load in Raft Protocol

In Raft protocol, the leader is the process replica tasked with managing client requests, making decisions, and overseeing the replication of log entries throughout the cluster. Within a Raft cluster, a single process replica serves as the leader, while the others function as followers. The leader takes charge of processing client requests and orchestrating the replication of logs across the entire cluster. When the leader encounters a failure or becomes unreachable, a new leader is elected through a leader election procedure. This leader election procedure guarantees that only one leader remains active at any given time, even in scenarios involving failures or network partitions (see Section 2.7.2).

This chapter investigates different approaches for tackling the problems of leader load in the Raft protocol as the arrival rate increases. We seek to minimise the number of acknowledgements a leader receives from followers before committing the sequence number attached to a message in its log by providing various ways of organizing the followers in the Raft cluster of environments.

The remainder of this chapter is structured as follows: First, we present the Raft performance bottleneck and the rationale behind our design approach for Raft-variants. This is followed by a description of the design objectives and the possible assumptions based on the proposed protocols. We designed two Raft-variants: (i) chain Raft (RaftCh) and (ii) balanced fork Raft (RaftBf) by considering group sizes of $N = 3$ and $N = 5$. When $N = 3$, only RaftCh will apply. When $N > 3$ (ideally, $N = 5$), we have many ways to organize the followers: we explored all possible organisations of the followers that will maintain the already existing strong consistency model associated with the Raft protocol. This led us to structure the followers in this study into a chain and a balanced fork, without violating the strong consistency model of the Raft protocol. Finally, this will be followed by the restrictive fault-tolerance assumptions and the developments of the proposed Raft variants.

5.1 Motivation - Raft Performance Bottleneck

We chose the Raft order protocol in this research because of its popularity and ease of practical implementation with several open-source implementations [147]. Its safety properties have been formally specified and proven, and its efficiency is comparable to other leader-based order protocols. It is worth noting that the Raft cluster consists of process replicas which can have different states- leader, candidate, or followers. All the processes initially start as followers and when one of them is elected by receiving a majority of votes, it becomes a leader. When the leader crashes, the first follower to detect the leader's crash changes its state to a candidate, votes for itself and requests votes from other followers. The candidate becomes the new leader when it receives the majority of the votes (see Section 2.7.2). Hence, throughout the remainder of this chapter, we will denote a leader as "L" and a follower as "F". Therefore, Figure 5.1 depicts a standard logical message dissemination of the Raft cluster environment with $N = 5$ and $N = 3$ processes, respectively. In the case of the $N = 3$ cluster, there is one leader (L) and two follower nodes (F_1 and F_2). For the $N = 5$ cluster, there are four follower nodes (F_1 , F_2 , F_3 , and F_4) and a leader, L. As shown in Figure 5.1, every client request is routed through the leader, and the leader assigns the sequence number for client requests so they can be processed in the (identical) order of those sequence numbers. Thus, client requests arrive at the leader queue and are stored on a first-come, first-serve (FIFO) basis. Hence, the leader picks the message at the head of the queue and broadcasts it to the followers. The leader requires a majority of acknowledgements from the followers to commit a client request to the sequence number it assigned it. Once the sequence number is committed, then, only then, the leader and other followers execute the request in that order before the leader sends a response back to the client.

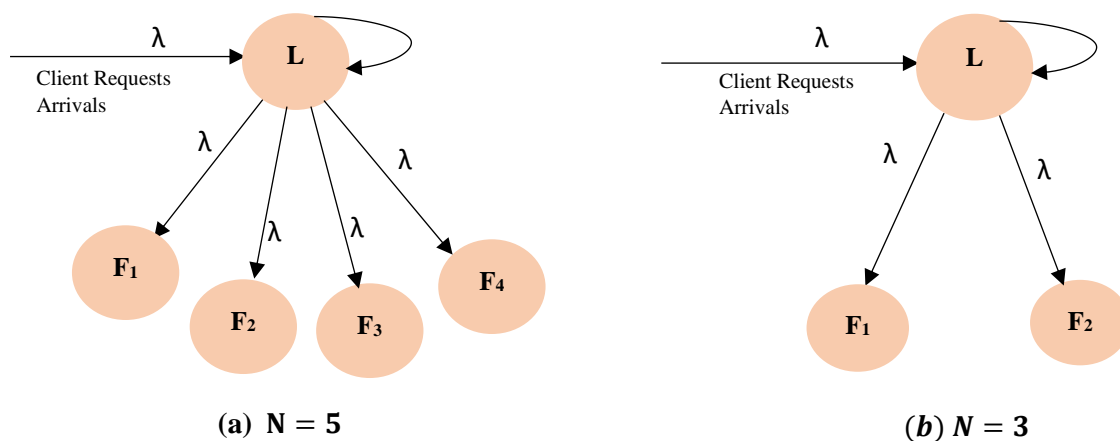


Figure 5.1 Conventional Logical Message Dissemination of Raft

So, for a cluster of 3 processes, as depicted in Figure 5.1b, the leader requires a maximum of 2 acknowledgements - a majority of 3, one from itself and one from any of the two followers. This validates Raft crash tolerant as 1 follower can crash, and the system will survive the crash. However, the leader must receive all incoming acknowledgements and process half of them. If λ is the message arrival rate, the leader requires a maximum acknowledgement rate of 2λ before committing the sequence number it assigned a client request. Similarly, for a cluster of 5 nodes illustrated in Figure 5.1a, the leader requires a maximum of 3 acknowledgements – a quorum of 5, one from itself and two from any of the four followers. However, the leader receives all incoming acknowledgements, processing half of them. Thus, it requires an acknowledgement rate of 4λ before committing a sequence number of a client request. Hence, the leader works extremely hard when the leader receives increased client requests.

Consequently, we ran a simple experiment to show that the Raft protocol completely saturates at some point under increased load at the leader, highlighting our motivation to look for an alternative mechanism. In the experiment, we used a cluster of 5 processes and variable message arrival rate, λ , which ranges from 25, 50, 100, 200 up to 1300 messages per second. We assumed the message processing delay to be 3ms and message transmission delays to be 30ms. We anticipate that message processing should occur rapidly, thus influencing our decision to choose a small value. In the literature [148], it is suggested that for optimal performance, an ideal latency range is between 30ms and 40ms. This consideration informs our choice of 30ms for message transmission delay. The performance metrics used in our simulation are latency and throughput.

Latency: Latency is the time difference between when a client request arrives at the leader and when the client receives a response. If t_a is the arrival time for a client request, after ordering the request, processes execute the request using the processing delay of 3ms as stated earlier. Then L returns the response as shown in Figure 5.2. If t_r is the response time for the client request, then, $t_r - t_a$ is the latency for that client request.

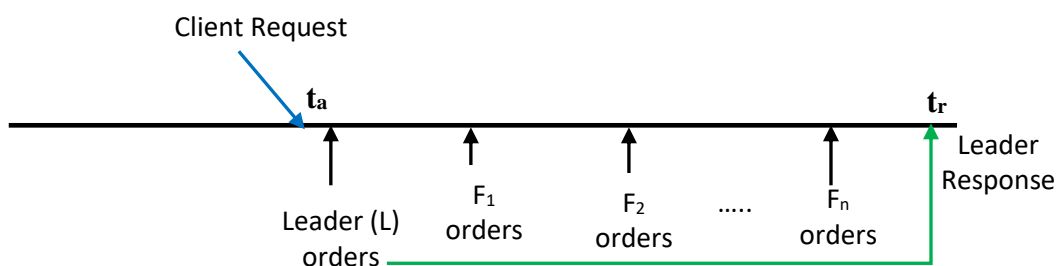


Figure 5.2 Raft Response Time Estimation

Thus, we define response time as the amount of time required for a system (cluster environment) to respond to a client request or complete a client request. This encompasses the entire time interval from when a client initiates a request to when the system responds.

Throughput: We computed the throughput as the average number of messages delivered (aNoMD) to the clients within the simulation time. We ignored the first 100 leader response times for a more stable response time. Suppose t_{101} is the time it took for the 101st response to be delivered to the 101st client and t_{last} is the time it took for the leader to send a response to the last client before the simulation terminates. Then throughput is calculated as:

$$Throughput = \frac{Number\ of\ Responses}{t_{last} - t_{101}}$$

The experiments were repeated 10 times for a 95% confidence interval for latency and throughput. The simulated point represents the result of a simulation run where ten million messages pass through the system. Intuitively, we expect the leaders' throughput to increase as the arrival rate increases.

However, the throughput will remain relatively constant at some point despite the increased arrival rate. Subsequently, the throughput may degrade as the arrival rate increases, indicating that the leaders' capacity must have been exhausted. Similarly, we expect the latency will rise gradually until it reaches a specific point, after which it remains relatively constant. As a result, as long as the arrival rate keeps rising, the latency will eventually plateau. Indeed, that's what we observed, as shown in Figure 5.3 and Figure 5.4.

Thus, for these parameters used in the simulation, the system displays a sign of saturation as $\lambda > 200$ while it completely reaches saturation point as $\lambda > 700$ and then becomes unstable when $\lambda > 1000$.

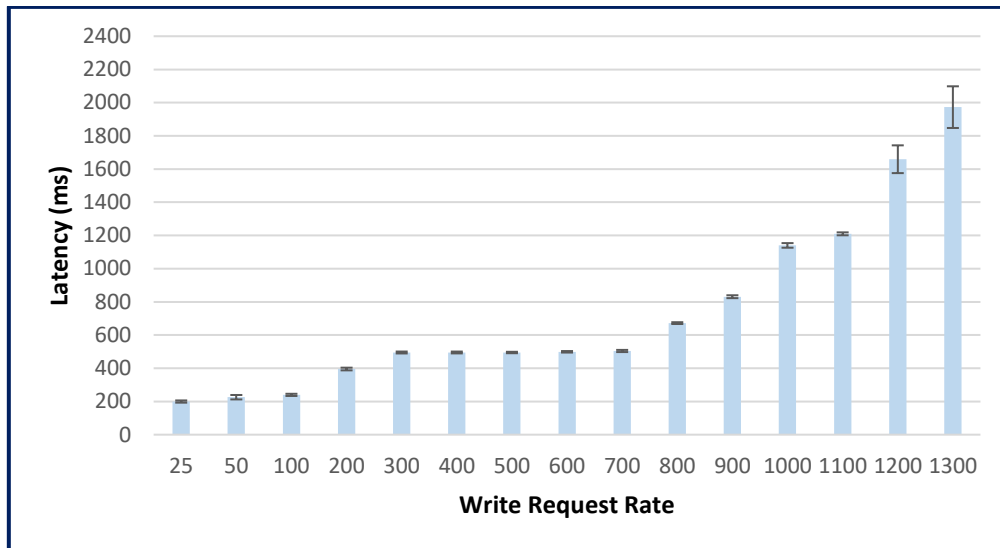


Figure 5.3 Raft Latency vs. Arrival Rate

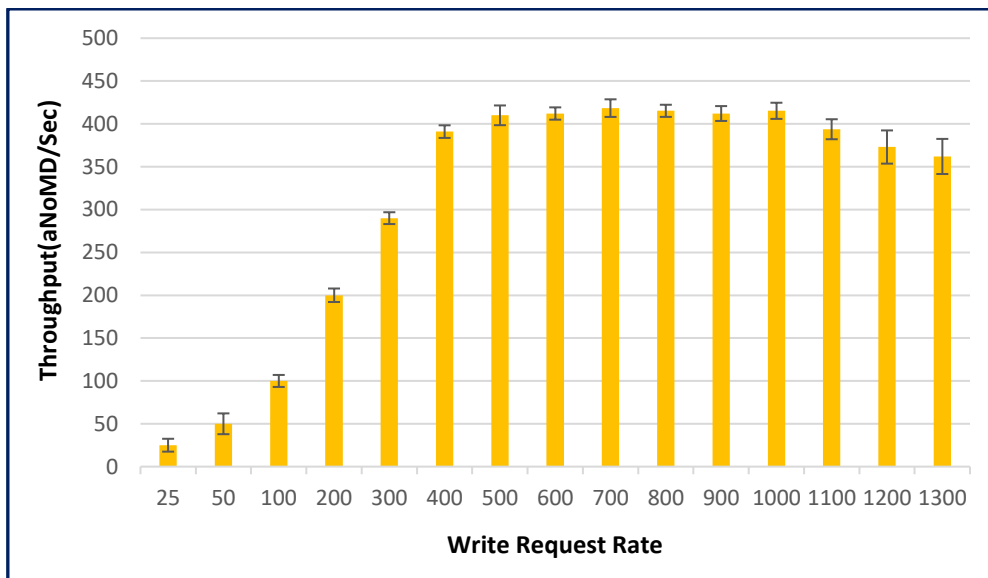


Figure 5.4 Raft Throughput vs. Arrival Rate

Moreover, in the Raft protocol, the bottlenecks at the leader are unavoidable, though we argued in this study that the communication steps and high load on the leader could be reduced while ensuring the protocols' safety, liveness characteristics and correctness are maintained. Active replication would have provided the most effective substitute for the Raft protocol's logical message dissemination mechanism in balancing the load and latency at the leader.

However, the fundamental idea of active replication is that the processes within the cluster behave independently, handling and responding to each client request. The disadvantage of this strategy is that it inevitably requires significant resource consumption and is likely to cause bottlenecks, especially in situations of high load and potentially violate the design idea of the

leader-based protocol, in this case, Raft, where only the leader executes client requests and coordinates client request replication to followers. Therefore, it might be simpler and more beneficial to examine how to tackle the leader load in the Raft protocol to improve the overall system performance while still maintaining the integrity of the Raft protocol using a novel approach.

Therefore, we design Raft variants, new leader-based total order protocols, with N , $N > 2$, where N is the number of processes within the cluster. We modified the traditional Raft logical dissemination framework into (i) Chain Raft (RaftCh) and Balanced Fork Raft (RaftBf) variations, and (ii) we use a novel idea that permits the leader to commit the sequence number attached to the client message when it receives a single explicit acknowledgement from the last follower from either the chain or fork. The outcome is that the last follower on either the chain or fork sending a single explicit acknowledgement for sequence number commitment at the leader is unique, unlike the quorum of explicit acknowledgements required for the same task in the conventional Raft. We achieved our goal using these three approaches:

Firstly, in a RaftCh, we assumed that messages are passed from the leader to the followers in a uni-direction. If a leader receives a client request, it sequences the request with a unique number that enforces FIFO and then passes it as a message, m to the follower next to it. If the follower following the leader is not the last follower on the chain, it forwards a copy of the message to the next follower following itself. This continues until the last follower on the chain receives the message. When the last follower receives the message, it generates an acknowledgement, $ack(m)$ and sends it to the leader. Upon the reception of the $ack(m)$, the leader commits the client message sequence number, executes the message via its state machine. Upon ascertaining that a log entry has been committed by receiving a commit message (Append Entries include commit index) announcement from the leader, the followers proceed to apply the entry to its local state machine for execution, adhering to the order within the log. Then both the leader and the followers execute the client request in the committed order and the leader replies to the clients with the results of the message execution.

Secondly, in a RaftBf, the leader broadcasts the message, m to the successive followers on either fork. When each follower on either fork receives the message, if they are not the last follower on either fork, each independently forwards a copy of the message to the next follower following them, respectively. This continues until the last followers on either fork receive the message, after which each generates an $ack(m)$ separately and sends it to the leader. The leader only needs a *single* $ack(m)$ from either of the last followers on the fork to commit the sequence

number assigned on the client request. When both $\text{ack}(m)$ arrive simultaneously at the leader, the leader only receives and executes one $\text{ack}(m)$ and ignores the other.

Thirdly, we upheld all other necessary assumptions in the traditional Raft required for *leader elections*, *membership changes*, and crash-tolerant assumptions in which $2f + 1$ processes are required for the system to be operational, where f is the number of accommodated crashes expressed as $f = \left\lfloor \frac{N-1}{2} \right\rfloor$. Leader election provides the guidelines for electing a new leader in cases where a leader crashes and membership changes define the operation for adding, removing crashed processes or replacing processes as documented in Raft [3].

5.2 Design Objectives

We aim to tackle the leader load during a high workload by proposing two Raft protocol variations. By investigating these various Raft variations, we intend to reduce the leader's load during periods of high workload and enhance the protocol's general performance and scalability. Simultaneously, the section emphasises retaining the Raft protocol's easily understandable and simply implementable assumptions for leader election and membership changes. We accomplish our purpose in two ways.

First, we consider a collection of specific fault-tolerant assumptions constraining the system: Processes can crash independently. This means a process crash does not immediately force another process to crash. Also, regardless of process crashes, the system guarantees that at least $\left\lceil \frac{N+1}{2} \right\rceil$ processes are always operational and connected. This mechanism ensures that the system continues to function effectively despite the crash.

Secondly, we implement an innovative technique in our design by enabling only a few selected follower(s) to transmit an acknowledgement to the leader. The resulting modification attempts to reduce the leader's time spent receiving and processing the number of acknowledgements from followers. This is in contrast to leader of the traditional Raft protocol which receive all acknowledgements and processes half of them. In our new design, we use a principle known as "balanced fork" and "chain" to organise the followers in the new Raft variation. This idea guarantees an efficient follower configuration, improving the system's effectiveness and performance.

It is crucial to emphasize that the newly proposed protocols presented here differ from Raft solely in the logical message dissemination framework and the required number of acknowledgements for committing a log entry. Importantly, these protocols have been shown

to uphold all the invariants essential for maintaining the crash-recovery phases of Raft without alteration. Consequently, implementing them using existing Raft implementations is a straightforward process.

5.3 Assumptions

This section defines the assumptions made when designing the proposed Raft variants. The traditional Raft assumption was retained except for the modification in A1.4 and A1.5.

A1.1 - Leader Crashes and Recovery

When the leader crashes and recovers subsequently, it does not attempt to join the system immediately until the operative processes have elected a new leader. When the old leader recovers from the crash, it joins the system as a follower. Leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all the entries in its log). The new leader handles inconsistencies by forcing the followers to duplicate its own log since its log is the most up-to-date log. This means conflicting entries in follower logs will be overwritten with entries from the leader's log.

Figure 5.5 shows some leader election scenarios that might occur as the leader crashes. Suppose we have an $N = 5$ cluster, L is the leader, F_1 , F_2 , F_3 , and F_4 are followers; all the processes are operational in the Raft protocol in term 1. In term 2, following the leader crash, any of the followers can become the leader. Suppose F_2 becomes the first to notice the leader crash (see Section 2.7.2); it changes into a candidate, votes for itself, and sends a request vote remote procedure call (RPC) to other followers. F_2 becomes the new leader if it receives a majority of the votes from the followers. In term 3, suppose that F_2 crashes and the old leader remains crashed; then the following scenarios can arise:

1. $\{F_1, F_3, F_4\}$ can form the quorum and supposes all the followers time out simultaneously, increase their term, become candidates, vote for themselves and request votes from each other. No candidate gets majority votes (because votes are split); hence, the election fails (deadlock occurs meaning no way out). When this happens, each candidate will time out and begin a new election after increasing its term and starting a new cycle of request vote RPCs.
2. Any of $\{F_1, F_3, F_4\}$ can become the leader in the new term; if any of them timeout first, vote for themselves, send a request vote RPC, and receive a majority of the votes.

3. L revives and joins the cluster $\{L, F_1, F_3, F_4\}$. While it is feasible to successfully elect a leader since the cluster already possesses a quorum of followers $\{F_1, F_3, F_4\}$, L is unable to participate in the election until a new leader has been successfully elected. In this scenario, L can only join the cluster as a follower and will inevitably receive the duplicate log from the current leader, as its log is the most recent or up-to-date.

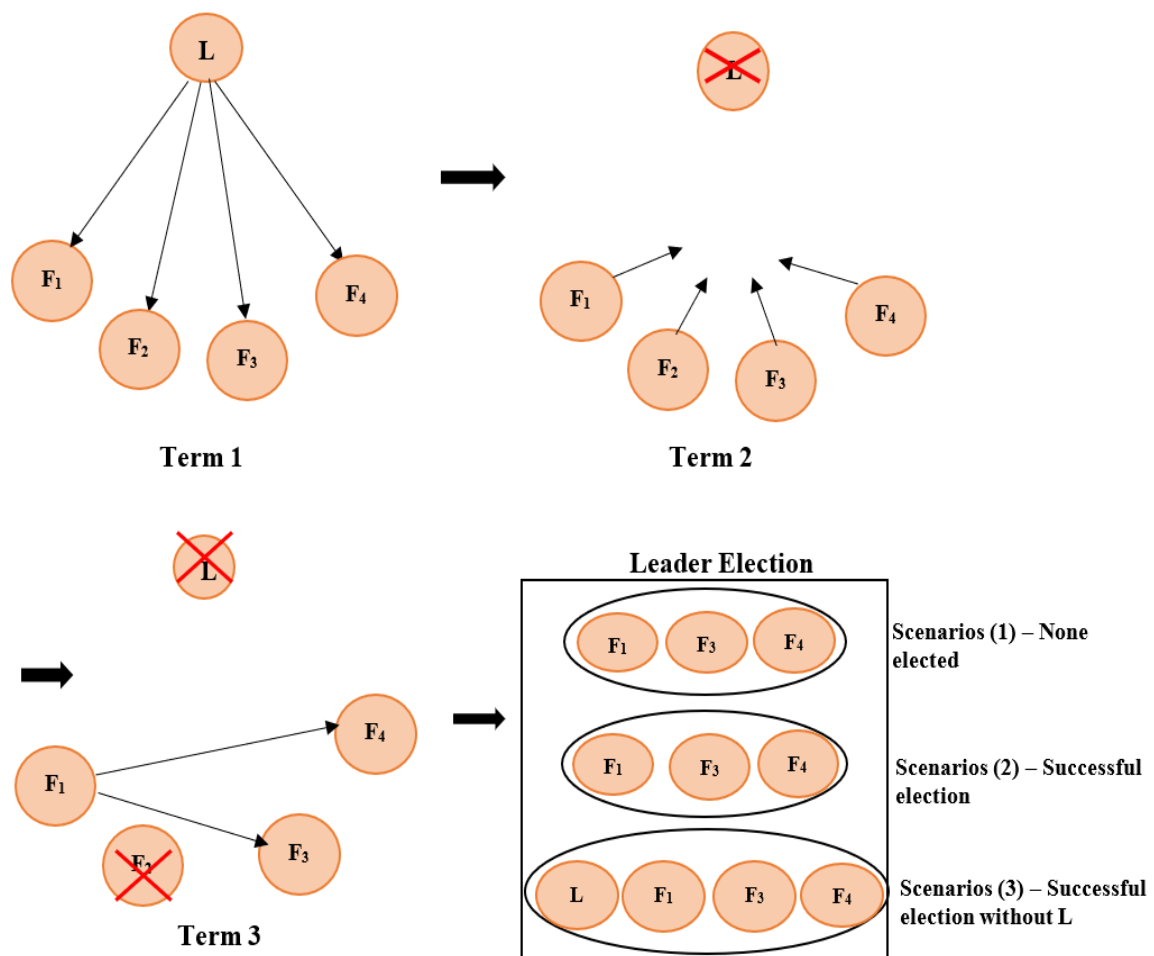


Figure 5.5 Leader Election Scenarios

It is important to highlight that Scenarios (2) and (3) lead to a successful leader election due to the leader election requirements. These requirements specify that a process can elect a new leader if it possesses the latest log entries (states) from the previous terms.

A1.2 - Process Crashes

When precisely $\left\lceil \frac{N+1}{2} \right\rceil$ processes are executing the Raft protocol; no process may fail. As a result, a majority is always in place, enabling the election of a new leader in the event of a leader crash and the continuation of committing log, executing it via the local state machine and sending clients' responses in the event of a follower crash. A process not being honest can affect the security of the system. Hence, we excluded the Byzantine failure assumption so that processes cannot return malicious or dishonest values during message communication and transmission; instead, we assumed a fail-stop where a process fails and stops communicating with other processes. In this way, operative processes can only forward honest messages to other processes, while processes returning dishonest messages never occur.

A1.3 - Follower and Candidate Crashes

Leader crashes are far more challenging to manage than follower and candidate crashes, which are managed similarly. Future RequestVote and AppendEntries RPCs sent to a follower or candidate will fail if it crashes (or if the network connection between the follower and the leader fails). Raft resolves these problems by allowing the leader to retry the sending of RPCs indefinitely to the crash process; if the crashed process restarts, the RPC will conclude successfully.

A1.4 - Number of Acknowledgement

In the traditional Raft protocol, the leader does not commit client requests until it has received all acknowledgements from the followers and processed half of the followers' acknowledgements including self to get a majority of N . In Raft, followers make independent, explicit acknowledgements to the leader. However, the modification we suggested in our approach is that many (not all) followers make implicit acknowledgements and only a few make explicit acknowledgements to the leader. Thus, the leader only needs a single explicit acknowledgement from either the last followers in the RaftBf structure or the last follower in the RaftCh structure. This modified condition enables the leader to speed up the process of committing client requests while still maintaining the essential degree of total order and fault tolerance. Our solution attempts to improve the overall effectiveness and responsiveness of the leader in the Raft protocol by minimising the necessary number of acknowledgements required to commit a request.

A1.5 – Replication Structure

Raft uses a star-like topology where the leader replicates client requests by broadcasting them in the form of AppendEntries RPC to all followers. Chain Raft uses a chain topology where the client requests are sequentially replicated down the chain from the leader node to the last follower. Balanced fork Raft uses a fork/tree-like topology where the leader replicates log entries to equal sub-followers in the forks in parallel until it reaches the last followers on either fork

A1.6 - Reliable Server Communication

Raft process replicas communicate reliably using remote procedure calls (RPC). Raft protocol uses RPC to create a strong and dependable communication mechanism for scheduling leader elections (RequestVote RPCs), replicating log entries (AppendEntries RPCs), and guaranteeing the consistency and availability of the system. For instance, when a leader sends a message m to all followers, m is eventually received by all operative followers. Messages are also received in the order that they are transmitted; thus, if a leader process sends m_1 followed by m_2 , all operative followers will receive m_1 before m_2 .

5.4 Design Approach

5.4.1 Single Explicit Acknowledgement

The conventional Raft requires that all followers send their respective explicit acknowledgements, $ack(m)$ in response to the reception of the message (m) broadcast from the leader. This procedure ensures that the leader must receive all the $ack(m)$ from followers, and processes half of them before the leader knows that m is correctly replicated across all the followers. Nevertheless, the leader's task of receiving all acknowledgements from all the followers and processing half of them introduces additional load overhead at the leader process, potentially impacting the overall efficiency of the communication procedure of the Raft operations.

Consequently, in a bid to mitigate this bottleneck identified at the leader replica, our design approach ensures that only the last follower on the proposed framework is required to send a single $ack(m)$ to the leader before the leader knows that the message m is properly replicated across all cluster processes. This means that the last follower sending a single acknowledgement to the leader is similar to *cumulative acknowledgement* [149] found in the TCP protocol. Cumulative acknowledgement involves the receiver process acknowledging the correct reception of a message by using a later acknowledgement. This, in turn, will implicitly

conveys to the sender (leader) that the preceding messages it sends were received correctly by all processes within the cluster.

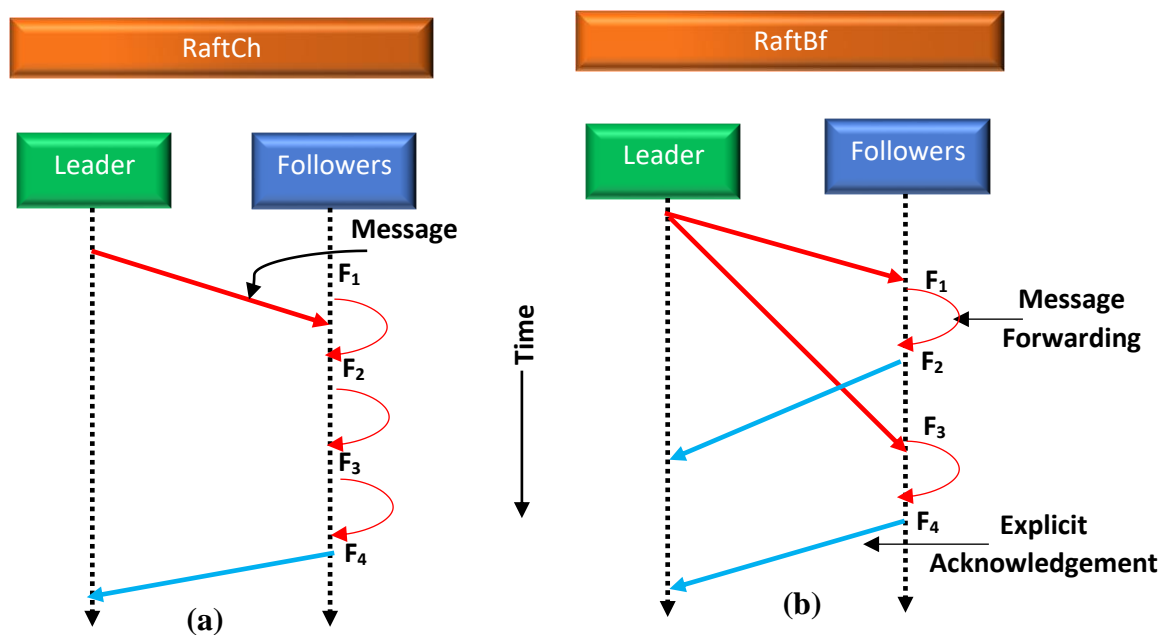


Figure 5.6 Explicit and Implicit Acknowledgements

Replicating cumulative concept in Raft, for a cluster of $N = 4$ processes, when a leader receives explicit $\text{ack}(m)$ from process F_{N-1} , the leader understands implicitly that the message m is correctly replicated in process F_{N-2} and process F_{N-3} , respectively. Figure 5.6a shows the single explicit $\text{ack}(m)$ from the last follower, F_4 to the leader for a cluster of $N = 5$ processes which acts as a cumulative acknowledgement of all the followers for the message broadcast from the leader. Similarly, for the RaftBf (Figure 5.6b), the leader receives 2 explicit $\text{acks}(m)$ and is required by design to process only one of them. It is important to note from the diagram (Figure 5.6) that the red arrow shows the message broadcast from the leader to the followers while the blue arrow shows the acknowledgement from the last follower to the leader.

Additionally, in our proposed Raft variants protocol, apart from the last follower, every other follower does not transmit $\text{ack}(m)$ for every message m it receives from the leader, and such omission was chosen to reduce the inbound traffic at the leader as shown in Figure 5.6. When the leader receives $\text{ack}(m)$ from the last follower within the proposed framework, it proceeds to commit the sequence number attached to the request. The leader then executes the request using its local state machine after which it sends a commit message to the followers. When the followers receive the commit message, they execute the client request in question using their own local state machine to have the same results as the leader. The leader initiates the TO

delivery(m) process to transmit the execution outcome of the request to the client. However, at present, followers in LogCabin (Traditional Raft Implementation platform) do not handle read-only requests but only the leader replica does.

The efficient operation and evaluation of RaftCh and RaftBf are noticeable when client requests arrive rapidly and consistently. Specifically, in scenarios characterized by a high load where requests are received swiftly and frequently, the use of a single acknowledgement is anticipated to be effective. This approach, single ack(m), has the potential to reduce inbound traffic within the leader process, contributing to the overall efficiency of the system under conditions of intense request arrival rates.

5.4.2 Commit Messages

The leader sends commit messages to the followers (each AppenEntries includes the commit index, which communicates to the followers the updated client entries the leader has committed), enabling them to commit the same client request, committed by the leader, within their respective local state machines for execution. Upon confirmation that a log entry is committed after receiving commit messages from the leader, a follower applies the entry to its local state machine, following the order of the log. It is important to note that the leader's responsibility lies solely in the TO delivery process, ensuring the delivery of the execution outcome of the client request to the client.

5.4.3 Switch to/from Raft

The Raft variants presented in this thesis are tailored for optimal performance when the leader and all n followers are in a correct state, where $n = N - 1$. The design of the Raft variants also incorporates the ability to seamlessly transition to the conventional Raft network framework in the event of a leader or follower crash. Once the crashed process is recovered through Raft's crash-recovery mechanism, the system reverts to its own logical message dissemination framework. Assumption A1.3 is leveraged for the implementation of this functionality.

5.5 Design Difference between Raft and Raft-variants

In this section, we highlight the differences between Raft and Raft-variant's logical message dissemination structure, log replication, the number of acknowledgements, and the use cases, see Table 5-1.

1. Logical Message Dissemination Structure

- a. Raft uses a star-like topology where the leader which serves as the central process communicates to all followers connected to it.
- b. Chain Raft uses a chain topology where the leader process communicates only to the follower next to it, which helps to propagate the leader's communication sequentially until it reaches the last follower on the chain.
- c. Balanced fork Raft uses a fork/tree-like topology where the leader communicates only to the equal sub-followers connected to it, the sub-followers help to forward the leader's communication until it reaches the last followers on either fork.

2. Log Replication

- a. In the Raft, a leader replicates client requests by broadcasting them in the form of AppendEntries RPC to all followers.
- b. In the Chain Raft, client requests are sequentially replicated down the chain from the leader process to the last follower.
- c. In the Balanced fork Raft, the leader replicates log entries to equal sub-followers in the forks in parallel until it reaches the last followers on either fork.

3. Request Acknowledgement

- a. In the Raft, a leader requires a majority/quorum acknowledgements including self to commit the sequence number attached to the client request in its log.
- b. In the Chain Raft, a leader requires a single acknowledgement from the last follower on the chain to commit the sequence number attached to the client request in its log. Apart from the last follower, others acknowledge the client's requests from the leader implicitly. Thus, the last follower sending a single acknowledgement to the leader is similar to the cumulative acknowledgement found in TCP.
- c. In the Balanced fork Raft, a leader requires a single acknowledgement from the last follower on either fork to commit the sequence number attached to the client request in its log. Where two acknowledgements for the same request arrive at the leader process from the last followers on either fork, the leader processes only one and ignores the other.

4. Use Case

- a. Raft focuses on simplicity and understandability with a leader-based approach that ensures strong consistency and fault tolerance.

- b. Chain Raft optimizes throughput and reduces latency by using a sequential, chained replication structure while ensuring strong consistency and fault tolerance.
- c. Balanced fork Raft enhances performance and scalability through a balanced forked replication structure that parallelizes log replication and distributes the load evenly.

Table 5-1 Difference Between Raft and Raft variants

Features	Raft	Chain Raft	Balanced Fork Raft
Replication Structure	Star-like topology	Chain topology	Forked/tree-like topology
Log Replication	AppendEntries RPC to all followers	Sequential propagation down the chain	Parallel propagation across forks
Request Acknowledgement	A leader requires majority acknowledgements including self to commit a client request	A leader requires a single acknowledgement from the last follower on the chain to commit a client request	A leader requires a single acknowledgement from the last follower on either fork to commit a client request
Use Cases	Simplicity, understandability	Throughput optimization, reduced latency	Performance, scalability, balanced load distribution

5.6 Raft-variants Protocol Details

This section discusses the responsibilities and functions of the leader in the proposed Raft variants. They are as follows:

- L1.** Leader picks a message m from the head of its incoming buffer, sequences and transmits m to all followers either as a unicast in RaftCh or as a broadcast in RaftBf including itself.
- L2.** On receiving m with the sequence number attached to it from itself, the leader logs m and then sends an acknowledgement, $\text{ack}(m)$, to itself.
- L3.** On receiving the single $\text{ack}(m)$ from the last follower, the leader commits the sequence number on m , to its local state machine knowing that m has been correctly replicated.
- L4.** The leader's state machine then executes the request according to the order it assigned the client request.
- L5.** Leader then sends a response to the client with the results of the execution of the client's request, TO deliver(m) before TO deliver(m'), $m_{\text{ts}} < m'_{\text{ts}}$.

5.6.1 Protocol 1: RaftCh

Raft variations vary according to the group size. Specifically, for $N = 3$, there is only a single variant, RaftCh, and for $N = 5$, two variants exist—RaftCh and RaftBf. However, this section exclusively delves into the discussion of RaftCh. The depiction in Figure 5.7a showcases RaftCh with $N = 3$, while Figure 5.7b illustrates RaftCh with $N = 5$.

Let $\Pi = \{P_1, P_2, P_3, \dots, P_N\}$, be the set of process replicas for the RaftCh protocol connected in a chain framework with variable cluster sizes. We conceptualized the communication framework as asynchronous, devoid of time constraints on communication delays, and characterized by probabilistic message inter-transmission times. In our design, a leader L can only send a unicast message to the follower following it and receive a single acknowledgement from the last follower on the chain framework.

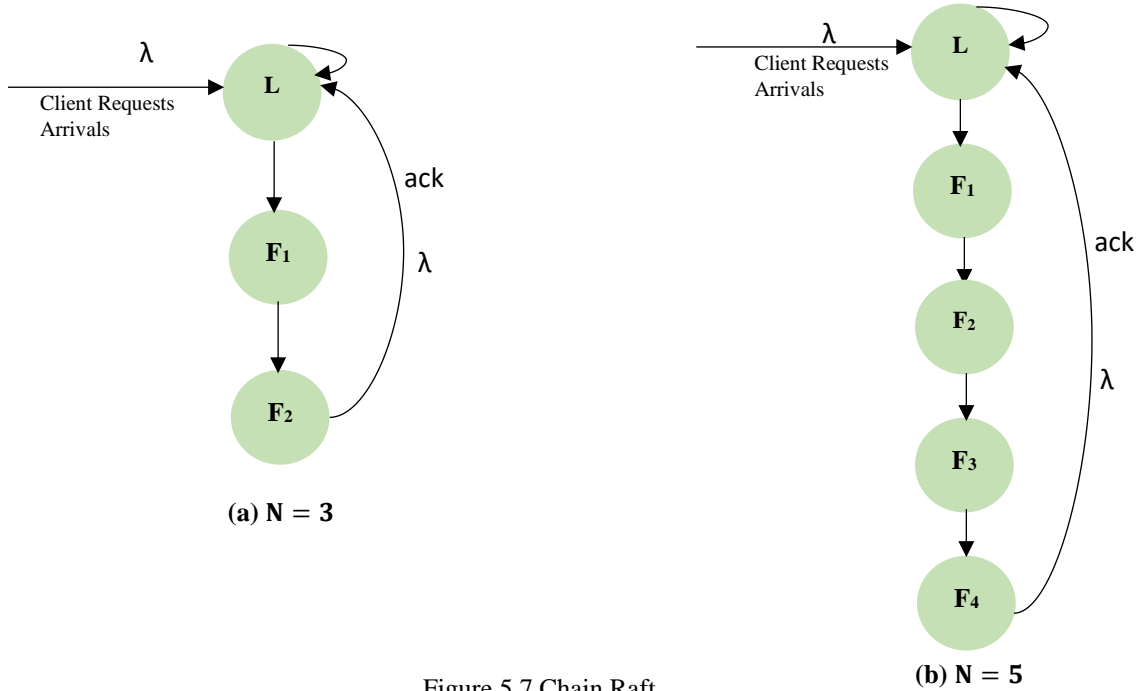


Figure 5.7 Chain Raft

In the RaftCh system, when the leader in chain Raft receives a client request, it picks the message at the head of the queue, sequences it and transmits the request as a unicast to next follower following it on the chain, F_1 (see Figure 5.7). When F_1 receives the message, it forwards a copy of the message to the next follower if it is not the last follower on the chain. This forwarding continues until the last follower on the chain receives the message. When the last follower F_2 (see Figure 5.7a) or F_4 (see Figure 5.7b) receives the message, it knows that it is the last follower on the chain, generates $ack(m)$ and sends it to the leader, L . When the L receives the $ack(m)$, it commits the sequence number attached to the message and sends a

commit message to the followers. The leader then executes the message with followers (after they have performed $\text{commit}(m)$ as a result of the commit message from the leader). The leader sends a response to the clients. Thus, a leader in the RaftCh requires a single λ for receiving $\text{ack}(m)$ from the last follower, whereas, in the conventional Raft, it requires 2λ (for $N = 3$) and 4λ (for $N = 5$) for the same task (see Section 5.1). RaftCh implementation illustrates its effectiveness in settings when client requests arrive quickly and frequently. RaftCh, in particular, demonstrates efficiency and success in decreasing inbound traffic on the leader process under high-load conditions characterised by a quick and frequent request arrival rate.

However, the protocol steps for a follower in RaftCh are as follows:

- F1.** A follower, on receiving the message m from the leader, logs m and forwards a copy of m to the next follower on the chain if it is not the last follower on the chain. This ensures that m is correctly replicated along the chain.
- F2.** When a follower receives a message m during the RaftCh protocol's log replication process, and that follower appears to be the last follower on the chain, the follower stops forwarding m . Instead, it generates an $\text{ack}(m)$ and sends it to the leader. This $\text{ack}(m)$ indicates to the leader that m has been correctly replicated and is now ready for the leader to process it further.
- F3.** On receiving $\text{commit}(m)$ from the leader, the followers commit the sequence number attached to the message to their respective local state machine for execution in the committed order.

5.6.2 Protocol 2: RaftBf

Figure 5.8 illustrates the simple design of RaftBf for an $N = 5$ group size. As shown in the diagram, when the leader receives a client request, it attaches a sequence number to it and broadcasts it as a message to the followers on both forks, F_1 and F_3 . When F_1 and F_3 receive the message, each logs m and forwards a copy to their respective successors, F_2 and F_4 , since they are not the last followers on either fork. However, when F_2 and F_4 receive the message, each logs m and stops forwarding m since each knows that it is the last follower on either fork. Thus, F_2 and F_4 generate $\text{ack}(m)$ and send it to the leader independently. The leader commits the sequence number attached to m when it receives the $\text{ack}(m)$ from either F_2 or F_4 . In the RaftBf design, the leader is required to receive and execute only a single $\text{ack}(m)$ just as required in RaftCh (see Section 5.6.1). However, in a case where the $\text{ack}(m)$ from F_2 and F_4

arrives at the leader simultaneously, the leader only processes one and ignores the other. The leader then executes the request with the followers and replies to the client with the result of the message execution.

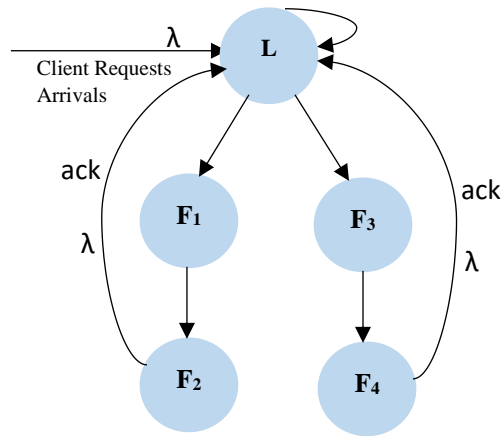


Figure 5.8 Balanced Fork Raft

RaftBf uses this design approach to enhance communication and coordination among the RaftBf cluster, enabling effective message commitment and decreasing network traffic at the leader process while eliminating redundant acknowledgements.

Furthermore, the protocol steps for a follower in RaftBf are discussed below:

- F1.** When followers F_1 and F_3 receive the message m from the leader, each logs m and forwards a copy of m to their respective successor, F_2 and F_4 if they are not the last follower on either of the fork.
- F2.** When F_2 and F_4 receive m from F_1 and F_3 and happen to be the last follower on either fork, each logs m and stops forwarding m . F_2 and F_4 independently generate an $\text{ack}(m)$ and send it to the leader. The leader receives only one $\text{ack}(m)$ and ignores the other before committing m . So, if $F_2 \text{ack}(m)$ arrives first at the leader, it commits the message, and when $F_4 \text{ack}(m)$ arrives later, the leader ignores it.
- F3.** On receiving $\text{commit}(m)$ from the leader, the followers commit the sequence number attached to the message to their respective local state machine for execution in the committed order.

Moreover, in the Raft variants, the leader's log comprises the list of committed messages along with their respective sequence numbers. Therefore, upon receiving $\text{ack}(m)$, if the sequence number matches that of a previously committed message, the leader disregards the $\text{ack}(m)$.

Conversely, if $\text{ack}(m)$ pertains to a message not yet committed by the leader, it proceeds with the message commitment. This mechanism is notable in RaftBf to ensure the use of a single $\text{ack}(m)$ during log replications and the discarding of $\text{ack}(m)$ is crucial in RaftBf, especially since the last followers on the fork sends $\text{ack}(m)$ to the leader independently. Consequently, the leader conducts a check to ascertain whether the received $\text{ack}(m)$ is redundant (due to the associated request already being committed) and subsequently discards it. However, such discarding of $\text{ack}(m)$ is not implemented in RaftCh, as the leader receives only one $\text{ack}(m)$ from the last follower on the chain framework. Hence, RaftCh and RaftBf represent straightforward protocols involving a switch to or from Raft in the event of a process crash, utilizing a single acknowledgement. The message complexity for both protocols is N unicast per client request and TO delivery while Raft requires $2(N - 1)$ unicast per client request for all $N \geq 3$, where N is an odd number.

5.7 Summary

In this chapter, we modified the well-known Raft protocol under its original fault-tolerance assumptions and restricted fault-tolerance assumptions, which are yet practical. Two variants of the Raft protocol have been designed: one uses a chain topology, and the other uses a balanced fork topology. These two Raft variants offer an approach requiring the last follower on the chain or balanced fork to send a single $\text{ack}(m)$ to the leader before the sequence number assigned to messages is committed. This mechanism reduces unnecessary traffic at the leader process. It is important to note that these two Raft variations, notably the chain Raft and balanced fork Raft, are novel and new to the best of our knowledge.

The performance of these Raft variations will be evaluated in the next chapter, and the outcomes will be compared with the Raft itself and those of the DCTOP, a leaderless protocol for the group sizes $N = 3$ and $N = 5$. It's important to stress that our goal is to accurately describe and analyse the results rather than to demonstrate that the Raft variants are superior to DCTOP. Through these evaluations, we hope to learn more about the performance characteristics and potential advantages of various Raft variations compared to DCTOP.

Chapter 6

Performance Evaluation of Raft-variants, and DCTOP

This chapter provides a comprehensive performance evaluation of the key concepts introduced in Chapter 5. The evaluation focuses on seven issues:

- i. Performance comparison of RaftCh and Raft with a group size $N = 3$.
- ii. Performance comparisons among Raft, RaftCh, and RaftBf with a group size $N = 5$.
- iii. Performance comparison of RaftCh and DCTOP. The performance of RaftCh and DCTOP is made with a group size $N = 3$. The evaluation assumed a non-zero processing delay representing the protocols' actual processing circumstances.
- iv. Performance comparison between RaftCh and RaftBf with a group size $N = 5$. We assumed a zero-processing delay since both protocols have identical processing characteristics.
- v. Performance comparison between RaftCh and RaftBf with a group size $N = 5$. Here, we assumed a non-zero processing delay to study the impact of processing delay in both protocols.
- vi. Performance comparison of RaftCh for $N = 3$ and $N = 5$ group sizes including a non-zero message processing delay.
- vii. Performance comparisons among RaftCh, RaftBf and DCTOP with a group size of $N = 5$. The evaluation includes a non-zero processing delay to thoroughly evaluate the protocols' effectiveness in practical situations.

First, we detail a discrete event simulation that simulates Raft, RaftCh for $N = 3, 5$ and RaftBf for $N = 5$. This is followed by the simulation of DCTOP, as discussed in Section 4.2.1, for a group size of $N = 5$. The singular distinction in the simulation of DCTOP in this chapter lies in

the inclusion of message processing delay, whereas in Section 4.2.1, we assumed zero processing delay. This modification guarantees we capture the same processing equivalent as Raft variants since we compare leader-based with a leaderless protocol. Finally, it is worth noting that the simulation we reported in this chapter focused only on studying the performance of these protocols under crash-free settings.

6.1 Simulation

The simulation of Raft variants allows us to assess and compare their performance features and behaviours under various settings. We can learn how these Raft protocol variants operate under different settings and analyse their applicability for diverse distributed system contexts by running simulations. The simulations are designed to offer numerical information on crucial variables, including throughput and latency. These measures let us evaluate the Raft variants' efficiency, scalability, and fault-tolerance potential.

In addition, by using simulations, we tend to examine how various factors, such as group size, message processing delays, and message transmission delays, can affect the effectiveness of these variants. These aid in comprehending how the various Raft variations operate and function under different workload circumstances. Additionally, simulations allow us to assess how well the Raft variations perform compared to other protocols or approaches, such as the Raft and DCTOP.

The comparison of Raft variants with DCTOP is not intended to prove that Raft variants are better than DCTOP. Instead, our emphasis is on reflecting on the results of both protocols to understand their performance trends. Raft variants do not have to be better or worse than DCTOP. However, we hypothesize that the Raft variants cannot perform worse under any given set of circumstances than the conventional Raft protocol, as shown in Section 5.1. This is because the variations from the traditional Raft procedure differ mainly in the number of acknowledgements the leader must receive before committing any client requests and the logical message dissemination. The Raft variations maintain the same leader-based structure and workings as the original Raft protocol, ensuring the same assurances of safety and correctness. To reduce communication overhead and the potential saturation regions of the leader process and increase overall efficiency, the improvements made to the variants generally concentrate on modifying the organizations of the followers and also minimising the number of acknowledgements sent to the leader for committing any client message, in this case, a single acknowledgement, unlike a quorum acknowledgement in the traditional Raft.

Simulating Raft variants offers insightful information about their performance, behaviour, and trade-offs so researchers and professionals may decide whether to adopt and use them in distributed systems.

In assessing the performance of the Raft protocol and its variants, our discrete event simulation [147] will utilise the simulation experiment explained in Section 5.1. However, the evaluation of Raft variants and the DCTOP will use the machine characteristics as described in Section 4.2.1 and follow the subsequent experimental setup.

Furthermore, in discrete event simulations, the difference between Raft, Raft variants and DCTOP can be highlighted by the way events are handled, propagated, and ordered in the system. Here are the explicit differences which are summarised in Table 6-1:

1. Raft and Raft variants

Replication Structure:

- **Central Leader:** A single leader process manages the ordering and replication of messages.
- **Follower Processes:** All other processes follow the leader's instructions for message ordering.

Event Types:

- **Send Event:** The leader sends a message to all followers.
- **Receive Event:** Followers receive messages from the leader.
- **Commit Event:** Once a majority of followers acknowledge receipt, the leader commits the message and instructs followers to commit.

Event Handling:

- **Send Event:** Initiated by the leader replica when a client request/message arrives for execution which requires replication and ordering according to the sequence number attached to the client request by the leader replica.
- **Receive Event:** Followers execute incoming messages from the leader according to the order they receive them.
- **Commit Event:** The leader commits the message after receiving acknowledgements from a quorum of followers including self, then instructs followers to commit.

Message Propagation:

- **Centralized:** Messages are propagated from the leader to all followers.
- **Quorum-Based:** The leader waits for acknowledgements from a majority (quorum) of followers before committing the message.

Ordering:

- **Leader-Determined:** The leader determines the order of messages, ensuring total order.
- **Single Point of Ordering:** All messages go through the leader, which ensures a consistent global order.

Failure Handling:

- **Leader Election:** In case of leader failure, a new leader is elected.
- **Recovery:** The new leader ensures all followers are consistent before resuming normal operations.

2. DCTOP**Replication Structure:**

- **Ring Topology:** Processes are organized in a ring, with each process communicating with its CN.
- **Decentralized Coordination:** There is no central leader; all processes participate equally in message sending and ordering.

Event Types:

- **Send Event:** A process sends a message to its CN in the ring.
- **Receive / Delivery Event:** A process receives a message from its ACN in the ring, and attempts delivery to the application layer if the message is stable and at the head of DQ.

Event Handling:

- **Send Event:** Any process can initiate a send event, starting the propagation of a message around the ring. We assumed concurrent send events for all process replicas.
- **Receive / Delivery Event:** Processes receive messages from their ACN and execute them accordingly. Processes can attempt delivery to the application layer if the message is stable and at the head of DQ.

Message Propagation:

- **Sequential:** Messages are propagated sequentially around the ring, passing from one process to the next until they reach the ACN of message origin.
- **Distributed:** Each process executes messages independently and ensures they follow the correct order before passing them on.

Ordering:

- **Distributed Determination:** Ordering is determined by the message timestamp, which is used to sequence messages as they are passed around the ring.

- **Local Clocks:** Processes use logical clocks for message timestamps to help maintain order, ensuring messages are delivered in the correct sequence.

Failure Handling:

- **Process Recovery:** If a process replica fails, the ring can be reconfigured to bypass the failed replica.
- **Redundancy:** Messages may be retransmitted to ensure they reach all processes despite failures.

Table 6-1 Simulation Differences Between Raft, Raft-variants and DCTOP

Feature	Raft, and Raft variants	DCTOP
Replication Structure	Central leader with follower processes	Decentralized ring topology
Event Types	Send, Receive, Commit	Send, Receive/Delivery
Event Handling	Leader manages send and commit, followers receive	Each process handles send, receive/ delivery events
Message Propagation	Centralized from leader to followers	Sequential around the ring
Ordering	Leader-determined total order	Distributed determination using logical clocks for message timestamps
Failure Handling	Leader election and recovery	Process bypass and message retransmission

Table 6-2 Parameters of the Simulation

Symbol	Meaning	Values
N	Number of nodes within the cluster	3, 5
π	Message processing delay (ms)	3
μ	Message transmission delay (ms)	30
λ	Message arrival rate (messages/s)	25
SMT	Simulation duration (s)	125000, 2500000

The parameters we used for the simulation and their values are summarized in Table 6-2. The variables were carefully chosen to replicate a real-world message processing and transmission setup between the leader node and the followers in a typical Raft protocol environment. These values were meant to provide a helpful evaluation of the system's performance while also reflecting real-world conditions. We developed a simulation environment that closely matched

the behaviour of a Raft leader in actual situations by considering variables like message processing delay and inter-transmission delay. The system cluster in our simulation has varying group sizes of $N = 3$ and $N = 5$ nodes. The simulation time (*SMT*) we considered ranges from 125,000 to 2,500,000 seconds. We utilised a Poisson distribution with an arrival rate of $\lambda = 25$ messages per second to model the arrival of communications. We also included an average message processing delay $\pi = 3\text{ms}$ to capture the processing time for each message. The inter-transmission delay between messages had a mean of $\mu = 30\text{ms}$ and was distributed exponentially. These decisions were taken to replicate the system's genuine message arrival and processing timeframes, as we envisage that message processing should happen quickly, hence influencing our selection of a small value. In the literature [150], a ping rate of less than 100ms is considered acceptable, but for optimal performance, latency in the range of 30-40ms is desirable, and this formed our choice of $\mu = 30\text{ms}$.

Performance Metrics

The proposed protocols' message complexity, latency, and throughput are evaluated using three performance criteria. For enabling accurate comparisons and evaluations of protocol performance, these performance measures are crucial, in our opinion. The following list includes descriptions of various performance metrics definitions and how they are calculated:

Message Complexity: Given that the focus of this study is on reducing Raft leader load, comparing message overhead would be useful to determine how protocol improvement affects the original Raft. Theoretically, for each protocol, the total number of unicasts needed to fulfil a client request is evaluated. As a result, the protocol with the lowest number of unicasts is considered to have a lower message cost.

Latency: In the Raft variants, latency is calculated as the time elapsed between when a client request arrives at the leaders' incoming queue and when the client receives a response from the leader. For example, let t_c and t_l be the instance of times when a client sends a request to the leader and receives a response from the leader respectively; $t_l - t_c$ defines the maximum latency response for that request. The maximum latencies for each duration were computed, and the experiment was repeated 10 times for a confidence interval of 95%.

Throughput: we computed the throughput as the average number of messages delivered (aNoMD) within the simulation duration; like latencies, we calculated a 95% confidence interval for 10 experiments for each simulation duration.

6.2 Evaluation

The simulation ran using a varying arrival rate of 25, 50, 100..., and 1300 messages per second and message processing and transmission delay of 3ms and 30ms to evaluate the performance of Raft and its variants, as discussed in Section 5.1. We ran the simulation 10 times for each arrival rate to calculate the 95% confidence interval. The expanded arrival rates help us to carefully observe the Raft protocol's behaviour and its variations. We get essential insights and can make significant inferences based on these critical parameters by observing how the systems behave at different arrival rates.

Message Complexity

Theoretically, when $N = 3$, RaftCh offers a message complexity of 3 unicasts (2 unicasts to replicate request from the leader, L to all followers, F_1 and F_2 ; 1 ack(m) unicast from the last follower to the leader, (see Figure 5.6a) per client write request and TO-delivery from leader to the client, whereas Raft needs 4, unicasts (2 unicasts to replicate request, 2 ack(m) unicasts from the two followers to the leader) and TO-delivery from leader to the client. RaftCh and RaftBf have identical message costs compared to Raft, for example, in an $N = 5$ cluster size, both RaftCh and RaftBf exhibit a message complexity of 5 unicast per client write request while Raft incurs a message complexity of 8 unicast per client write request. Consequently, for $N \geq 3$, specifically odd number, Raft requires $2(N - 1)$ whereas RaftCh or RaftBf protocols exhibit an expected message complexity of N per client request.

Furthermore, the leader in RaftCh and RaftBf, regardless of N , is expected to receive 1 follower ack per client request, unlike $(N - 1)$ follower acks received in Raft. For example, the leader in RaftCh and RaftBf with $N = 5$ is expected to receive 1 follower ack(m) per client request, while the leader receives 4 follower acks(m) in Raft.

6.2.1 Raft and RaftCh

In Figure 6.1, the average maximum latency values are plotted against the write request rate in a cluster of group size $N = 3$. The figure indicates that Raft and RaftCh have comparable latencies up until $\lambda = 800$, with Raft doing relatively better than RaftCh. However, Raft shows a sign of saturation when $\lambda > 400$, as latencies increment did not result in a significant increase in throughput, see Figure 6.2. Still, it quickly saturates as the arrival rate continues to increase, $\lambda > 800$, while RaftCh remains relatively in a steady state even as the arrival rate increases further. This difference can be attributed to the design differences in RaftCh such as the usage of chain topology for logical message dissemination, sequential log replication, and

a single explicit acknowledgement from the last follower on the chain is required before a leader commits a message as outline in Section 5.5, and this helps RaftCh maintain its performance even at increasing arrival rates.

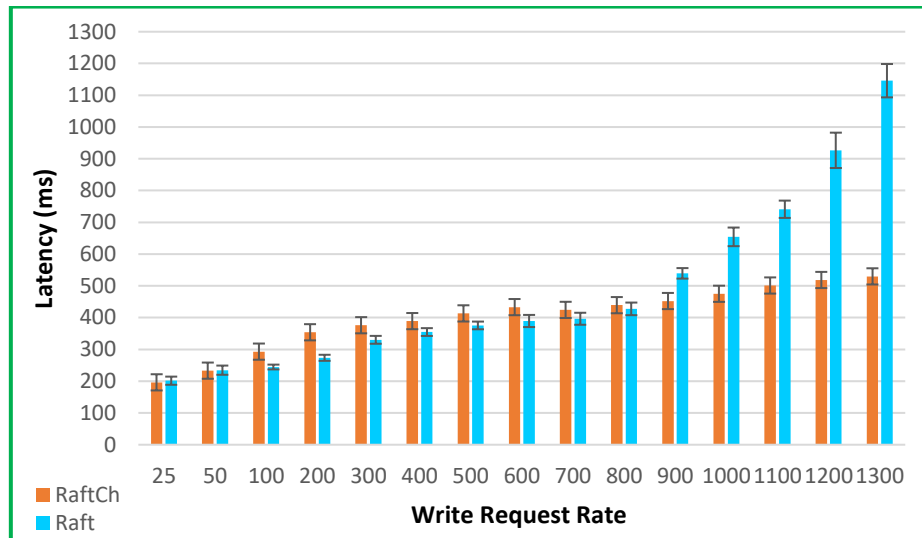


Figure 6.1 Latency Comparison between RaftCh and Raft

In contrast, in the Raft, the leader executes acknowledgements from all the followers even though it requires one ack from any follower, including itself, to commit a sequence number assigned to messages. As the figure shows, this becomes a bottleneck as the clients' arrival rates increase.

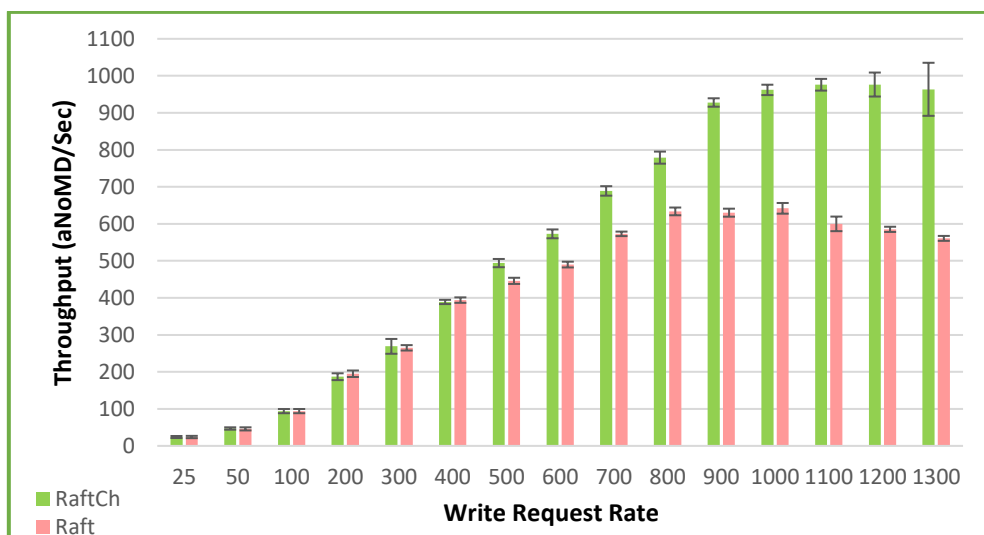


Figure 6.2 Throughput Comparison between Raft and RaftCh

Moreover, Figure 6.2 plots the average maximum throughput values against the varied write request rate in a cluster of group size $N = 3$. Our observation shows that the throughput values of both protocols are nearly identical as both protocols' throughput increases as the arrival rate

increases until it reaches a steady state. However, RaftCh maintains a significantly higher throughput than Raft for $\lambda > 400$, with a throughput improvement of 10% at $\lambda = 500$ and 42% at $\lambda = 1300$. This significant throughput improvement in the RaftCh can largely be attributed to the sequential log replication and a single ack required by the leader to commit the sequence number attached to the client request. It is worth noting that Raft throughput degrades further at $\lambda \geq 1100$, indicating the Raft system's capacity is exhausted.

Table 6-3 Performance Comparison for RaftCh, $N = 3$

Write Request Rate	Latency	Throughput
25	3%	0%
50	1%	2%
100	-20%	0%
200	-29%	-4%
300	-14%	1%
400	-10%	-1%
500	-10%	10%
600	-11%	14%
700	-7%	17%
800	-3%	19%
900	16%	32%
1000	27%	33%
1100	32%	39%
1200	44%	40%
1300	54%	42%

Table 6-3 presents the performance improvement of RaftCh over Raft in a cluster of $N = 3$. As the table shows, Raft outperformed RaftCh in average maximum latency up to $\lambda = 300$, that is, when the system is slightly loaded. At the same time, the Raft latency difference over RaftCh is relatively low between 400 and 800 rates since the difference is 11% or less, and this suggests that Raft and RaftCh have comparable average maximum latencies at these write request rates ($\lambda = 400, 500, \dots, 800$). This significant latency improvement of Raft over RaftCh can be attributed to the fact that in Raft, as the system is slightly loaded, the leader communicates directly with both followers. This reduces the communication path length and minimizes the round-trip time (RTT) for log entry replication while each log entry in the RaftCh must traverse from the leader to the first follower and then to the second follower, effectively doubling the RTT compared to direct leader-to-follower communication. In addition, in the Raft, the parallel

replication and faster acknowledgement process (since the leader is attached to only two followers), lead to lower latency and higher throughput, as multiple log entries can be in-flight concurrently while the sequential nature in RaftCh imposes higher latency for each log entry and limits the throughput as each step in the chain must complete before the next can begin. However, as the write request rate increased beyond $\lambda = 800$, RaftCh demonstrated a substantial improvement over Raft regarding average maximum latency. This occurs because, as more requests arrive at the leader process, the requirement for majority acknowledgement to commit a request in Raft becomes a bottleneck. This results in requests waiting longer to be processed by the leader, thereby increasing latency and reducing throughput. Thus, for the write request rate, $\lambda > 400$, RaftCh did offer significantly better average maximum throughput than Raft.

6.2.2 Raft, RaftCh and RaftBf

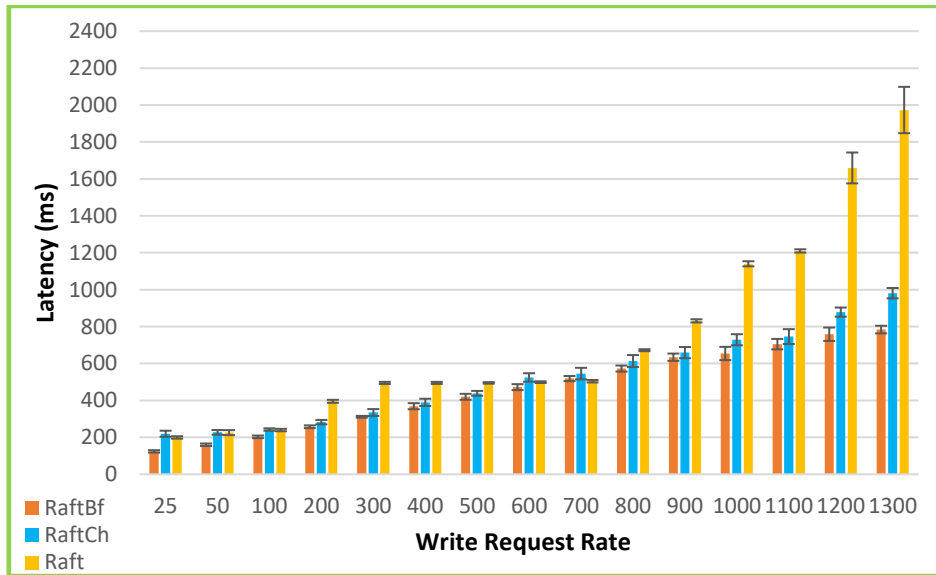


Figure 6.3 Latency Comparison for Raft, RaftBf, and RaftCh

In this section, we aimed to reflect on the simulation results of the Raft variants with Raft in a cluster environment of $N = 5$. Figure 6.3 compares the latency of the Raft and the various Raft variations across varying arrival rates. The graph clearly shows that Raft-variant protocols consistently provide faster client responses than Raft for all the simulated arrival rates. Hence, RaftBf improves latency performance by 38% at $\lambda = 25$ and 60% at $\lambda = 1300$ over Raft, while RaftCh improves latency performance by 29% at $\lambda = 200$ and 50% at $\lambda = 1300$ over Raft. These improvements can be attributed to the design frameworks of both Raft variants, as parallel replication of client requests in the RaftBf minimizes the number of sequential jumps

needed for log entry dissemination, thereby improving latency. In contrast, sequential replication in the RaftCh enhances latency by allowing multiple requests to be replicated along the chain concurrently, though the overall latency may still be higher compared to parallel replication due to its sequential nature. The idea of requiring a single acknowledgement by the leader to commit a request in Raft variants is a plausible approach for improving both latency and throughput, as it speeds up the commitment process by minimizing waiting times. On the other hand, Raft did offer better average maximum latency than RaftCh when the write request rate is slightly low ($\lambda = 25, 50, \text{and } 100$) but shows a latency spike at $\lambda \geq 200$ indicating a sign of saturation, which remained relatively constant till $\lambda = 700$ and quickly saturates at $\lambda \geq 800$. This observed behaviour can be attributed to the ack rate of 4λ that the leader needs to process before committing a sequence number on the client request in a cluster of five processes. However, when comparing the throughput of Raft and the Raft variants, Raft variants consistently outperform Raft in average maximum throughput, as illustrated in Figure 6.4. It is worth noting that Raft variants have comparable throughputs, with RaftBf having a throughput improvement of 58% at $\lambda=1300$ over Raft and RaftCh having throughput improvements of 59% at $\lambda=1300$ over Raft. However, Raft throughput increases when the system is slightly loaded (between 25 and 400 rates), but it remains relatively constant as $\lambda > 400$. However, at $\lambda > 1000$, the throughput suffered further degradation.

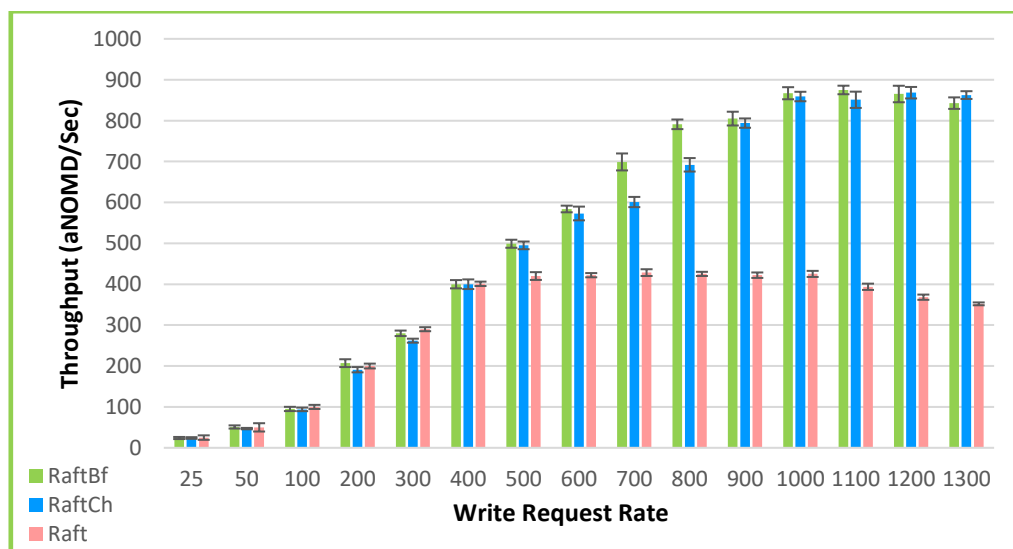


Figure 6.4 Throughput Comparison for Raft, RaftBf, and RaftCh

Table 6-4 and Table 6-5 show the performance improvement results of Raft-variants over Raft. As the tables show, there is a significant latency improvement for both Raft-variant approaches. However, the RaftCh experiment demonstrated worse latency performance compared to Raft, particularly at various request arrival rates ($\lambda=600$, and 800).

The experiment shows a latency degradation of up to -10% at $\lambda=25$ and a throughput reduction of -11% at $\lambda=300$. It is challenging to explain this result, though it may be related to the longer chain; each request traverses before reaching the last follower and the transmission delay, which could mean that each follower transmission delay distribution produced a high delay at that instant before forwarding a copy of the request. Also, a sudden spike or burst on client request arrival means that Raft parallel replication will handle it better compared to RaftCh's sequential approach. This sudden spike may lead to higher contention for CPU and memory which increases latency in RaftCh due to its sequential replication approach since each process waits for the previous process to execute incoming requests. These advantages of Raft where the system is slightly loaded ($\lambda=25, 50, 100$) as observed from Table 6-4 and Table 6-5 make Raft better suited for small clusters ($N=3$ or 5) where the overhead of managing a leader is outweighed by the performance gains in log replication and consistency management. The results in these tables are notable since they show that both Raft-variants significantly enhance their throughput when the number of write requests rises from 500 to 1300 while the system is operating at a steady state. However, Raft produced a better throughput when the system is lightly loaded, $\lambda = 25, 50, 100, \dots, \text{and } 400$, even though at $\lambda = 400$, Raft and its variants offer considerable throughput.

Table 6-4 Latency Performance Improvements of Raft-variants

Latency		
Write Request Rate (λ)	RaftBf	RaftCh
25	38%	-10%
50	29%	-1%
100	15%	-1%
200	35%	29%
300	37%	32%
400	26%	21%
500	15%	11%
600	5%	-5%
700	-3%	-8%
800	15%	9%
900	24%	21%
1000	43%	36%
1100	42%	38%
1200	54%	47%
1300	60%	50%

Overall, in terms of throughput, the throughputs offered by both Raft-variants are generally equivalent.

Table 6-5 Throughput Performance Improvements of Raft-variants

Throughput		
Write Request Rate (λ)	RaftBf	RaftCh
25	-4%	-4%
50	2%	-6%
100	-5%	-6%
200	3%	-5%
300	-4%	-11%
400	0%	0%
500	16%	15%
600	28%	26%
700	39%	29%
800	46%	39%
900	48%	47%
1000	51%	50%
1100	55%	54%
1200	57%	58%
1300	58%	59%

Furthermore, the subsequent simulation focuses on the performance comparison of the different Raft variants with DCTOP, a novel ring-based leaderless total order protocol developed in this study. Each Simulation run took approximately 35 to 695 hours and simulated Raft variants and DCTOP cluster operational period of at least 29 days following the simulation discussed in Section 6.1. This prolonged simulation period provided us with an extensive understanding of the system's behaviour across an extended practical operating period.

6.2.3 RaftCh and DCTOP

Figure 6.5 plots the average maximum latency values against the number of clients transmitting messages, ranging from 5,000 to 100,000 clients in a cluster of group size $N = 3$. The figure clearly indicates that RaftCh has lower latencies than DCTOP for all messages sent during the simulation, with a 4% average maximum latency improvement. This improvement can be attributed to the presence of message processing delay included in simulating DCTOP, aligning it with the RaftCh operations. This contrasts with the simulation of DCTOP reported in Section 4.2.1, where we considered zero processing delay. Additionally, when $N = 3$, the response time

of the leader process to the clients will be quicker, thereby improving the latency performance of RaftCh.



Figure 6.5 Latency Comparison between RaftCh and DCTOP

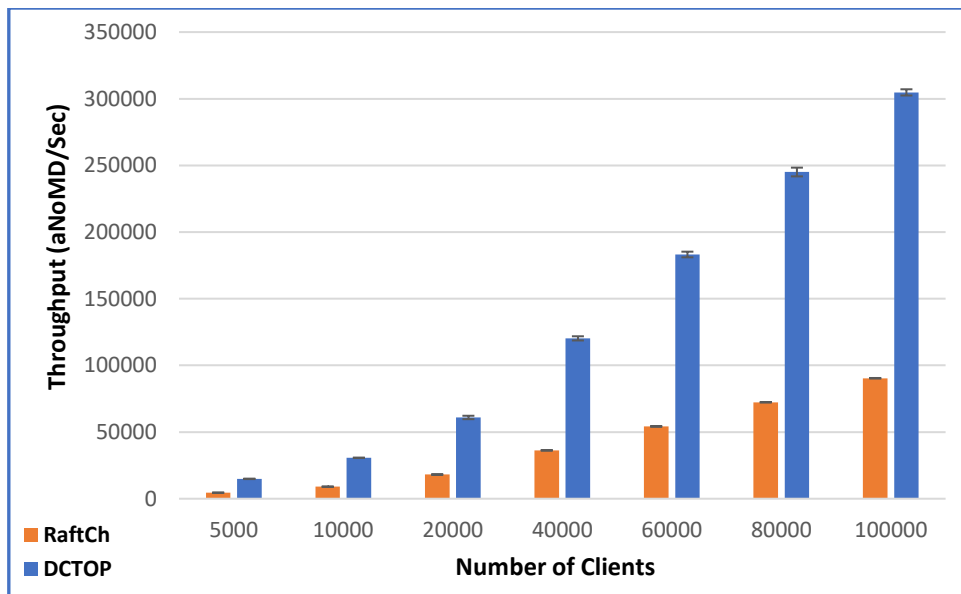


Figure 6.6 Throughput Comparison between RaftCh and DCTOP

Figure 6.6 plots the maximum throughput values against the number of messages transmitted in a cluster of group size $N = 3$. The figure depicts that DCTOP has a significantly higher throughput than RaftCh for all the number of messages sent during the simulation time. Thus, we can make two observations from Figure 6.6. Firstly, the throughput values of both protocols

are nearly identical as both protocols' throughput increases as we increase the simulation's durations. This attribute can be explained by the fact that more extended simulations might help the system use its resources more effectively, allowing more client requests to be processed. As a result, the system's total throughput increases due to an increment in the number of client messages processed. Secondly, for all the simulation times we considered, we observed that DCTOP had a 70% average maximum throughput improvement over RaftCh. This improvement can be attributed to the designed nature of DCTOP. In DCTOP, every process can coordinate client messages independently, enabling parallelism and concurrency of executing client requests, unlike in RaftCh where the leader only coordinates the client messages and, thus, underutilizing the capabilities of follower processes since they spend most of their time replicating the message from the leader rather than individually handling client requests. For example, in a cluster of $N = 3$ processes, if the leader process coordinates an average of 5000 messages, then the DCTOP is expected to coordinate an average of 15000 messages, while assuming each process coordinates an average of 5000 messages, respectively. Finally, because message coordination is distributed across many processes, the DCTOP has a higher total throughput than the RaftCh.

Table 6-6 Performance Improvement of DCTOP (throughput) and RaftCh (latency) against each other

#Clients	Latency	Throughput
5000	2%	70%
10000	5%	71%
20000	3%	70%
40000	4%	70%
60000	4%	70%
80000	4%	70%
100000	3%	70%

Table 6-6 shows the improvements in latency and throughput obtained during the comparison of DCTOP and RaftCh for $N = 3$. Remarkably, DCTOP offers a significant throughput improvement compared to RaftCh, affirming the effectiveness of ring-based leaderless order protocols in achieving maximum attainable throughput. In addition, we observed a slight latency improvement of RaftCh over DCTOP. These results are especially noteworthy because they were achieved during the simulation's steady-state phase.

6.2.4 RaftBf and RaftCh with Zero Processing Delay

Figure 6.7 plots the maximum latency values against the number of messages transmitted for a cluster of group size $N = 5$. In addition to the simulation discussed in Section 6.1, we assumed a zero-processing delay for the simulation of RaftCh and RaftBf since both protocols are leader-based and have similar operational capacities.

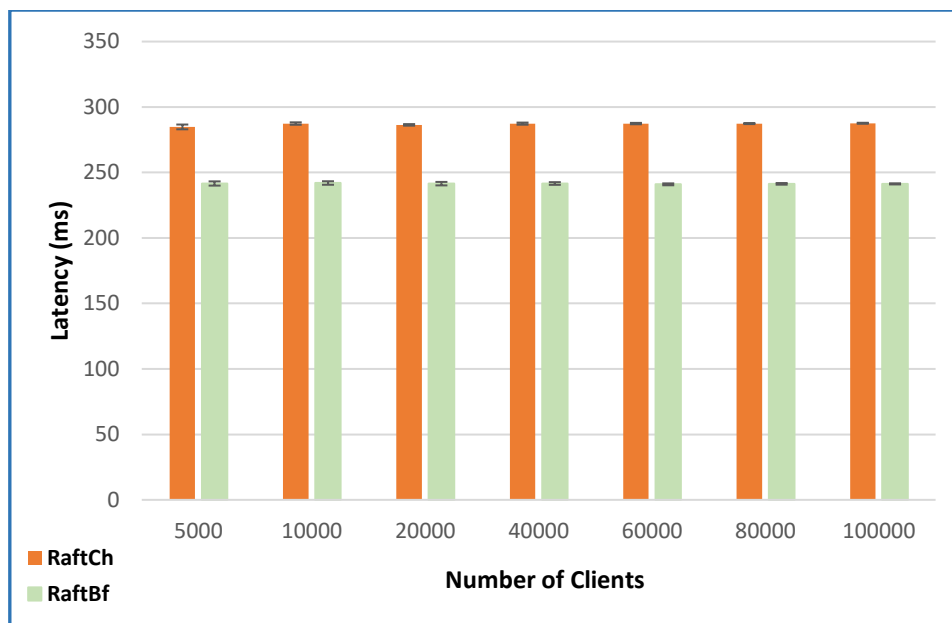


Figure 6.7 Latency Comparison for RaftBf and RaftCh (Zero Processing Delay)

The figure shows that RaftBf has shorter latencies than RaftCh for all the workloads we considered. RaftBf maintained a consistent 19% average maximum latency improvement over RaftCh for all the workloads. This can be attributed to the logical structural benefits of the RaftBf over the RaftCh. RaftBf generally exhibits shorter latencies in comparison to RaftCh. This is primarily attributed to the concurrent replication of client write requests across the fork, even though the reception time of these requests may vary among followers. The last follower on either fork in a RaftBf generates an $\text{ack}(m)$ and transmits it to the leader, making message propagation more effective. As a result, there will be less delay because the messages will not have to travel the entire framework sequentially.

In contrast, RaftCh adds more latency because a message from the leader passes from the leader to the followers sequentially until it reaches the final follower process in the chain. The logical design structure in RaftBf offers improved message propagation compared to RaftCh, providing lower latencies and better performance. For example, in a cluster size $N = 5$, a message in RaftBf is required to make 2 jumps to travel from the leader to the final follower process on either fork. RaftCh, on the other hand, requires that messages traverse the entire chain, making 4 jumps to reach the final follower process within the same cluster size.

Consequently, RaftBf boasts of a shorter path than RaftCh, requiring fewer jumps for messages to reach all destinations. This contributes to reduced latency and increased efficiency.

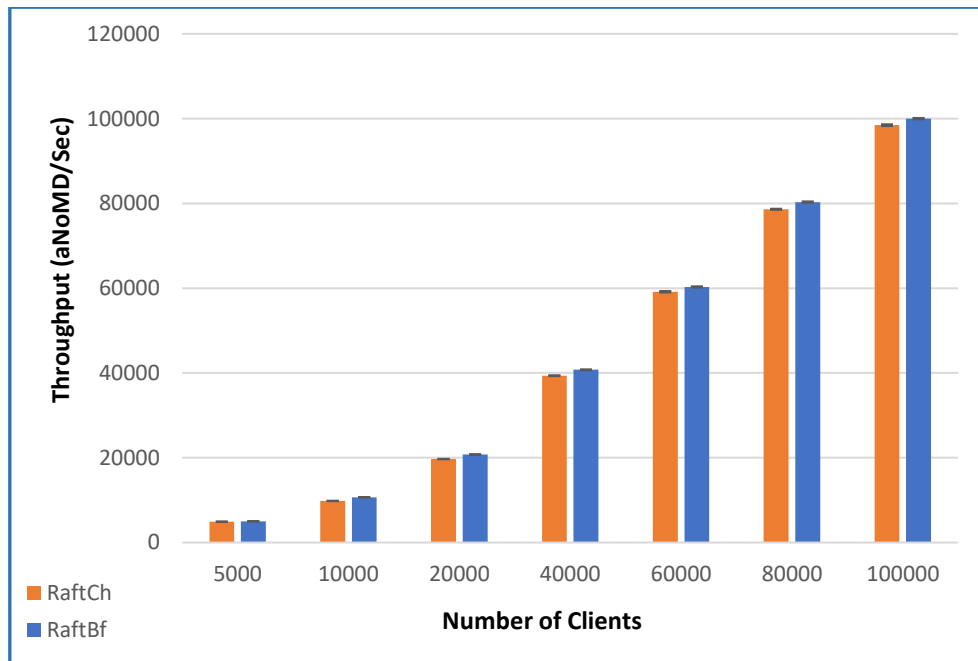


Figure 6.8 Throughput Comparison for RaftBf and RaftCh (Zero Processing Delay)

When considering the average maximum throughput, as shown in Figure 6.8, we observed that both RaftBf and RaftCh have comparable throughput levels, with RaftBf showing an average of 3% throughput improvement over RaftCh for all the simulated workloads which can attributed to the explanation given above. This implies that both protocols can efficiently process messages and oversee growing workloads, with only slight variation in throughput.

Table 6-7 Performance Improvement of RaftBf over RaftCh for Zero Processing Delay

#Clients	Latency	Throughput
5000	18%	2%
10000	19%	8%
20000	19%	5%
40000	19%	3%
60000	19%	2%
80000	19%	2%
100000	19%	2%

The improvements in latency and throughput achieved under conditions of zero processing delay are shown in Table 6-7. Notably, RaftBf consistently outperforms RaftCh in performance across all numbers of messages transmitted within the simulation period.

However, this improvement was observed during the simulation's steady-state period, demonstrating RaftBf's persistent and stable advantages over RaftCh.

6.2.5 RaftBf and RaftCh with Non-Zero Processing Delay

The simulation results reported in this section are like those discussed in Section 6.1, except that we included message processing delay to study its impact on the performance of both protocols.

Figure 6.9 shows that RaftBf consistently has shorter latencies than RaftCh across all workloads evaluated in this section, consistent with the conclusions stated in Section 6.2.4. In addition, RaftBf outperforms RaftCh by a significant average maximum latency improvement of 37% for all the simulated workloads. This improvement can be attributed to the message processing delay and other design characteristics of RaftBf, as explained in Section 6.2.4.

However, RaftBf and RaftCh show comparable throughput levels, as shown in Figure 6.10, which shows the average maximum throughput plotted against the various workloads. For all simulated workloads, RaftBf outperforms RaftCh with an average throughput improvement of 5%. Notably, the addition of message processing delay had a minimal impact on the throughput performance of either protocol. The throughput improvement increased slightly, from 3% when the simulation used zero processing delay to 5% when message processing delay was employed.

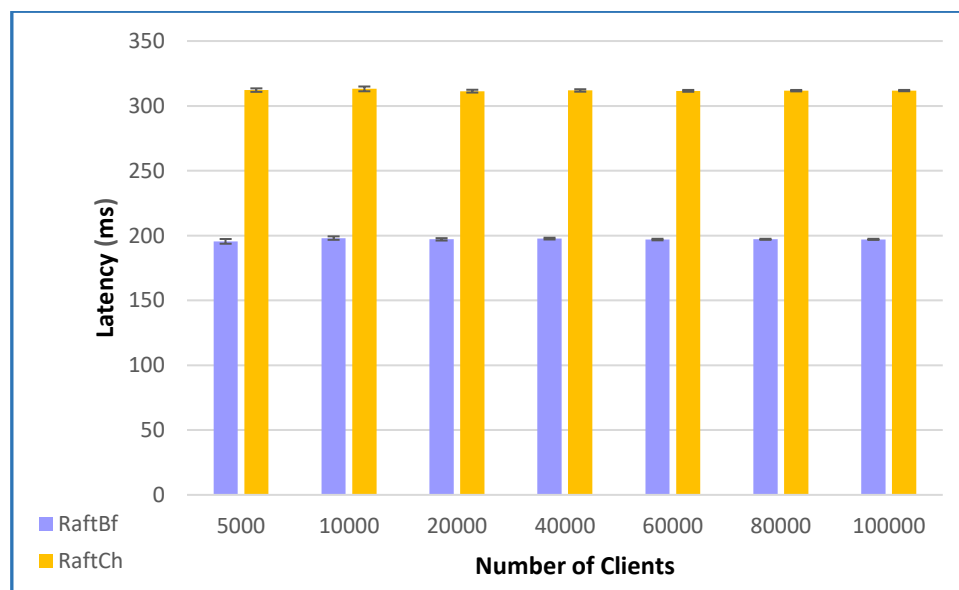


Figure 6.9 Latency Comparison for RaftBf and RaftCh (Non-Zero Processing Delay)



Figure 6.10 Throughput Comparison for RaftBf and RaftCh (Non-Zero Processing Delay)

Table 6-8 Performance Improvement of RaftBf over RaftCh for Non-Zero Processing Delay

#Clients	Latency	Throughput
5000	37%	6%
10000	37%	5%
20000	37%	4%
40000	37%	5%
60000	37%	5%
80000	37%	5%
100000	37%	5%

Table 6-8 provides the improvements in latency and throughput obtained under conditions of non-zero processing delay. Remarkably, RaftBf performed better than RaftCh for all the number of client requests considered in the steady state phase of the simulation, demonstrating the consistent benefits of RaftBf over RaftCh.

6.2.6 RaftCh (N = 3, and N = 5)

In this section, we intend to compare the simulation results of RaftCh over varied cluster environment sizes (N = 3 and N = 5). As depicted in Figure 6.11, when N = 3, the RaftCh offers a better latency than when N = 5 for all the simulated workloads. Specifically, N = 3 provides a 32% latency performance improvement over N = 5. This can be attributed to the fact that when the chain is shorter (N = 3), the leader process will send a response to the clients more quickly.

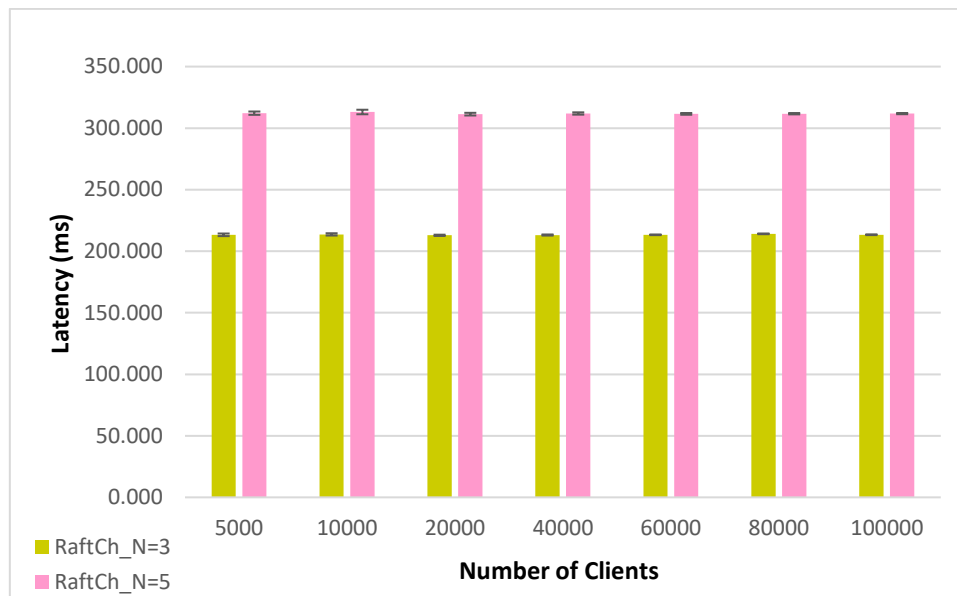


Figure 6.11 Latency Comparison of RaftCh for $N = 3$ and $N = 5$

However, as the chain gets longer ($N = 5$), the leader process responds to clients more slowly, leading to higher latencies. Thus, in a RaftCh, $N = 3$ offers a better latency performance than $N > 3$ for the simulated workloads.



Figure 6.12 Throughput Comparison of RaftCh for $N=3$ and $N=5$

Figure 6.12 plots the average maximum throughput against the number of client requests for different cluster environment sizes. From the comparison, we observed that $N = 3$ and $N = 5$ have comparable average maximum throughput, with $N = 3$ offering a slight average improvement of 3% across all simulated workloads.

This relatively low improvement indicates that RaftCh provides a similar average maximum throughput regardless of the group size.

Table 6-9 Performance Improvement of RaftCh when $N = 3$ over when $N = 5$

#Clients	Latency	Throughput
5000	32%	3%
10000	32%	2%
20000	32%	3%
40000	32%	3%
60000	32%	3%
80000	31%	3%
100000	32%	2%

Table 6-9 provides the RaftCh improvements in latency and throughput obtained under different cluster sizes. Interestingly, RaftCh performed better with $N = 3$ than $N = 5$, offering a 32% latency and an average of 3% throughput improvements for all the examined numbers of client requests within the steady state period of the simulation. This recurring pattern demonstrates the benefits of using a smaller group size $N = 3$ instead of $N = 5$ in RaftCh.

6.2.7 RaftCh, RaftBf and DCTOP

In this section, we aimed to reflect on the simulation results of the Raft variants with DCTOP in a cluster environment of $N = 5$. Figure 6.13 compares the latency of DCTOP and the various Raft variations across varying group sizes (N) and message processing delays. The graph clearly shows that Raft-variant protocols consistently provide faster client responses than DCTOP for $N = 5$. This is due to the Raft-variant protocols' leader capacity to quickly commit client message sequence numbers and return results to clients with a single acknowledgement from the last follower process unlike in DCTOP where the message stability, which inherently implies crashproof, is only known by the last process of the message origin. This stable message can be TO-delivered if it is at the head of the delivery queue. After which, it sends an acknowledgement, $\mu(m)$ to its clockwise neighbour, which is eventually passed around the ring structure until the ACN of the $\mu(m)$ origin receives $\mu(m)$. Therefore, except for the last process of the message origin, other processes require receiving $\mu(m)$ to know that the messages within their buffer are stable and crashproofed and can be TO-delivered to the high-level application process. As noted earlier, a message becomes crashproof once it has been received by at least $f+1$ processes (see Section 3.3.3). This entire process may contribute to additional latency.

RaftBf improves latency performance by 65% over DCTOP, while RaftCh improves latency performance by 45% over DCTOP. This can be attributed to the design framework of both Raft variants as earlier explained, precisely the novel idea of the leader committing a message using a single acknowledgement from the last follower process. Regarding latency, these findings clearly show that Raft variants regularly outperform DCTOP across all simulated workloads.

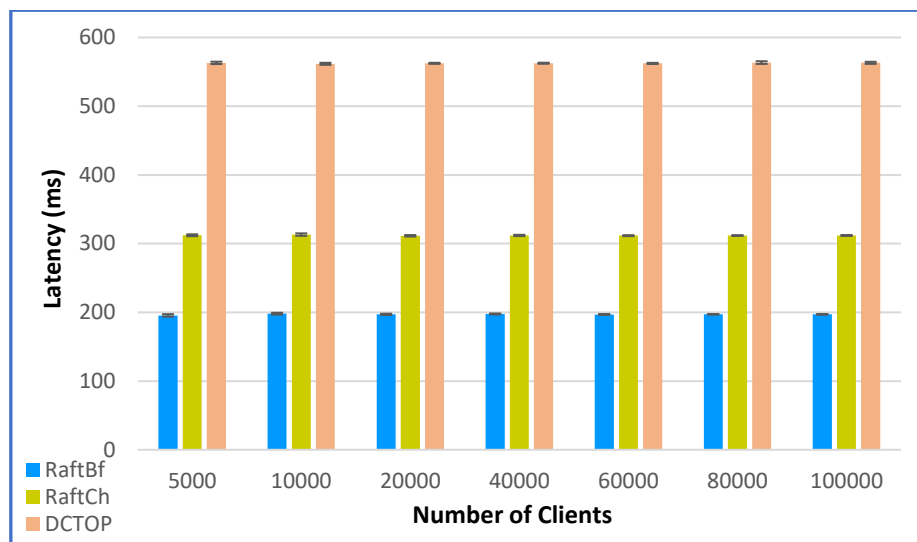


Figure 6.13 Latency Comparison for RaftBf, RaftCh, and DCTOP

However, when comparing the throughput of DCTOP and the Raft variants, DCTOP consistently outperforms both RaftBf and RaftCh in average maximum throughput, as illustrated in Figure 6.14. Across all simulated workloads, DCTOP achieves an 82% average maximum throughput improvement over RaftBf and an 83% average maximum throughput improvement over RaftCh. This can be attributed to the aggregated throughputs of the individual processes within the DCTOP ring cluster, as each process executes client requests independently and sends responses to the high-level application process. By distributing the workload across all processes in the cluster, DCTOP achieves parallelism that improves overall system efficiency and reduces bottlenecks typically associated with leader-based approaches. Additionally, the independent execution of client requests minimizes contention and ensures that the system remains responsive even under high loads enhancing throughput and provides better fault tolerance, as the failure of a single process does not significantly disrupt the overall operation of the system. The results of these investigations consistently demonstrate that ring-based leaderless total order protocols, such as DCTOP, can achieve the optimum achievable throughput. The 82% and 83% throughput improvement over RaftBf and RaftCh, respectively, show that both Raft variants possess comparable throughput capabilities.

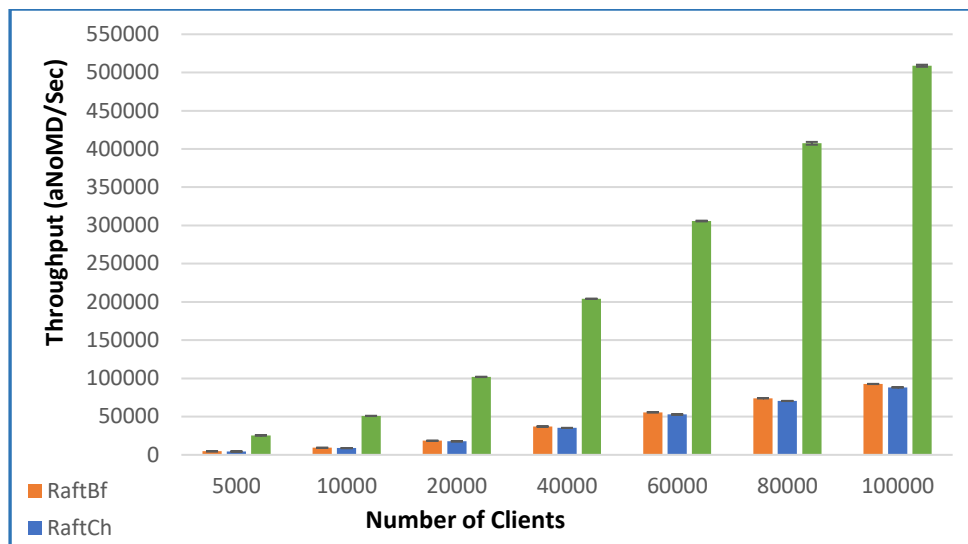


Figure 6.14 Throughput Comparison for RaftBf, RaftCh, and DCTOP

Table 6-10 Latency Performance Improvement of RaftBf and RaftCh over DCTOP

#Clients	RaftBf	RaftCh
5000	65%	45%
10000	65%	44%
20000	65%	45%
40000	65%	45%
60000	65%	45%
80000	65%	45%
100000	65%	45%

Table 6-10 shows the latency performance improvements achieved by the Raft variants over DCTOP. The results show considerable improvements in latency, with RaftBf showing a remarkable 65% latency improvement when compared to DCTOP. Similarly, RaftCh outperforms DCTOP by 45% in terms of latency. These consistent findings show that Raft variants, as leader-based total order protocols, consistently do better than DCTOP, a leaderless ring-based total order protocol, in terms of latency across all simulated workloads.

However, Table 6-11 provided the DCTOP throughput improvements over the Raft variants. The result shows that DCTOP has a significant throughput improvement of 82% over RaftBf and an 83% throughput improvement over RaftCh. The findings from this steady state simulation results reinforce the evidence that ring-based total order protocols, such as DCTOP, are proven to achieve the highest attainable throughput. This is attributed to the ability of each process to execute clients' requests independently such that the system throughput becomes the aggregated throughput of all the processes within the cluster environment.

In contrast, in the Raft variants, the leaders' throughput is distributed across all the followers, thus making the system's throughput dependent on the throughput of the leader process.

Table 6-11 Throughput Performance Improvement of DCTOP over RaftBf and RaftCh

#Clients	DCTOP Improvement over RaftBf	DCTOP Improvement over RaftCh
5000	82%	83%
10000	82%	83%
20000	82%	83%
40000	82%	83%
60000	82%	83%
80000	82%	83%
100000	82%	83%

6.3 Summary

We have innovatively modified the widely used Raft order protocol by introducing Raft variants through a novel approach. This approach involves the utilization of a single acknowledgement for committing the order placed on clients' requests and introduces new logical message dissemination compared to the traditional Raft. The outcome is a reduction in latency and an enhancement in throughput in most scenarios we considered. This novel approach has led to the development of new total order protocols called RaftCh and RaftBf. This chapter presented a comprehensive performance evaluation of the traditional Raft, the Raft-variants (RaftBf, RaftCh) and the DCTOP protocols. The simulation results reflect the following observations:

1. Raft variants provide performance advantages over Raft as the arrival rate increases, though Raft did better with lower arrival rates.
2. Our distinct single acknowledgement for committing the order placed on clients' requests by the leader and the new logical message dissemination is a viable substitute for the same task in the traditional Raft protocol, which requires at least a majority $\left\lceil \frac{N+1}{2} \right\rceil$ of acknowledgements.
3. The design modifications we made over the Raft, a leader-based protocol, provide a significant latency performance improvement over DCTOP, a ring-based leaderless protocol, for all the scenarios considered in the simulations.
4. RaftBf consistently outperforms RaftCh in latency improvements for a cluster size of $N > 3$, particularly $N = 5$, regardless of the presence or absence of message processing

delay. Thus, RaftBf consistently provides lower latencies and faster client responses than RaftCh for all the simulated workloads.

5. DCTOP consistently provides significant throughput improvement over Raft variants in all the simulated workloads.
6. Additionally, we have shown that the RaftCh protocol offers better latency performance with a cluster size of $N = 3$ than with $N = 5$ for all the simulated workloads.

Consequently, our results appear to show that Raft, a leader-based protocol with the proposed variants, provides better average maximum latency improvements than DCTOP. However, compared to Raft variants, DCTOP shows significant throughput improvements. These results seem to demonstrate the trade-offs between throughput performance and latency when comparing leader-based total order protocol and leaderless ring-based total order protocol. These trade-offs may depend on the specific needs of the distributed system and use cases:

- a. Raft-variants, leader-based protocols could be essential where low latency and strong consistency are critical requirements such as CockroachDB [150] and Amazon DynamoDB [151].
- b. DCTOP, a leaderless ring-based total protocol, might be a preferred choice where high throughput and high availability are necessary requirements such as Amazon Dynamo [152].

Finally, the system developer's decision between leader-based total order protocol and leaderless ring-based total order protocol should follow the system architecture's requirement, the limitations and the trade-offs that best support those requirements.

Chapter 7

Conclusion

This chapter summarizes the work presented in this thesis, focusing on addressing the inefficiencies and limitations of existing state-of-the-art research in leaderless ring-based total order protocols such as LCR and leader-based total order protocols such as Raft. The contributions and limitations of these works are outlined before we discuss future research in the area. The rest of this chapter is arranged as follows: The main contributions in this thesis are outlined in Section 7.1. Then, in Section 7.2, we review our recommendations for potential future applications. The limitations of our research are described in Section 7.3. Finally, in Section 7.4, we consider various future research directions.

7.1 Thesis Summary

In this thesis, we have explored a comprehensive background for the state of the art in total order processing. Additionally, we have explored how the potential limitations of existing work on leaderless ring-based total order protocols like LCR, as well as leader-based total order protocols like Raft, can be addressed.

In Chapter 3, we developed the Daisy Chain Total Order Protocol-DCTOP, a unique ring-based leaderless total order protocol. The motivation for this work was to develop a ring-based leaderless total order protocol that addresses several key challenges: (i) using a logical clock for message timestamping, which can be represented with a single integer, unlike a vector clock used for message vector timestamp that requires N bits, one for each process, thereby reducing information overhead; (ii) dynamically determining the last process to order concurrent messages unlike a fixed last process for performing the same task; and (iii) relaxing the crash failure assumptions to facilitate quicker message delivery, even before the ACN of the message origin declares it stable, as timestamp stability can occur quickly in concurrent messaging scenarios. In addition, we considered two approaches in our work: the greedy sending approach and the fairness-controlled approach.

The key goals of our approach are twofold: (1) improving the LCR protocol's latency by employing Lamport logical clocks for message sequencing and (2) adding the dynamic concept of the "last" process to order concurrent messages while maintaining optimal attainable throughput. We then undertake a detailed performance study of our new protocol, DCTOP, in contrast to the LCR protocol. Throughput and latency were used as the primary metrics for comparison to analyse each performance under a variety of conditions.

Next in Chapter 4, we presented a detailed performance study of our new protocol, DCTOP, in contrast to the state-of-the-art LCR protocol. The results of our extensive evaluation show that DCTOP consistently outperforms LCR in terms of latency across all group sizes ($N=4, 5, 7$, and 9) considered, using both the greedy sending approach and fairness control primitives. Furthermore, the evaluation demonstrates that the use of fairness control primitives led to significant performance improvements. DCTOP consistently exhibited lower latency than LCR for all transmitted messages and varying group sizes. It is important to emphasize that these improvements in DCTOP over LCR can be attributed to the relaxation of the crash failure assumptions, along with other techniques we employed.

In Chapter 5, we examined the Raft total order protocol and developed two variants: Chain Raft (RaftCh) and Balanced Fork Raft (RaftBf). The goal of these variants is to provide alternative logical message dissemination structures and to reduce the number of acknowledgements required for the leader to commit the order assigned to a message in its log. This is achieved by ensuring that the leader process only needs one acknowledgement—either from the last follower in the chain (in RaftCh) or from the last follower in each sub-fork (in RaftBf)—before committing messages to its state machine for execution. This contribution indicates an essential milestone in leader-based total order protocols such as Raft because, to the best of our knowledge, no prior research has investigated how Raft's performance might be improved through variations of the protocol. RaftCh and RaftBf are acceptable for practical implementation because they both meet the criteria for establishing crash tolerance, as evidenced in the Raft protocol. In addition, we improved the performance and scalability of the Raft protocol by adding these Raft variants, eliminating the leader load issues, while attaining increased system effectiveness. The findings of this chapter establish the groundwork for further investigation and acceptance of these variants in Raft protocol implementations and offer insightful information about the possible advantages of using RaftCh and RaftBf in real-world systems.

Finally, in Chapter 6, we presented a performance evaluation of the proposed Raft variants-RaftBf and RaftCh compared to the state-of-the-art Raft protocol and DCTOP, our new leaderless ring-based total order protocol. The evaluation aimed to assess how these protocols perform in terms of throughput and latency under various simulated workloads. Our results show that in the cluster environments we considered, RaftBf and RaftCh consistently maintained low latency compared to Raft and DCTOP across all simulated workloads, although Raft exhibited low latency under light system loads. This indicates that the proposed Raft variants are more effective at reducing latency than both Raft and the leaderless DCTOP system when the system is heavily loaded. However, it is important to note that DCTOP outperforms the Raft variants in terms of throughput. These findings highlight the inherent trade-offs between throughput and latency when comparing leader-based and leaderless ring-based total order protocols. The insights gained from evaluating RaftBf, RaftCh, Raft, and DCTOP contribute to a deeper understanding of their strengths and limitations, helping to guide the selection and implementation of suitable protocols based on the specific needs of a distributed system, with considerations of latency, throughput, and the trade-offs involved.

7.2 Recommendations

We made several critical recommendations based on observations drawn from our results. Firstly, DCTOP offers better latency improvement than LCR in the presence or absence of fairness control primitive. However, DCTOP provides more significant latency improvement in the presence of fairness control primitive than in the absence while maintaining maximum achievable throughput. Thus, DCTOP should be utilised in fairness control clusters of environments for better performance improvement.

Secondly, the findings of this study suggest that for a group size of 3 replicas, RaftCh gains a performance advantage over DCTOP in terms of average maximum latency. Thus, we recommend RaftCh is used even when the system cluster is $N = 3$, irrespective of workloads. Thirdly, in a cluster setting with $N = 5$ replicas, our findings show that RaftBf has lower latency than RaftCh and DCTOP, even though DCTOP offers more significant throughput performance improvements. RaftBf is thus recommended for $N = 5$ clusters because of its enhanced latency performance. RaftBf can also handle various permissible values for N , though $N = 5$ is considered ideal. In addition, the Raft protocol offers better latency and throughput when the system is lightly loaded.

Finally, by considering these insights from our simulation results, system designers and developers can choose and use the right protocols depending on their unique cluster environments and performance requirements.

7.3 Limitations

We have demonstrated through performance studies that DCTOP is a successful protocol for enhancing latency performance. Additionally, we have shown that Raft variants offer more significant latency improvements than Raft and DCTOP through our performance evaluation. However, our performance evaluation only considers crash-free settings. The performance study and evaluations for DCTOP, Raft and its variants were done in a crash-free environment, which is one of their limitations. Crash and failure are regular events in real-world distributed systems, and they can impact the overall system behaviour and performance. As a result, it would be beneficial to investigate and evaluate the performance of DCTOP and Raft variants under crash settings to determine their robustness and ability to manage failure scenarios. This would provide complete insight into their performance in realistic, real-world situations.

Moreover, we observed that the RaftCh and RaftBf protocols effectively improve the Raft protocol's performance and outperform DCTOP in terms of latency in our performance evaluation. However, it is crucial to note that our study was limited to a scenario where a maximum of 100,000 clients could send requests at any one time when comparing Raft variants and DCTOP. In addition, we used a maximum arrival rate of 1300 when comparing Raft and its variants. The RaftCh/RaftBf technique has a crucial limitation in that, like every leader-based system, it has an upper limit on the number of state-modelling requests it can handle at any one time. When the number of client requests exceeds the system's throughput limits, the protocols may experience performance bottlenecks and become unresponsive to client requests. This limitation is inherent in centralised techniques and is recognised by other leader-based protocols like Paxos [38, 39, 153]. When deploying the RaftCh/RaftBf protocols in real-world applications with large client request loads, it is critical to recognise and account for this limitation.

Another critical limitation of DCTOP and Raft variants is the need to evaluate their performance for a cluster size of $N > 9$. Our study was limited to a maximum of at most 9 processes in the cluster setting, which is a constraint for both DCTOP, Raft and its variants implementations.

While this offers valuable information about how they perform within this cluster range, it does not give a thorough knowledge of how these protocols might operate in more extensive deployments, $N > 9$. Additionally, In DCTOP, we assumed that $N=2f+1$ is required for the system to tolerate up to f failures, with f being a crucial parameter for achieving crash tolerance. We implemented two delivery policies, one of which is crashproof. According to the crashproof policy, when a process sends a message, at least $f+1$ operative processes must receive the message before any process can deliver it. The advantage of this approach is that crashproofness for a given message is achieved more quickly, potentially reducing latency. In contrast, the LCR protocol assumes $N=f+1$, meaning that for a message to be delivered, it must be received by all process replicas, which can increase message delivery latency. Therefore, the relaxation in our assumption may be one reason our system outperformed LCR. For the Raft variants, we assumed that the leader process needs to receive only one acknowledgement before it can commit a client message to its state machine for execution. This contrasts with the state-of-the-art Raft protocol, which requires a majority of acknowledgements, including the leader's own, to perform the same task. The combination of different logical message dissemination strategies in the Raft variants, along with the single-acknowledgement requirement for message commitment, likely contributed to the improved performance of the Raft variants compared to the original Raft protocol in most scenarios we considered.

Finally, while the provided discrete event simulation for DCTOP and Raft variants offer interesting improvements over LCR and Raft, it comes with some limitations that should be acknowledged: (i) The simulation assumes that events occur independently, and the future state depends only on the current state (Markov property). This simplifies the modelling but does not capture the complex dependencies and correlations present in real-world systems. (ii) The current implementation does not model any process or communication failures. In real distributed systems, processes can crash, and network partitions can occur. Handling such failures is crucial for a robust total order protocol. (iii) The simulation uses a small number of processes (e.g., $N=4, 5, 7$, and 9). Real distributed systems often involve a much larger number of processes. The small scale may hide issues that only become apparent in larger systems. (iv) The overhead of implementing checkpointing into our system is another performance limitation, as it can result in increased latency, reduced throughput, and higher consumption of computational resources like CPU, memory, and disk I/O.

7.4 Future Research

This section delves into prospective avenues for future research derived from the discoveries and contributions presented in this thesis.

7.4.1 Heterogenous Node Setting

DCTOP has been specially developed and evaluated for homogenous clusters, where machines are connected by a dedicated, fully switched network. However, it might not be appropriate for networks that are not dedicated or environments with diverse processes. To attain high performance in diverse situations, it will be interesting to investigate how ring-based communication patterns might be combined with other communication patterns, such as tree-based. Innovative solutions for effective communication in various cluster environments might result from understanding the dynamics and potential trade-offs of mixing these patterns.

7.4.2 Additional Implementation and Experiments

DCTOP and Raft variations can be extended to accommodate a wide range of failure scenarios. This involves implementing procedures for dealing with process failures and maintaining data consistency and durability in the presence of failures. In addition, DCTOP and the Raft variants are currently investigated for specific cluster sizes (for example, $N = 3$ and $N = 5$ for Raft and variants, $N = 3, 4, 5, 7,$ and 9 for DCTOP). Additional implementations can investigate dynamic cluster resizing, allowing processes to join or leave the cluster without impacting the protocol's accuracy and performance. Finally, DCTOP and Raft variant implementations can be integrated with real-world distributed systems and assessed in actual applications. This includes testing the protocols' performance, scalability, and usability in cloud environments, large-scale data centres, or other distributed systems.

7.4.3 Process Crashes Evaluation

As previously stated, Raft variants were designed with most of the traditional Raft assumptions. One of the assumptions is that Raft needs $2f + 1$ nodes to be operational, where f is the number of process failures. So, for $N = 3$ cluster, one failure can be tolerated. Further research could be done to investigate the performance of Raft variants when process crashes occur. Thus, when the Raft variant's protocol is being executed, one can simulate individual process crashes and failures by suddenly stopping a process to simulate a crash. This simulation may proffer more positive and negative insight into the performances of RaftBf and RaftCh.

As noted in Section 3.2.1 about DCTOP, relaxing the assumption of an imperfect failure detector can be a worthwhile area to investigate in future research. We can improve the DCTOP protocol's fault tolerance capabilities by removing assumption AS1 while keeping the perfect failure detector assumption made in LCR. Specifically, when a process crashes, the process connected to it can reliably acknowledge the failure, enabling more accurate failure detection and recovery operations. This assumption relaxation offers more opportunities to investigate its effects on system performance, fault tolerance, and recovery mechanisms. It can entail adjusting the current protocols to include new features for perfect failure detection, recovery, and data consistency maintenance. This can help DCTOP become more dependable and resilient in real-world situations where process crashes occur more frequently or erratically.

7.4.4 Request Batching

Request batching is a mechanism clients use to combine many individual requests into a single, more significant request before delivering them to the distributed cluster. It is often a client-side optimisation technique used to increase the effectiveness of client communications within a distributed cluster. Clients send unique requests to change the distributed system's state using the Raft variants. These requests are then evaluated by the Raft variant's processes using a log replication process. The replicated log only contains one record for each client request. However, a non-trivial open research area will allow clients to send requests in batches instead of individual requests from clients to the Raft variant's cluster. The performance study of such research could make a meaningful research contribution.

References

- [1] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quema. (2006). High Throughput Total Order Broadcast for Cluster Environments. *In: International Conference on Dependable Systems and Networks (DNS'06)*, 2006, pp. 549-557, doi: 10.1109/DSN.2006.37.
- [2] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. (2010). Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Transactions on Computer Systems*, Vol. 28(2), July 2010, pp. 1-32, .<https://doi.org/10.1145/1813654.1813656>.
- [3] D. Ongaro, and J. Ousterhout. (2014). In Search of an Understandable Consensus Algorithm (extended version). <http://ramcloud.stanford.edu/Raft.pdf>. (Accessed online on 10/04/2020).
- [4] F. B. Schneider. (1990). The State Machine Approach: A Tutorial. *In: Lecture Notes in Computer Science : A Workshop on Fault-Tolerant Distributed Computing*, pp. 18-41, 1990, <https://doi.org/10.1007/Bfb0042323>.
- [5] F. B. Schneider. (1990). Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Vol. 22(4), Dec. 1990, pp. 299-319, <https://doi.org/10.1145/98163.98167>.
- [6] L. Lamport. (1978). The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks (1976)*, Vol. 2(2), May 1978, pp. 95-114, [https://doi.org/10.1016/0376-5075\(78\)90045-4](https://doi.org/10.1016/0376-5075(78)90045-4).
- [7] L. Lamport. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*. Vol. 21(7), July 1978, pp. 558-565, <https://doi.org/10.1145/359545.359563>.
- [8] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. (1978). SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *In: Proceedings of the IEEE*, Vol. 66 (10), Oct. 1978, pp. 1240-1255, doi: 10.1109/PROC.1978.11114.
- [9] A. S. Tanenbaum, and M. V. Steen. (2007). Distributed Systems Principles and Paradigms. 2nd Edition, ISBN: 0-13-239227-5, 2007, pp. 140-152, Pearson Prentice Hall, Upper Saddle River, NJ 07458.
- [10] E. Yildirim, M. Balman, and T. Kosar. (2011). 'Data-Aware Distributed Computing'. In T. Kosar (ed.), *Data Intensive Distributed Computing: Challenges and Solutions for*

- Large-scale Information Management*. 1st Edition, ISBN:978-1-61520-971-2, July 2011, pp. 1-27, Information Science Reference-Imprint of: IGI Publishing, Hershey, PA.
- [11] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. (2012). Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *In: Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'12)*, June 2012, pp. 10.
 - [12] J. Dean, and S. Ghemawat. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, Vol. 51(1), January 2008, pp. 107-113, <https://doi.org/10.1145/1327452.1327492>.
 - [13] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. (2008). Newtop: A Fault-Tolerant Group Communication Protocol. *In: Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995, pp. 296-306, doi: 10.1109/ICDCS.1995.500032.
 - [14] R. Guerraoui, and A. Schiper. (1996). Fault-Tolerance by Replication in Distributed Sstems. *In: International Conference on Reliable Software Technologies-Ada-Europe'96*, June 1996, pp. 38-57, <https://doi.org/10.1007/BFb0013477>.
 - [15] L. Lamport, and N. Lynch. (1990). Distributed Computing: Models and Methods. *In: Formal Models and Semantics, A Volume in Handbook of Theoretical Computer Science*, 1990, pp. 1157-1199, <https://doi.org/10.1016/B978-0-444-88074-1.50023-8>.
 - [16] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. (2000). Replication of Data. In A. K. Elmagarmid (ed.), *Replication Techniques in Distributed Systems*, ISBN: 0-7923-9800-9, 2000, pp. 13-58, Kluwer Academic Publishers, USA.
 - [17] M. Van Steen, and A. S. Tanenbaum. (2017). Distributed Systems. 3rd Edition. ISBN: 1543057381, 9781543057386, 2017, pp. 297-316, CreateSpace Independent Publishing Platform, USA.
 - [18] X. Défago, A. Schiper, and P. Urbán. (2004). Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys (CSUR)*, Vol. 36(4), December 2004, pp. 372–421. <https://doi.org/10.1145/1041680.1041682>.
 - [19] R. Elmasri, and S. Navathe. (2000). Fundamentals of Database System. 4th Edition. ISBN: 0321122267, 2000, pp. 803-830, Addison-Wesley, Boston.
 - [20] E. Sakic, and W. Kellerer. (2018). Response Time and Availability Study of RAFT Consensus in Distributed SDN Control Plane. *IEEE Transactions on Network and Service Management*, Vol. 15(1), March 2018, pp. 304-318. <https://doi.org/10.1109/TNSM.2017.2775061>.
 - [21] F. Cristian. (1991). Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, Vol. 34(2), February 1991, pp. 56-78, <https://doi.org/10.1145/102792.102801>.

- [22] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. (2016). XFT: Practical Fault Tolerance Beyond Crashes. *In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. Nov. 2016, pp. 485–500.
- [23] M. Barborak, A. Dahbura, and M. Malek. (1993). The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, Vol. 25(2), June 1993, pp.171-220, <https://doi.org/10.1145/152610.152612>.
- [24] R. Guerraoui, and A. Schiper. (1997). Consensus: The Big Misunderstanding [Distributed Fault Tolerant Systems]. *In: Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997, pp. 183-188, doi: 10.1109/FTDCS.1997.644722.
- [25] E. W. Vollset and P. D. Ezhilchelvan. (2005). Design and Performance-study of Crash-Tolerant Protocols for Broadcasting and Reaching Consensus in MANETs. *In: 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, 2005, pp. 166-175, doi: 10.1109/RELDIS.2005.15.
- [26] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. (2012). Practical Hardening of Crash-Tolerant Systems. *In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC 12*, June 2012, pp. 453-466, <https://dl.acm.org/doi/proceedings/10.5555/2342821>.
- [27] A. Choudhury, G. Garimella, A. Patra, D. Ravi, and P. Sarkar. (2018). Crash-Tolerant Consensus in Directed Graph Revisited. *In: 25th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2018)*, June 2018, pp. 55-71, Cham. https://doi.org/10.1007/978-3-030-01325-7_10.
- [28] M. Pease, R. Shostak, and L. Lamport. (1980). Reaching Agreement in the Presence of Faults. *Journal of the ACM (JACM)*, Vol. 27(2), April 1980, pp. 228–234. <https://doi.org/10.1145/322186.322188>.
- [29] R. D. Schlichting, and F. B. Schneider. (1983). Fail-stop Processors: An approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, Vol. 1(3), August 1983, pp. 222–238, <https://doi.org/10.1145/357369.357371>.
- [30] F. B. Schneider. (1984). Byzantine Generals in Action: Implementing Fail-stop Processors. *ACM Transactions on Computer Systems (TOCS)*, Vol. 2(2), May 1984, pp. 145–154, <https://doi.org/10.1145/190.357399>.
- [31] L. Gong, and X. Qian. (1995). Fail-stop Protocols: An Approach to Designing Secure Protocols. *In: DCCA-5, Proceedings of the 5th International Working Conference on Dependable Computing for Critical Applications*, 1995, pp. 44-55.
- [32] S. Chandra, and P. M. Chen. (1998). How fail-stop are faulty programs?. Digest of Papers. *In: Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, 1998, pp.240-249, doi: 10.1109/FTCS.1998.689475.
- [33] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. (2000). Understanding Replication in Databases and Distributed Systems. *In: Proceedings of*

- the 20th IEEE International Conference on Distributed Computing Systems*, 2000, pp. 464-474, doi: 10.1109/ICDCS.2000.840959.
- [34] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. (1996). Replication of Messages. In A. K. Elmagarmid (ed.), *Replication Techniques in Distributed Systems*, 1996, ISBN 0-306-47796-3, pp. 79-80, Kluwer Academic Publishers, USA .
 - [35] L. Lamport. (2019). The Part-Time Parliament. *Concurrency: the Works of Leslie Lamport*, Association for Computing Machinery, October 2019, pp. 277–317. <https://doi.org/10.1145/3335772.3335939>.
 - [36] K. Birman, and T. Joseph. (1987). Exploiting virtual synchrony in distributed systems. *ACM SIGOPS Operating System Reviews*, Vol. 21(5), Nov. 1987, pp. 123–138. <https://doi.org/10.1145/41457.37515>.
 - [37] O. Rahneva, and N. Pavlov. (2021). Distributed Systems and Applications in Learning. ISBN: 978-619-7663-06-8, 2021, pp. 13-55, Plovdiv University Press, Plovdiv.
 - [38] T. D. Chandra, R. Griesemer, and J. Redstone. (2007). Paxos Made Live: An Engineering Perspective. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, August 2007, pp. 398–407. <https://doi.org/10.1145/1281100.1281103>.
 - [39] L. Lamport. (2001). Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)* Vol. 32(4), December 2001, pp.51-58.
 - [40] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. (2010). ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, June 2010, pp. 11.
 - [41] M. Burrows. (2006). The Chubby Lock Service for Loosely-Coupled Distributed Systems. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Nov. 2006, pp. 335–350.
 - [42] J. Pu, M. Gao, and H. Qu. (2021). SimpleChubby: A Simple Distributed Lock Service. [https://www.scs.stanford.edu/14au-cs244b/labs/projects/pu_gao_qu.pdf]. (Accessed online on 26/08/2021).
 - [43] D. Ongaro, and J. Ousterhout. (2014). In Search of an Understandable Consensus Algorithm. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, June 2014, pp. 305–320.
 - [44] R. Friedman, and R. Van Renesse. (1997). Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In: *Proceeding of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997, pp. 233-242, doi: 10.1109/HPDC.1997.626423.
 - [45] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. (1995). The Totem Single-ring Ordering and Membership Protocol. *ACM Transactions on*

- Computer Systems (TOCS)*, Vol. 13(4), Nov. 1995, pp. 311-342, <https://doi.org/10.1145/210223.210224>.
- [46] B. A. Forouzan. (2007). *Data Communications and Networking*. 4th Edition. ISBN: 9780073250328, 2007, pp. 7- 13, McGraw Hill Higher Education, New York.
 - [47] W. Stallings. (2007). *Data and Computer Communications*. 8th Edition. ISBN: 0-13-243310-9, 2007, pp. 446-473, Pearson Prentice Hall, Upper Saddle River, NJ 07458.
 - [48] G. F. Coulouris, J. Dollimore, and T. Kindberg. (2012). *Distributed Systems: Concepts and Design*. 5th Edition. ISBN: 9780132143011, 2012, pp. 675-718, Pearson Education, Inc., Boston.
 - [49] Canalys. (2021). *Global Cloud Services Market Q2 2021*. <https://canalys.com/newsroom/global-cloud-services-q2-2021>, (Accessed online on 01/10/2022).
 - [50] E. Dubrova. (2013). *Fault-Tolerant Design: An Introduction*. ISBN: 1461421136, 9781461421139, 2013, pp. 1-2, Springer Science & Business Media.
 - [51] *Getting Started with Replication Berkeley DB Applications*. (2015). https://docs.oracle.com/cd/E17076_03/html/gsg_db_rep/C/index.html, (Accessed online on 02/10/2022).
 - [52] J. Vučković. (2006). *Modular Algorithms for Component Replication* (PhD. Thesis). *Technical Report UBLCS-2006-13*, March 2006, pp. 3-4.
 - [53] F. B. Schneider. (1993). *Replication Management using the State-machine Approach*. *Distributed Systems*. 2nd Edition. ISBN: 0201624273, pp. 169-197, ACM Press/Addison-Wesley Publishing Co., USA.
 - [54] F. Pedone, R. Guerraoui, and A. Schiper. (2003). The Database State Machine Approach. *Distributed and Parallel Databases*, Vol. 14(1), July 2003, pp. 71-98, <https://doi.org/10.1023/A:1022887812188>.
 - [55] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. (1993). The Primary-backup Approach. *Distributed Systems*. 2nd Edition. ISBN: 0201624273, pp. 199-216, ACM Press/Addison-Wesley Publishing Co., USA.
 - [56] M. P. Herlihy, and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 12(3), July 1990, pp. 463-492. <https://doi.org/10.1145/78969.78972>.
 - [57] T. D. Chandra, and S. Toueg. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, Vol. 43(2), March 1996, pp. 225-267, <https://doi.org/10.1145/226643.226647>.
 - [58] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. (2002). A Realistic Look at Failure Detectors. *In: Proceedings of the International Conference on Dependable Systems and Networks*, June 2002, pp. 345-353, doi: 10.1109/DSN.2002.1028919.

- [59] S. Sastry, and S. M. Pike. (2007). Eventually Perfect Failure Detectors using ADD Channels. *In: Proceedings of the 5th international conference on Parallel and Distributed Processing and Application*, Aug. 2007, pp. 483-496, <https://dl.acm.org/doi/proceedings/10.5555/2395970>.
- [60] E. Brewer. (2010). A Certain Freedom: Thoughts on the CAP Theorem. *In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC '10)*, July 2010, pp. 335. <https://doi.org/10.1145/1835698.1835701>.
- [61] S. Bagui, and L. T. Nguyen. (2015). Database Sharding: To Provide Fault Tolerance and Scalability of Big Data on the Cloud. *International Journal of Cloud Applications and Computing (IJCAC)*, Vol. 5(2), April 2015, pp. 36-52, <https://doi.org/10.4018/IJCAC.2015040103>.
- [62] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. (2001). Partial Replication in the Database State Machine. *In: Proceedings IEEE International Symposium on Network Computing and Applications*, Oct. 2001, pp. 298-309, doi: 10.1109/NCA.2001.962546.
- [63] R. Emerson. (2016). Scalable Coordination of Distributed In-memory Transactions. PhD Thesis, <http://hdl.handle.net/10443/3336>, 2016, pp. 8-36.
- [64] M. Hosseini, D. T. Ahmed, S. Shirmo.hammadi, and N. D. Georganas. (2007). A Survey of Application-layer Multicast Protocols. *IEEE Communications Surveys & Tutorials*, Vol. 9(3), Sept. 2007, pp. 58-74, doi: 10.1109/COMST.2007.4317616.
- [65] B. N. Levine, and J. J. Garcia-Luna-Aceves. (1998). A Comparison of Reliable Multicast Protocols. *Multimedia Systems*, Vol. 6(5), Sept. 1998, pp. 334-348, <https://doi.org/10.1007/s005300050097>.
- [66] D. Malkhi, and M. Reiter. (1996). A High-Throughput Secure Reliable Multicast Protocol. *In: Proceedings 9th IEEE Computer Security Foundations Workshop*, August 1996, pp. 9-17, doi: 10.1109/CSFW.1996.503686.
- [67] P. Verissimo, L. Rodrigues, M. Baptista. (1989). AMp: A Highly Parallel Atomic Multicast Protocol. *In: Symposium proceedings on Communications Architectures & Protocols*, August 1989, pp. 83-93. <https://doi.org/10.1145/75246.75256>.
- [68] M. F. Kaashoek, and A. S. Tanenbaum. (1996). An Evaluation of the Amoeba Group Communication System. *In: Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996, pp. 436-447, doi: 10.1109/ICDCS.1996.507992.
- [69] S. Armstrong, A. Freier, and K. Marzullo. (2022). Multicast Transport Protocol. <https://www.rfc-editor.org/rfc/rfc1301>, (Accessed online on 16/05/2022).
- [70] R. Carr. (1985). The Tandem Global Update Protocol. *Tandem Systems Review*, Vol. 1(2), 1985, pp. 74-85.

- [71] K. P. Birman, and R. V. Renesse. (1993). Reliable Distributed Computing with the Isis Toolkit, ISBN: 978-0-8186-5342-1, 1993, pp. 416, IEEE Computer Society Press, Washington, DC, USA.
- [72] H. Garcia-Molina, and A. Spauter. (1991). Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems (TOCS)*, Vol. 9(3), August 1991, pp. 242–271. <https://doi.org/10.1145/128738.128741>.
- [73] U. Wilhelm, and A. Schiper. (1995). A Hierarchy of Totally Ordered Multicasts. *In: Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995, pp. 106-115, doi: 10.1109/RELDIS.1995.526218.
- [74] R. Baldoni, S. Cimmino, and C. Marchetti. (2006). A Classification of Total Order Specifications and its Application to Fixed Sequencer-based Implementations. *Journal of Parallel and Distributed Computing*, Vol. 66(1), January 2006, pp. 108-127, <https://doi.org/10.1016/j.jpdc.2005.06.021>.
- [75] S. Navaratnam, S. Chanson and G. Neufeld. (1988). Reliable Group Communication in Distributed Systems. *In: Proceedings of the 8th International Conference on Distributed System*, June 1988, pp. 439-446, doi: 10.1109/DCS.1988.12546.
- [76] K. P. Birman, and T. Clark. (1994). Performance of the Isis Distributed Computing Toolkit, *Technical Report, TR-94-1432*, pp. 1-31, June 1994.
- [77] J.-M. Chang, and N. F. Maxemchuk. (1984). Reliable Broadcast Protocols. *ACM Transactions on Computer Systems (TOCS)*, Vol. 2(3), August 1984, pp. 251-273, DOI:10.1145/989.357400.
- [78] F. Cristian, S. Mishra, and G. Alvarez. (1997). High-performance Asynchronous Atomic Broadcast. *Distributed Systems Engineering*, Vol. 4(2), June 1997, pp. 109, doi: 10.1088/0967-1846/4/2/005
- [79] J. Kim, and C. Kim,. (1997). A Total Ordering Protocol using a Dynamic Token-passing Scheme. *Distributed Systems Engineering*, Vol. 4(2), June 1997, pp. 87, doi: 10.1088/0967-1846/4/2/003.
- [80] B. Whetten, T. Montgomery, and S. Kaplan. (1994). A High Performance Totally Ordered Multicast Protocol. *In: Proceedings of the International Workshop on Theory and Practice in Distributed System*, Sept. 1994, pp. 33-57. https://doi.org/10.1007/3-540-60042-6_3.
- [81] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. (1989). Preserving and using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, Vol. 7(3), Aug. 1989, pp. 217–246. <https://doi.org/10.1145/65000.65001>.
- [82] L. M. Malhis, W. H. Sanders, and R. D. Schlichting. (1996). Numerical Performability Evaluation of a Group Multicast Protocol. *Distributed Systems Engineering*, Vol. 3(1), March 1996, pp. 39, doi:10.1088/0967-1846/3/1/006.

- [83] T. P. Ng. (1991). Ordered Broadcasts for Large Applications. *In: Proceedings of the Tenth Symposium on Reliable Distributed Systems*, Sept./Oct. 1991, pp. 188-197, doi: 10.1109/RELDIS.1991.145423.
- [84] L. E. Moser, P. Melliar-Smith, and V. Agrawala. (1993). Asynchronous Fault-Tolerant Total Ordering Algorithms. *SIAM Journal on Computing*, Vol. 22(4), Aug. 1993, pp. 727-750, <https://doi.org/10.1137/0222048>.
- [85] K. P. Birman, and T. A. Joseph. (1987). Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS)*, Vol. 5(1), Feb. 1987, pp. 47-76. <https://doi.org/10.1145/7351.7478>.
- [86] S.-W. Luan, and V. D. Gligor. (1990). A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1(3), July 1990, pp. 271-285, <https://doi.org/10.1109/71.80156>.
- [87] U. Fritzke, P. Ingels, A. Mostefaoui, and M. Raynal. (2001). Consensus-based Fault-Tolerant Total Order Multicast. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12(2), Feb. 2001, pp. 147-156, <https://doi.org/10.1109/71.910870>.
- [88] B. Chaurasia, and A. Verma. (2020). A Comprehensive Study on Failure Detectors of Distributed Systems. *Journal of Scientific Research*, Vol. 64(2), July 2020, pp. 250-260, DOI: 10.37398/JSR.2020.640235.
- [89] R. Palmieri. (2016). Leaderless Consensus: The State of the Art. 2016 *In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1307-1310, doi: 10.1109/IPDPSW.2016.192.
- [90] I. Moraru, D. G. Andersen, and M. Kaminsky. (2022). Egalitarian Parliament. <https://www.usenix.org/system/files/nsdi13-paper14.pdf>, (Accessed online on 10/09/2022).
- [91] I. Moraru, D. G. Andersen, and M. Kaminsky. (2013). There is More Consensus in Egalitarian Parliaments. *In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, November 2013, pp. 358-372. <https://doi.org/10.1145/2517349.2517350>.
- [92] Y. Mao, F. P. Junqueira, and K. Marzullo. (2008). Mencius: Building Efficient Replicated State Machines for WANs. *In: Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*, Dec 2008, pp. 369-384, <https://dl.acm.org/doi/proceedings/10.5555/1855741>.
- [93] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. (2011). Megastore: Providing Scalable, Highly Available Storage for Interactive Services. *In: 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, January 2011, pp. 223-234.
- [94] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, and P. Hochschild. (2013). Spanner: Google's Globally Distributed

- Database. *ACM Transactions on Computer Systems (TOCS)*, Vol. 31(3), August 2013, pp. 22, <https://doi.org/10.1145/2491245>.
- [95] F. Junqueira, and B. Reed. (2013). ZooKeeper: Distributed Process Coordination. ISBN: 9781449361303, 2013, pp. 155- 176, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.
 - [96] F. Cristian, and S. Mishra. (1995). The Pinwheel Asynchronous Atomic Broadcast Protocols. *In: Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, April 1995, pp. 215-221, doi: 10.1109/ISADS.1995.398975.
 - [97] R. Ekwall, A. Schiper, and P. Urbán. (2004). Token-based Atomic Broadcast using Unreliable Failure Detectors. *In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, October 2004*, pp. 52-65, <https://dl.acm.org/doi/proceedings/10.5555/1032662>.
 - [98] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. (1996). Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, Vol. 39(4), April 1996, pp. 54-63. <https://doi.org/10.1145/227210.227226>.
 - [99] Y. Amir, L. E. Moser, P. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. (1993). Fast message ordering and membership using a logical token-passing ring. *In: Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993, pp. 551-560, doi: 10.1109/ICDCS.1993.287668.
 - [100] M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep. (1989). Group Communication in Amoeba and its Applications. *Distributed Systems Engineering*, Vol. 1(1), Sept. 1993, pp. 48, doi: 10.1088/0967-1846/1/1/006.
 - [101] M. F. Kaashoek, A. S. Tanenbaum, and S. F. Hummel. (1989). An Efficient Reliable Broadcast Protocol. *ACM SIGOPS Operating Systems Review*, Vol. 23(4), Oct. 1989, pp. 5-19. <https://doi.org/10.1145/70730.70732>.
 - [102] S. Mishra, L. L. Peterson, and R. D. Schlichting. (1993). Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering*, Vol. 1(2), pp. 87, 1993, doi:10.1088/0967-1846/1/2/004.
 - [103] G. V. Chockler, I. Keidar, and R. Vitenberg. (2001). Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, Vol. 33(4), Dec. 2001, pp. 427-469. <https://doi.org/10.1145/503112.503113>.
 - [104] R. De Prisco, B. Lampson, and N. Lynch. (1997). Revisiting the Paxos Algorithm. *In: Proceedings of the 11th International Workshop on Distributed Algorithms*, September 1997, pp. 111- 125.
 - [105] B. W. Lampson. (1996). How to Build a Highly Available System using Consensus. *In: Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG '96)*, Oct. 1996, pp. 1-17, <https://dl.acm.org/doi/proceedings/10.5555/645953>.

- [106] E. K. Lee, and C. A. Thekkath. (1996). Petal: Distributed Virtual Disks. *ACM SIGPLAN Notices*, Vol. 31(9), Sept. 1996, pp. 84–92. <https://doi.org/10.1145/237090.237157>.
- [107] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. (2004). Boxwood: Abstractions as the Foundation for Storage Infrastructure. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, Dec. 2004, pp. 8, <https://dl.acm.org/doi/proceedings/10.5555/1251254>
- [108] L. Lamport. (2001). Paxos Made simple. *ACM SIGACT News Distributed Computing Column*, Vol. 32(4), December 2001, pp. 51-58.
- [109] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. (2011). Paxos Replicated State Machines as the Basis of a High-performance Data Store. In: *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11)*, March 2011, pp. 141–154.
- [110] L. Lamport. (2005). Generalized Consensus and Paxos. Microsoft Research Technical Report MSR-TR-2005-33, (Accessed online on 5/06/2021).
- [111] L. Lamport. (2006). Fast Paxos. *Distributed Computing*, Vol. 19(2), October 2006, pp. 79–103. <https://doi.org/10.1007/s00446-006-0005-x>.
- [112] L. J. Camargos, R. M. Schmidt, and F. Pedone. (2007). Multicoordinated Paxos. In: *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, Aug. 2007, pp. 316–317, <https://doi.org/10.1145/1281100.1281150>.
- [113] B. Lampsom. (2001). The ABCD's of Paxos. In: *Proceedings of the Twentieth Annual ACM symposium on Principles of Distributed Computing (PODC '01)*, Aug. 2001, pp. 13, <https://doi.org/10.1145/383962.383969>.
- [114] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. (2010). Ring Paxos: A high-throughput atomic broadcast protocol. In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2010, pp. 527-536, doi: 10.1109/DSN.2010.5544272.
- [115] P. Fouto, N. Preguiça, and J. Leitão. (2022). High Throughput Replication with Integrated Membership Management. In: *USENIX Annual Technical Conference (USENIX ATC 22)*, July 2022, pp. 575-592.
- [116] R. Van Renesse, and F. B. Schneider. (2004). Chain Replication for Supporting High Throughput and Availability. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, Dec. 2004, pp. 7.
- [117] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany. (2020). Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020, pp. 351–368.

- [118] S. Ghemawat, H. Gobioff, and S.-T. Leung. (2003). The Google File System. *In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, Oct. 2003, pp. 29–43, <https://doi.org/10.1145/945445.945450>.
- [119] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, Vol. 26(2), June 2008, pp. 26, <https://doi.org/10.1145/1365815.1365816>.
- [120] A. Medeiros. (2012). ZooKeeper’s Atomic Broadcast Protocol: Theory and Practice, <http://www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf>, (Accessed online on 22/04/2022)
- [121] S. Haloi. (2015). Apache Zookeeper Essentials. ISBN 978-1-78439-132-4, 2015, pp. 43-57, Packt Publishing Ltd. Livery Place, 35 Livery Street Birmingham B3 2PB, UK.
- [122] A. ZooKeeper. (2022). The Apache Software Foundation. URL: <https://zookeeper.apache.org/online>, (Accessed online on 10/01/2022).
- [123] V. Hadzilacos, and S. Toueg. (1994). A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. *Technical Report*, pp. 1-84, May 1994.
- [124] G. T. Wu, and A. J. Bernstein. (1984). Efficient Solutions to the Replicated Log and Dictionary Problems. *In: Proceedings of the Third annual ACM symposium on Principles of distributed computing (PODC '84)*, Aug. 1984, pp. 233–242. <https://doi.org/10.1145/800222.806750>
- [125] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein. (2015). Building a Replicated Logging System with Apache Kafka. *In: Proceedings of the VLDB Endowment*, Vol. 8(12), August 2015, pp. 654–1655. <https://doi.org/10.14778/2824032.2824063>.
- [126] S. Guo, R. Dhamankar, and L. Stewart. (2017). DistributedLog: A High Performance Replicated Log Service. *In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, May 2017, pp. 1183-1194, doi: 10.1109/ICDE.2017.163.
- [127] J.-S. Ahn, W.-H. Kang, K. Ren, G. Zhang, and S. Ben-Romdhane. (2019). Designing an Efficient Replicated Log Store with Consensus Protocol. *In: Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'19)*, July 2019, pp. 8.
- [128] B. Liskov, and J. Cowling. (2012). Viewstamped Replication Revisited. MIT Technical Report MIT-CSAIL-TR-2012-021, <https://pmg.csail.mit.edu/papers/vr-revisited.pdf>, (Accessed online on 07/04/2020).
- [129] B. M. Oki, and B. H. Liskov. (1988). Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. *In: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*, Jan. 1988, pp. 8–17, <https://doi.org/10.1145/62546.62549>.

- [130] S. Kasampalis. (2010). Copy on Write Based File Systems Performance Analysis and Implementation, <https://sakisk.me/files/copy-on-write-based-file-systems.pdf>, (Accessed online on 05/03/2021).
- [131] F. P. Junqueira, B. C. Reed, and M. Serafini. (2011). Zab: High-performance Broadcast for Primary-backup Systems. *In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, June 2011, pp. 245-256, doi: 10.1109/DSN.2011.5958223.
- [132] J. Banks, J. S. CARSON II, and L. Barry. (2005). Discrete-event System Simulation, 4th Edition. ISBN: 978-0131293427, 2005, pp. 60-81, Pearson Prentice – Hall International Series in Industrial and Systems Engineering, USA.
- [133] T. Cooper. (2020). Performance Modelling of Distributed Stream Processing Topologies. PhD Thesis, 2020.
- [134] S. Robinson. (2014). Simulation: The Practice of Model Development and Use, 2nd Edition. ISBN: 9781137328021, 2014, pp. 15-24, Bloomsbury Publishing, London.
- [135] G. Hohpe, and B. Woolf. (2003). Enterprise integration patterns: Designing, Building, and Deploying Messaging Solutions. 1st Edition. ISBN-10: 9780321200686, pp. 108-135, Addison-Wesley Professional, Boston.
- [136] J. Kreps, N. Narkhede, and J. Rao. (2011). Kafka: A Distributed Messaging System for Log Processing. *In: Proceedings of the NetDB*, June 2011, pp. 1-7.
- [137] B. Christudas, and B. Christudas. (2019). Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud. ISBN: 978-1-4842-4500-2, pp. 105-145, Apress Berkeley, CA.
- [138] C. Aranha, C. Both, B. Dearfield, C. L. Elkins, A. Ross, J. Squibb, and M. Taylor. (2013). IBM WebSphere MQ V7. 1 and V7. 5 Features and Enhancements. <https://www.redbooks.ibm.com/redbooks/pdfs/sg248087.pdf>, (Accessed online on 21/08/2020).
- [139] P. C. Brown. (2011). TIBCO Architecture Fundamentals. ISBN: 9780132762427, pp. 67-72, Addison-Wesley Professional, Boston.
- [140] R. Guerraoui, F. Huc, and A.-M. Kermarrec. (2013). Highly Dynamic Distributed Computing with Byzantine Failures. *In: Proceedings of the 2013 ACM symposium on Principles of Distributed Computing (PODC '13)*, July 2013, pp. 176–183. <https://doi.org/10.1145/2484239.2484263>.
- [141] M. Nasreen, A. Ganesh, and C. Sunitha. (2016). A Study on Byzantine Fault Tolerance Methods in Distributed Networks. *In: Fourth International Conference on Recent Trends in Computer Science and Engineering (ICRTCSE 2016)*, 2016, pp. 50-54, <https://doi.org/10.1016/j.procs.2016.05.125>.
- [142] R. Bloem, N. Braud-Santoni, and S. Jacobs. (2016). Synthesis of Self-Stabilising and Byzantine-Resilient Distributed Systems. *In: International Conference on Computer*

- Aided Verification*, July 2016, pp. 157-176, https://doi.org/10.1007/978-3-319-41528-4_9.
- [143] V. G. Cerf, and R. E. Icahn. (1974). A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, Vol. 22(5), May 1974, pp. 637-648, doi: 10.1109/TCOM.1974.1092259.
 - [144] K. M. Chandy, and L. Lamport. (1985). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems (TOCS)*, Vol. 3(1), Feb. 1985, pp. 63–75. <https://doi.org/10.1145/214451.214456>.
 - [145] O. Bonaventure. (2016). *Computer Networking: Principles, Protocols and Practice*. ISBN-10: 1365185834, 2016, pp. 5-105, Lulu Press Inc., North Carolina.
 - [146] I. Mitran. (1982). *Simulation Techniques for Discrete Event Systems*. ISBN: 0521238854, 1982, pp. 1-194, Cambridge University Press, England.
 - [147] The Raft Consensus Algorithm. <https://raft.github.io/>. (Accessed online on 20/06/2023).
 - [148] I. Research. (2023). Network Latency- Common Causes and Best Solutions. <https://www.ir.com/guides/what-is-network-latency>, (Accessed online on 15/05/2023).
 - [149] R. De Oliveira, and T. Braun. (2006). A Smart TCP Acknowledgement Approach for Multihop Wireless Networks. *IEEE Transactions on Mobile Computing*, Vol. 6(2), Dec. 2006, pp. 192-205, doi: 10.1109/TMC.2007.19.
 - [150] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, and R. Poss. (2020). Cockroachdb: The Resilient Geo-Distributed SQL Database. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, June 2020, pp. 1493–1509, <https://doi.org/10.1145/3318464.3386134>.
 - [151] S. Sivasubramanian. (2012). Amazon dynamoDB: A Seamlessly Scalable Non-Relational Database Service. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*, May 2012, pp. 729–730, <https://doi.org/10.1145/2213836.2213945>.
 - [152] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. (2007). Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, Vol. 41(6), Dec. 2007, pp. 205–220. <https://doi.org/10.1145/1323293.1294281>.
 - [153] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. (2012). S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In: *2012 IEEE 31st Symposium on Reliable Distributed Systems*, Oct. 2012, pp. 111-120, doi: 10.1109/SRDS.2012.66.

Appendix A1 – Simulation Design

