Evaluation of an Efficient Ring-Based Total Order Protocol in a Fairness-Controlled Environment

digital

MDPI

Article

Evaluation of an Efficient Ring-Based Total Order Protocol in a Fairness-Controlled Environment

Agbaeze Ejem [1,*], Cosmas Ifeanyi Nwakanma [2], Ejem Agwu Ejem [3] and Juliet Nnenna Odii [4]

[1] Department of Computing, University of Worcester, City Campus, Castle St, Worcester WR1 3AS, UK
[2] Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506, USA; cosmas.nwakanma@mail.wvu.edu
[3] Department of Logistics and Supply Chain Management, Federal University of Technology Owerri, Owerri PMB 1526, Imo State, Nigeria; ejem.ejem@futo.edu.ng
[4] Department of Computer Science, Federal University of Technology Owerri, Owerri PMB 1526, Imo State, Nigeria; juliet.odii@futo.edu.ng
* Correspondence: a.ejem@worc.ac.uk; Tel.: +44-7879178581

Abstract

Crash-tolerant systems rely on total order protocols to ensure consistent request execution across replicated servers. The Logical Clock and Ring (LCR) protocol employs a ring-based, leaderless design that provides a high throughput but suffers latency inefficiencies under a high message concurrency due to its use of vector clocks and a fixed last-process rule for ordering concurrent messages. This paper presents the Daisy Chain Total Order Protocol (DCTOP), an enhanced version of LCR that integrates Lamport logical clocks for message sequencing and introduces dynamic last-process identification based on sender activity to accelerate message stabilisation and delivery. A modified fairness-control mechanism further balances message distribution among processes. The simulation results show that the DCTOP achieves an over 40% latency reduction compared to LCR while maintaining the same fairness and throughput across various cluster configurations.

Keywords: LCR; total order; fairness control; latency; throughput; performance comparison; simulation

Academic Editor: Yannis Manolopoulos

Received: 26 August 2025
Revised: 6 November 2025
Accepted: 7 November 2025
Published: 20 November 2025

Citation: Ejem, A.; Nwakanma, C.I.; Ejem, E.A.; Odii, J.N. Evaluation of an Efficient Ring-Based Total Order Protocol in a Fairness-Controlled Environment. *Digital* **2025**, *5*, 64. https://doi.org/10.3390/digital5040064

1. Introduction

A crash-tolerant system is designed to continue functioning despite a threshold number of crashes occurring and is crucial to maintaining a high availability of systems [1–4]. Replication techniques have made it possible to create crash-tolerant distributed systems. Through replication, redundant service instances are created on multiple servers, and a given service request is executed on all servers so that, even if some servers crash, the rest will ensure that the client receives the response. The set of replicated servers hosting service instances may also be referred to as a replicated group and simply as a group. A group is a collection of distributed processes in which a member process communicates with other members only by sending messages to the full membership of the group [5]. Typically, a client can send its service to any one of the redundant servers in the group and the server that receives a client request, in turn, disseminates it within the group so all can execute. Thus, different servers can receive client requests in a different order, but, despite this, all servers must process client requests in the same order [6,7]. To accomplish this, a total order mechanism that employs logical clock is utilised to guarantee that replicated servers process client requests or simply messages in the same order [8]. 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. Thus, a total order protocol is a procedure 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. While total order protocols play a critical role in maintaining consistency and system reliability, achieving crash tolerance requires the implementation of additional mechanisms. One such mechanism, as defined in our work, is the crashproofness policy. Specifically, this policy dictates that a message is deemed crashproof and safe for delivery once it has successfully reached at least f+1 operative processes, where f is the maximum tolerated failures in a group.

Research into total-order broadcast and distributed consensus has evolved significantly beyond the traditional ring and token-passing systems, exemplified by the LCR algorithm [9] and the Fixed Sequence Ring (FSR) protocol [10] which employ neighbour-to-neighbour communication to maintain the same fairness and message order under asynchronous conditions. Contemporary consensus frameworks, including Apache Zookeeper [11,12], Chubby [13,14], Paxos [15], View-stamp replication [16], and Raft [17–19], rely on leader-based coordination to achieve strong consistency guarantees. However, this approach often introduces coordination overhead, which can hinder scalability. To address these limitations, protocols such as Fast Paxos [20], S-Paxos [21], and EPaxos [22] have been proposed to reduce decision latency by minimising or eliminating the reliance on a centralised leader, enabling a more decentralised or leaderless consensus execution. In the Byzantine fault-tolerant (BFT) domain, protocols like HotStuff [23] utilise multi-phase authenticated voting schemes, typically incurring a $O(N^2)$ communication complexity in the worst case to maintain agreement in the presence of multiple leader failures. Mir-BFT [24] enhances the throughput by parallelising the leader role, while Dumbo [25] and Dumbo-NG [26] optimise asynchronous consensus to reduce latency and communication overhead. Flutter [27] further improves responsiveness and throughput through pipelined and leaderless message dissemination. Emerging efforts such as Antipaxos [28] and BBCA-Ledger [29] explore new directions in high-throughput consensus: while Antipaxos achieves leaderless parallel agreement through *k-Interactive Consistency (k-IC),* BBCA-Ledger combines a low-latency broadcast path with a DAG-based fallback mechanism to sustain the throughput under network faults. Ring-based broadcast schemes, including Ring Paxos [30] which still internally elects a leader and chain replication [31], achieve message ordering with linear communication overhead by sequentially relaying messages across nodes. BPaxos [32] adopts a modular state machine replication design that scales consensus components independently to remove the leader bottleneck in traditional protocols. While this improves the throughput, it introduces a higher latency and greater implementation complexity. In contrast, the DCTOP achieves a low-latency concurrency through a simpler ring-based, leaderless structure using Lamport timestamping for efficient total order delivery. Building on the foundation established by ring-based broadcast [33] such as LCR, the DCTOP introduces dynamic last-process identification and a Lamport logical clock mechanism to enhance the ordering latency and determinism. Compared to consensus-based frameworks, the DCTOP offers a lightweight, topology-aware ordering service that minimises coordination costs. This makes it particularly well-suited for deployment in cloud and edge environments, where the overhead of full consensus protocols may be prohibitive. As such, the DCTOP represents a complementary alternative that prioritises ordered message dissemination with reduced synchronisation requirements. The aim of this research is to design and evaluate an efficient ring-based total order protocol that improves the message ordering latency and fairness in asynchronous distributed

environments. Specifically, the study seeks to enhance the classical LCR protocol. Despite the progress made by LCR, certain design choices may lead to increased latency. The LCR protocol utilises vector clock where each process, denoted as P_i, maintains its own clock as $VC_i = vc_{k,k=0,...N-1}$. A vector clock is a tool used to establish the order of events within a distributed system which can be likened to an array of integers, with each integer corresponding to a unique process in the ring. 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 within replica buffers or queues [34], and, secondly, it uses a fixed idea of the "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.

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 a 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 the 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. 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.

Prompted by the above potential drawbacks in LCR, a new total order protocol was design with N, $N > 2$, processes arranged in a unidirectional logical ring where N is the number of processes within the server clusters. Messages are assumed to pass among processes in a clockwise direction as shown in Figure 1. If a message originates from P_0, it moves to P_1 until it gets to P_3 which is the last process for P_0. The study aims to achieve the following objectives: (i) Optimise message timestamping with Lamport logical clocks, which uses a single integer to represent message timestamps. This approach is independent of N, unlike the vector timestamping used in LCR, which is dependent on N.

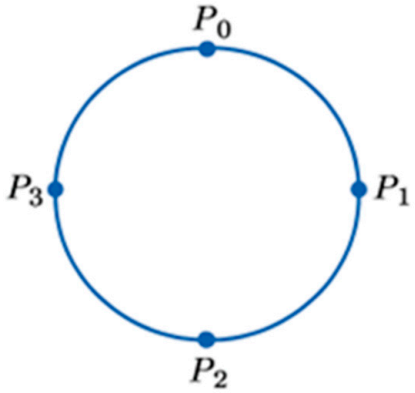


Figure 1. Last process concept.

(ii) Dynamically determine the "Last" Process for ordering concurrent messages. Instead of relying on a globally fixed last process for ordering concurrent messages, as in LCR, this study proposes a dynamically determined last process based on proximity to the sender in the opposite direction of message flow. This adaptive mechanism improves ordering flexibility and enhances system responsiveness under high workloads. (iii) Reduce message delivery latency. This study proposes reducing the value of f to $(N - 1)/2$ to minimise the overall message delivery latency and enhance system efficiency. This contrasts with the LCR approach, where f is set to $N - 1$. 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 the message delivery latency and impacting system performance. The goal of this study was accomplished using three methods: First, we considered a set of restricted crash assumptions: each process crashes independently of others and, at most, f processes involved in group communication can crash. Hence, the number of crashes that can occur in an N process cluster is bounded by $f = \left\lfloor \frac{N-1}{2} \right\rfloor$, 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 [17]. As a result, at least two processes are always operational and connected. Thus, an Eventually Perfect Failure Detector (♦P) was assumed in this study's system model, operating under the assumption that N = 2f + 1 nodes are required to tolerate up to f crash failures. This approach enables the new protocol to manage temporary inaccuracies, such as false suspicions, by waiting for a quorum of at least f + 1 nodes before making decisions. This ensures that the system does not advance based on incorrect failure detections. Secondly, the last process of each sender is designated to determine the stability of messages. It then communicates this stability by sending an acknowledgement message to other processes. When the last process of the sender receives the message, it knows that all the logical clocks within the system have exceeded the timestamp of the message (stable property). Then, all the received messages whose timestamp is less than the last process logical clock can then be totally ordered. In addition, a new concept of "deliverability requirements" was introduced to guarantee the delivery of crashproof and stable messages in total order. A message is crashproof if the number of messages hops $\geq f + 1$; that is, a message must make at least $f + 1$ number of hops before it is termed crashproof. Thus, the delivery of a message is subject to meeting both deliverability and order requirements. As a result of enhancements made in this regard, a new leaderless ring-based total order protocol was designed, known as the Daisy Chain Total Order Protocol (DCTOP). Thirdly, fairness is defined as the condition where every process P_i has an equal chance of having its sent messages eventually delivered by all processes within the cluster. Every process ensures messages from the predecessor 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.

1.1. Contributions

The contributions of this paper can be summarised as follows:

(i) Protocol-Level Innovations Within a Ring-Based Framework: This study introduces the DCTOP, a novel improvement to the classical LCR protocol while retaining its ring-based design. It introduces the following:

- a A Logical Clock is used for message timestamping which achieves efficient concurrent message ordering, reducing latency and improving fairness.
- b Dynamic Last-Process Identification is used to replace LCR's globally fixed last process assumption, accelerating message stabilisation and delivery.

(ii) Relaxed Failure Assumption: The DCTOP reduces N = f + 1 to N = 2f + 1, enabling faster message delivery with fewer failures. Importantly, this relaxation remains consistent with the theoretical lower bounds for consensus fault tolerance, which require at least N $\geq$ 2f + 1 replicas to guarantee safety in asynchronous distributed systems [35,36]. Thus, the DCTOP improves latency performance without violating the fundamental resilience limits established by the classical consensus theory.

(iii) Foundation for Real-World Deployment: While simulations excluded failures and large-scale setups, ongoing work involves a cloud-based, fault-tolerant implementation to validate the DCTOP under practical conditions.

1.2. Practical Integration and Application Scenarios of DCTOP

The DCTOP can be practically deployed as a lightweight ordering layer within modern distributed infrastructures. In a cloud or edge cluster, each process Pi in the DCTOP model corresponds to an individual service instance or virtual node, connected through asynchronous communication channels such as message-oriented middleware. The daisy-chain configuration forms a logical ring that can be established dynamically at runtime, allowing the DCTOP to operate independently of any centralised coordinator. Because each process communicates only with its immediate neighbours, the communication overhead is linear O(N) rather than quadratic, making the DCTOP suitable for large-scale, latency-sensitive environments. The DCTOP's logical clock mechanism ensures global ordering under varying network delays, providing consistent message sequencing even in the presence of asynchronous message arrivals. The DCTOP attains total order through deterministic topological progression and local timestamp stabilisation. This property makes it particularly useful as a complementary component in distributed logging, event streaming, and replicated state machine frameworks where high-frequency, ordered message dissemination is required without full consensus overhead. As a use case, the DCTOP can be integrated into a cloud-based logging replication service where each node in the ring represents a log replica [37,38]. Incoming events are timestamped, propagated clockwise, and stabilised through the ring, ensuring a consistent total order across all replicas. This approach offers an improved throughput and reduced coordination delay compared to leader-based ordering schemes, demonstrating the DCTOP's applicability to real-world, fault-tolerant distributed systems.

This paper is structured as follows: Section 2 presents the system model, while Section 3 outlines the design objectives and rationale for the DCTOP. Section 4 details the fairness control primitives. Section 5 provides performance comparisons of the DCTOP, LCR, and Raft in terms of the latency and throughput under crash-free and high-workload conditions. Finally, Section 6 presents the paper's conclusion.

2. DCTOP System Model

The system is modelled as a group of N processes, denoted by $\Pi = \{P_0, P_1, P_2, \ldots, P_{N-1}\}$ organised in a circular (ring-based) structure under an asynchronous communication framework. The message transmission intervals follow an exponential distribution, with no constraints on communication delays. The model assumes FIFO message delivery, ensuring that messages are received in the order sent. Each process P_i can transmit messages only to its clockwise neighbour (CN_i) and receive from its anticlockwise neighbour (ACN_i). Formally,

- $CN_i = P_0 \; if \; i = N-1; CN_i = P_0 \; if \; i = N-1;$
- $ACN_i = P_{i-1}, \; or \; ACN_i = P_{N-1} \; if \; i = 0.$

Messages, therefore, propagate exclusively in the clockwise direction. Each process maintains a stability clock (SC_i), representing the largest stable timestamp known to P_i. When a message m becomes stable, SC_i is updated as $SC_i = \max(SC_i, m\_ts)$.

The number of hops between any two processes from P_i to P_j is denoted as $Hops_{i,j}$:

(i) $Hops_{i,j} = 0 \; if \; i = j$;
(ii) $Hops_{i,j} = j - i \; if \; j > i$;
(iii) $Hops_{i,j} = j + N - i \; if \; j < i$.

3. DCTOP—Daisy Chain Total Order Protocol

The DCTOP system employs a group of interconnected process replicas, with a group size of N, where N is an odd integer, $N \geq 3$ and, at most, 9, to provide replicated services. The main goals of the system design are threefold:

(a) First, to improve the latency of LCR by utilising Lamport logical clocks (LC) for sequencing concurrent messages;
(b) Second, to employ a novel concept of the dynamically determined "last" process for ordering concurrent messages, while ensuring optimal achievable throughput;
(c) Third, the relaxation of the crash failure assumption in LCR.

3.1. Data Structures

The data structures associated with each process P_i, message m, and the μ message are discussed in this section as used in the DCTOP system design and simulation experiment:

Each process P_i has the following data structures:

1. Logical clock (LC_i): This is an integer object initialised to zero and used to timestamp messages.
2. Stability clock (SC_i): As specified in Section 2. SC_i is initially set to zero.
3. Message Buffer ($mBuffer_i$): This field holds the sent or received messages by P_i
4. Delivery Queue (DQ_i) This queue holds messages pending delivery.
5. Garbage Collection Queue (GCQ_i): Once a message has been delivered by P_i, it is subsequently transferred to GCQ_i for garbage collection.

M is used to denote all types of messages used by the protocol. Usually, there are two types of M: the data message denoted by m, and an announcement or ack message that is bound to a specific data message. The latter is denoted as μ(m) when it is bound to m. μ(m) is used to announce that m has been received by all processes in the group. The relationship between m and its counterpart μ(m) is shown in Figure 2.

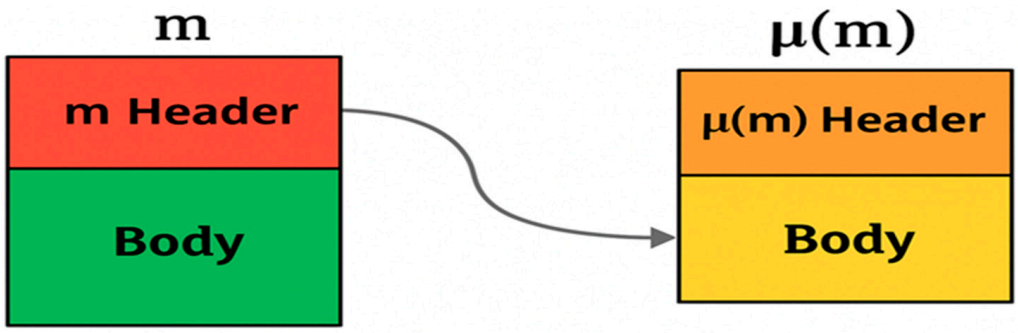


Figure 2. Relationship between m and $\mu(m)$.

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 2).

A message M has at least the following data structures:

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

3.2. DCTOP Principles

The protocol comprises three design aspects: (i) message handling (sending, receiving, and forwarding), (ii) timestamp stability, and (iii) crashproofing, which was introduced in Section 1. This subsection focuses on the first two.

1. Message Sending, Receiving, and Forwarding: The Lamport logical clock is used to timestamp a message *m* within the ring network before *m* is sent. Therefore, *m_ts* denotes the timestamp for message *m*.

The system as shown in Figure 3 uses two main threads in any process P_i to handle message transmission and reception in a distributed ring network. The send(m) thread operates by dequeuing a message m from the non-empty $SendingQueue_i$ when allowed by the transmission control policy (see Section 4). It timestamps the message with the current value of the LC_i as $m\_ts = LC_i$, increments LC_i by one afterwards, and then places the timestamped message into the $OutgoingQueue_i$ for transmission as shown in Figure 3. A copy of the message is also stored in $mBuffer_i$ for the local record. On the receiving side, the receive(m) thread dequeues a message m from the $IncomingQueue_i$ when permitted by the transmission control policy, updates the LC_i as $LC_i = \max\{(m\_ts + 1), LC_i\}$, and delivers the message to process P_i for further handling. Typically, *m* is entered in $mBuffer_i$ and may be forwarded if required to CN_i by entering a copy of *m* with the destination set to CN_i into the $OutgoingQueue_i$. A message is forwarded to CN_i only if it has not completed a full cycle in the ring; once it does, forwarding stops. However, once the message completes a full cycle in the ring network, it is no longer forwarded, and the process stops. When two messages are received in succession, they are transmitted in the same sequence; however, their delivery may not be immediate or consecutive, as determined by the transmission control policy. As shown in Figure 3, received messages enter $IncomingQueue_i$, and are logged in $mBuffer_i$ and forwarded copies are placed in $OutgoingQueue_i$ in receive order.

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) The term "stable" is used 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 [39].

(3) When TS becomes stable in P_i, the process can potentially total order (TO) deliver all previously received but undelivered $m\prime$, $m\prime\_ts \leq$ TS, because the stability of TS eliminates the possibility of P_i ever receiving any $m\prime$, $m\prime\_ts \leq$ TS in the future.

Building on the message-handling and stability mechanisms discussed above, the DCTOP algorithm integrates these principles into a unified procedure for ensuring total-order delivery across all processes. The key operational steps are summarised in Section 3.3.

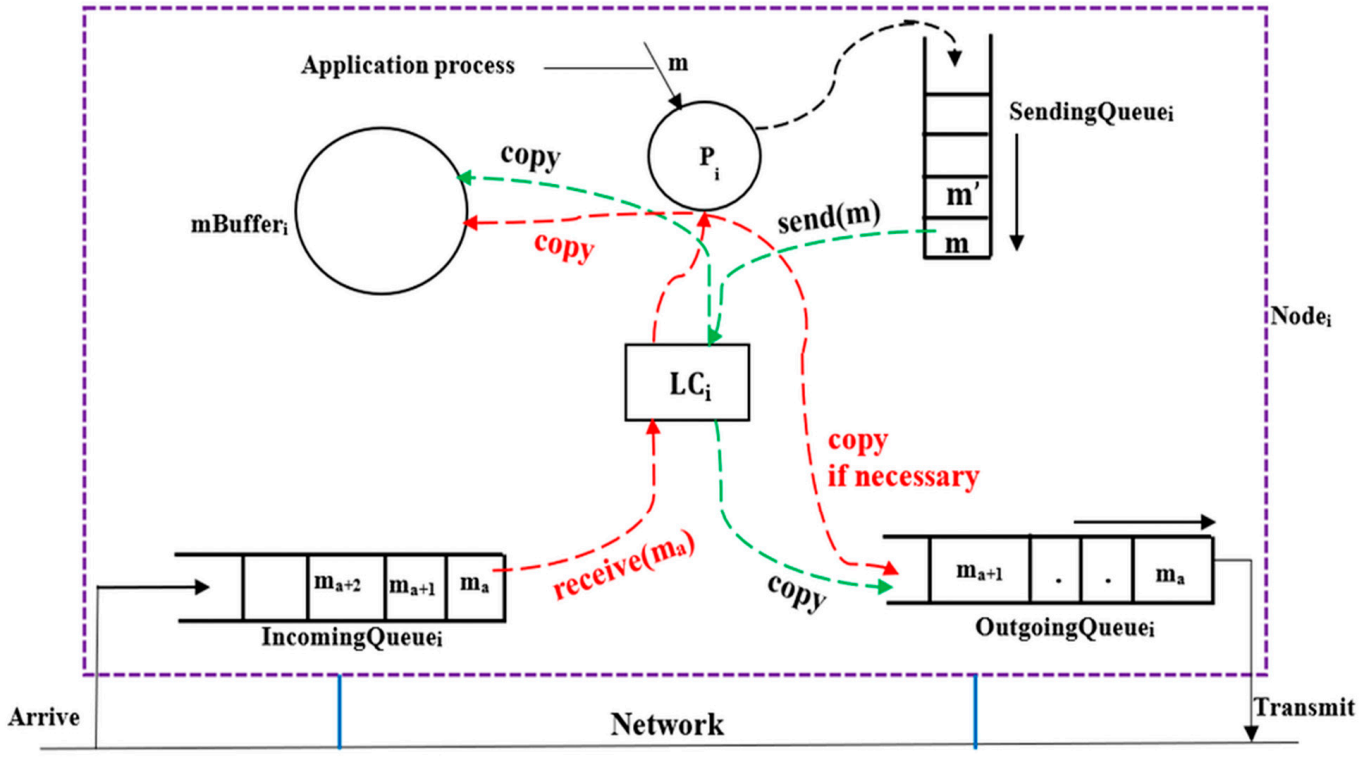


Figure 3. Message sending, receiving, and forwarding.

3.3. DCTOP Algorithm Main Points

The DCTOP algorithm's main points are outlined 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 carries out the following:
   - It checks if $Hops_{j,i} \geq f$. If this condition holds, then m is considered crashproofed and is not delivered immediately. Additionally, m_flag is set to true, and m is deposited accordingly. If m is not crashproofed, m_flag remains false.
   - It then checks if $P_j \neq CN_i$. If so, it sets m destination, m_destn = CN_i and deposits m in its OutgoingQueue$_i$.
   - Otherwise, m is stable, and then it updates SC_i as SC_i = max {SC_i, m_ts}, and transfer all received m, m_ts $\leq SC_i$ to DQ_i. Then, it forms μ(m), sets μ(m)_origin = P_i and μ(m)_destn = CN_i, and deposits μ(m) in OutgoingQueue$_i$.
3. When P_i receives μ(m), it knows that every process has received m:
   - If m *in* μ(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 μ(m); otherwise, if $Hops_{j,i} < f$, P_i sets m_flag = true and μ(m)_destn = CN_i, and deposits μ(m) in $OutgoingQueue_i$.
   - However, if m *in* μ(m) indicates a higher stabilisation in P_i, i.e., m_ts > SC_i, P_i updates SC_i as SC_i = max { SC_i, m_ts}, and transfers all m, m_ts $\leq SC_i$ to DQ_i

- If $P_j = CN_i$, P_i *ignores* μ(m); otherwise, it sets μ(m)_destn = CN_i and deposits μ(m) in $OutgoingQueue_i$.

4. Whenever DQ_i is non-empty, P_i deques *m* from the head of DQ_i and delivers *m* to the 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 (see Figure 4).

Uniform Total Order Delivery and Garbage Collection at *Pi*

```
1.  Procedure utoDeliver(m)
2.    Repeat until DQi is empty
3.        Dequeue m=head (DQi)
4.        If m_flag=true:
5.              Deliver m to the application process.        {deliver a message}
6.              Add m to GCQi.
7.              If DQi is empty or (m_ts < head (DQi):
8.                  Update DCi=m_ts.                         {delivery counter}
```

Figure 4. Algorithm for handling uniform total order delivery in DCTOP.

It is important to note that the DCTOP maintains total order. Thus, if P_i forms and sends *m* and then *m*′, (i) every process receives *m* and then *m*′; (ii) μ(*m*) will be formed and sent before μ(*m*′); and (iii) any process that receives both μ(*m*) and μ(*m*′) will receive μ(*m*) and then μ(*m*′). The message-handling logic discussed above ensures that all processes consistently propagate, stabilise, and acknowledge messages. To complete the protocol description, the following delivery requirements formally define when a message may be safely delivered to the application layer while preserving total order: A message m is deliverable to the high-level application process by P_i once it is both stable and crashproof. Two such messages m and m′ are delivered in total order; m precedes m′ if and only if m_ts < m′_ts or, when timestamps are equal, m_origin > m′_origin.

In summary, the pseudocode representations for total order message delivery, message communication, and membership changes are presented in Figures 4–6, respectively.

Figures 4–6 collectively illustrate the operational flow of the DCTOP protocol. Figure 5 presents the main steps for message multicast and processing, covering initialisation, timestamping, forwarding, and crashproof handling among processes. Figure 6 extends this by showing how the protocol manages group membership changes, ensuring message recovery, queue stabilisation, and consistent delivery when nodes join or leave the system. Finally, Figure 4 demonstrates the uniform total order delivery (utoDelivery) and garbage collection mechanism, where stable messages are delivered to the application layer and archived to preserve correctness and efficiency.

For clarity, Table 1 provides a summary of the main notations used in the membership change procedure as illustrated in Figure 6. These notations correspond to the elements referenced in Steps 1–9 of the algorithm, supporting the reproducibility and interpretation of the DCTOP reconfiguration logic.

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. $LC_i \leftarrow \{0, \ldots..0\}$ {*local logical clock*}
6. $SC_i \leftarrow \{0, \ldots..0\}$ {*stability clock*}
7. $SendingQueue_i \leftarrow \emptyset$ {*outgoing message queue*}
8. Group G ← 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 DQi in total order
30. Form μ(m) with $\mu(m)\_destn=CN_i$
31. Enqueue μ(m) in $OutgoingQueue_i$ {*forward the μ(m)*}
32. **If** (M = μ(m)) **then**
33. **Check if μ(m) is meaningful or not:**
34. If μ(m) contains m where $m\_ts \leq SC_i$ and $Hops_{i,j} \geq f$, discard μ(m).
35. Otherwise:
36. If $Hops_{i,j}<f$
37. Search for m in mBufferi or $DQueue_i$
38. Mark m_flag=true {*message is crashproof*}
39. $\mu(m)\_destn = CN_i$
40. Enqueue μ(m) in $OutgoingQueue_i$ {*forward the μ(m)*}
41. If $m\_ts > SC_i$
42. Update SC_i=m_ts
43. Stabilize all m, $m\_ts \leq SC_i$ in mBufferi {*m is stable*}
44. Move these messages to DQ_i in total order
45. If $P_j=CN_i$, discard μ(m)
46. Otherwise, forward μ(m) to CN_i

Figure 5. Algorithm for handling message multicasting in DCTOP.

Membership Change Steps Executed by any Pi in Survivors(G)

Upon Membership Change (G' ← new group)

1. **Determine Lead Survivor**
   - i. P_i multicasts ($Last_i$_ts, $Last_i$_origin) to all other Survivors $P_s \in G$.
   - ii. After receiving ($Last_s$_ts, $Last_s$_origin) from all P_s, compute:
     - $Last_E$, $Last_L$, and LEAD-Survivor.
2. **Send Pending Messages**
   - (a) For each message $m \in SendingQueue_i$, timestamp and send m to all $P_s \in G$
     - Clear $SendingQueue_i$.
     - (b) For each Survivor P_s, send any missing messages from $mBuffer_i$, TO_Queue_i or QCG_i such that $m \gg Last_s$.
     - (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 $P_s \in G$:
     - i. Receive m.
     - ii. If $m \in \{mBuffer_i, TO\_Queue_i, GCQ_i\}$; discard m (duplicate).
     - iii. Otherwise, store m in $mBuffer_i$.
     - (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 $P_s \in G$.
   - (b) Repeat the following until $mBuffer_i$ is empty:
     - i. Remove m from $mBuffer_i$ in Total order
     - ii. Enqueue m into TO_Queue_i
5. **Handle Joiners (Executed by Lead Survivor)**

   If P_i = LEAD-Survivor and Joiners(G') ≠ 0
   - (a) Compute checkpoint C
   - (b) Send Invite (C, TO_Queue_i) to each $P_j \in$ NewComer(G')
6. **Resume Delivery in Previous Group GPrev**
   - (a) Repeat until TO_Queue_i is empty:
     - i. Dequeue m.
     - ii. Deliver m to the application process.
     - iii. Enqueue m into GCQ_i.
   - (b) Send $Completed_i$ to all $P_k \in G$.
7. **Initialize for New Group G'**
   - i. Wait until $Completed_k$ is received from all $P_k \in G'$.
   - ii. Initialize DCTOP variables.
   - iii. Clear buffers and queues.
   - iv. Resume DCTOP in G'.

   For Joiner Process P_j:
8. **Receive Checkpoint Update**
   - i. Wait until Invite (C, Q) is received from all $P_i \in G$.
   - ii. Apply checkpoint C.
   - iii. Set $TO\_Queue_j \leftarrow Q$.
9. **Follow Steps 6 and 7 for Survivors**

Replace P_i references with P_j

Figure 6. Algorithm for handling membership changes in DCTOP.

Table 1. Notations used in DCTOP membership change algorithm.

Notations	Description
Pi	Process *i* in the group (active node).
G	Group of DCTOP processes executing the protocol at any given time
Π	The set of processes in a group
G′	New group formed after membership change
Survivors(G)	Subset of processes that remain active after failure detection.
Gprev	The old group version that *immediately precedes G*
Lead − Survivor	The process elected to coordinate recovery.
Joiners(G′)	Processes newly joining the group G′.
mBuffer$_i$	Buffer storing messages held by process Pi
TO_Queue$_i$	Total Order Queue; holds messages ready for delivery in order.
GCQ$_i$	Garbage Collection Queue: stores messages already delivered.
Last_ts	Timestamp of the last stable message before membership change.
Last_origin	The last stable message origin before membership change.
Last$_E$ and Last$_L$	The earliest and the latest among the last messages delivered by processes of Survivors(G)
C	Checkpoint of system state computed by Lead-Survivor.
Q	Queue of stable messages transmitted to new joiners.
Invite(C, Q)	Message sent by Lead-Survivor to new joiners containing the checkpoint and current total order queue.
Finished$_i$	Notification from process Pi after it finishes transmitting any missing/pending messages to other survivors (Step 2).
Ready$_i$	Notification from process Pi to all survivors that it has received and stored all messages required for the current recovery phase (Step 3)
Completed$_i$	Sent by Pi after it empties TO_Queue from the previous group and resumes normal delivery (Step 6)

3.4. Group Membership Changes

The DCTOP protocol is built on top of a group communication system [40,41]. Membership of the group of processes executing the 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 $\Pi = \{P_0, P_1, P_2, \ldots, P_{N-1}\}$ and $G \subseteq \Pi$ is always true. The membership change procedure is detailed in Figure 6 and all the notations used are described in Table 1. Note that 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 the processing of any message it has already started processing and then suspends all DCTOP activities and waits for the new G′ to be formed: the sending of *m* or μ(m) (by enqueueing into SendingQueue$_i$), receiving of *m* or μ(m) (from IncomingQueue$_i$), and delivering of m (from DQ$_i$) are all suspended. The group membership change works as follows: Each process P_i in the set of Survivors(G) (i.e., survivors of the current group G) exchanges information about the last message they TO-delivered. Once this exchange is complete, additional useful information is derived among all Survivors, which helps identify the Lead Survivor. Subsequently, each Survivor sends all messages from its respective SendingQueue to the other Survivors. If P_i has any missing

messages, they are sent to another Survivor, P_s, where P_s represents any Survivor process other than P_i. After sending, P_i transmits a $Finished_i$ message to all P_s processes, signalling that it has completed its sending. Upon receiving messages, P_i stores all non-duplicate messages in its buffer, $mBuffer_i$. The receipt of $Finished_s$ messages from all P_s processes confirms that P_i has received all expected messages, with duplicates discarded. P_i then waits to receive $Ready_s$ from every other P_s, ensuring that every Survivor P_s, has received the messages sent by P_i. At this point, all messages in $mBuffer_i$ are stable and can be totally ordered. If there are Joiners (defined as incoming members of G′ that were not part of the previous group (Gprev) but joined G′ after recovering from an earlier crash), the Lead Survivor sends its checkpoint state and TO_Queue to each P_j in the set of NewComer(G′), allowing them to catch up with the Survivors(G). Following this, all Survivors(G) resume TO delivery in Gprev. P_i then sends a $completed_i$ message to every process in G, indicating that it has finished TO-delivering in Gprev. Each Survivor waits to receive a $completed_k$ message from every other P_k in G before resuming DCTOP operations in the new G′. The Joiners, after replicating the Lead Survivor's checkpoint state, also performed the TO delivery of messages in Gprev and then resumed operations in the new G′ of the DCTOP. Hence, at the conclusion of the membership change procedure, all buffers and queues are emptied, ensuring that all messages from Gprev have been fully processed.

3.5. Proof of Correctness

Lemma 1 (VALIDITY). *If any correct process P_i uniformly total order Multicasts (utoMulticasts) a message m, then it will eventually uniformly total order Deliver m. For brevity, the term "uto" is used as a prefix in all related procedures, e.g., utoMulticast and utoDeliver.*

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 5). 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 ensure that P_i will deliver all messages stored in its $mBuffer_i$, TO_Queue_i, or GCQ_i, including m_i (Line 32 to 44 of Figure 5). 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 to be stable (Line 28 of Figure 5). This happens because, when the P_i timestamp 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 their mBuffer and sets $LC > m_i$_ts. It also checks if $Hops_{i,j} \geq f$; if so, then m_i is crashproof and it sets m_i_flag=true. The ACN_i sets m_i to be stable (Line 28 of Figure 5) and crashproof (Line 20 of Figure 5) at ACN_i, transfers m_i to DQ, and then it attempts to utoDeliver m_i (Lines 1 to 8 of Figure 4) 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 CN. Similarly, $\mu(m_i)$ is forwarded along the ring (Line 31 of Figure 5) until the ACN of $\mu(m_i)$_origin receives $\mu(m_i)$. When any process receives $\mu(m_i)$ and $Hops_{i,j} < f$, it knows that m_i is crashproof and stable, but, if $Hops_{i,j} \geq f$, 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 $i < k$ (this study assumed that, when messages have an equal timestamp, the message from a higher origin is ordered before the 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 is 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, then it eventually delivers m.

Note that, if f + 1 processes receive a message m, then m is crashproof, and, during the concurrent multicast, TS can become stable quickly, making m to be delivered even before the ACN of the m_origin receives m. □

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. The crash failure assumption in this study ensures that no false message is ever utoDelivered by a process. Thus, only messages that have been utoMulticast are utoDelivered. Moreover, each process maintains an *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 5 guarantee that no message is processed twice by process P_k. In the case of a membership change, Line 3a(ii) of Figure 6 ensures that process P_k does not deliver messages twice. Additionally, Lines 7(i–iv) of Figure 6 ensure that P_k's variables such as the logical and stability clock are set to zero, and the buffer and queues are emptied after a membership change. This is carried out because the processes had already delivered all the messages of the old group, discarding message duplicates (Line 3a(ii) of Figure 6) 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 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 ensures 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 in the presence of a membership change. 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 6a(ii) of Figure 6. 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 Line 1 to 9 of Figure 6. 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 (Line 6a(ii) of Figure 6). 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 be crashproof and stable. Remember, when any process receives $\mu(m_i)$ and $Hops_{i,j} < f$, it knows that m_i is crashproof and stable, but, if $Hops_{i,j} \geq f$, 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 3 of Figure 4). 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 be crashproof and stable, it can no longer receive any message m such that $m \prec m_i$. Indeed, a process P_c can only produce a message $m_c \prec m_i$ 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 an $\mu(m_i)$ for m_i will have first received m_c. Consequently, every process setting m_i to be crashproof and stable will have first received m_c. The second reason is a consequence of the fact that, for every message m that is utoMulticast in the system, the protocol ensures that m and μ(m) will be forwarded around the ring (Lines 25 and 31 of Figure 5), 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 the 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 it ever have m' at any time after t.

Proof (By Contradiction). Assume, contrary to Lemma, that P_i is to receive m', $m'\_ts \leq TS$, after t as shown in Figure 7a.

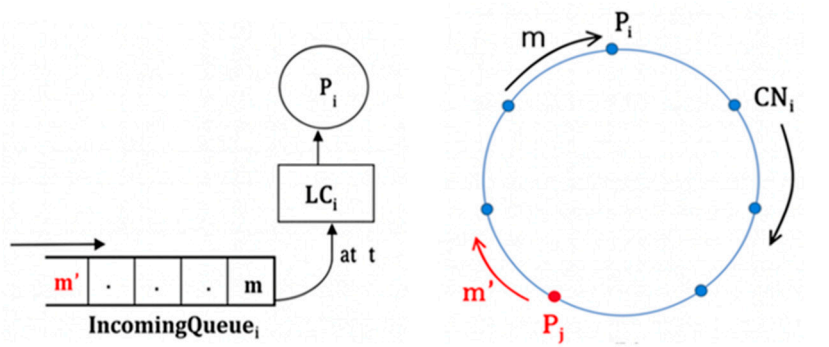
Case 1: Let $m\_origin = m'\_origin = P_j$. Therefore, imagine that P_j is the same as CN_i, $P_j \equiv CN_i$, as shown in Figure 7b. Given that $m'\_ts \leq TS = m\_ts$, $m'\_ts < m\_ts$ must be true when $m\_origin = m'\_origin$. Therefore, P_j must have sent m' first and then m.

Note the following:

(a) The link between any pair of consecutive processes in the ring maintains FIFO;
(b) Processes P_{j+1}, P_{j+2},.... 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. Therefore, 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 7b. Since P_i is the last process to receive m in the system, P_j must have received m before t; since m′_ts ≤ m_ts, P_j could not have sent $m\prime$ after receiving m. Therefore, the only possibility for m′_ts ≤ m_ts to hold is as follows: P_j must form and send $m\prime$ before it is received and forwarded m. For the cases of (a) and (b) in case 1, P_i must receive $m\prime$ before m. Therefore, the assumption made contrary to Lemma 1 cannot be true. Thus, Lemma 1 is proven. □



(a) Hypothetical incoming queue state contradicting total order

(b) Message flow in ring topology illustrating contradiction scenario

Figure 7. Example contradicting Lemma 1.

4. Fairness Control Environment

In this section, the DCTOP fairness mechanism was discussed: 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 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 $IncomingQueue_i$ 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 $SendingQueue_i$ consists 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 in its $IncomingQueue_i$ but, whenever the $SendingQueue_i$ is not empty, a rule is required to coordinate the sending and forwarding of messages to achieve fairness. 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, a data structure called *forwardlist* was introduced. 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 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 $forwardlist_i$.

5. Experiments and Performance Comparison

This section presents a performance comparison of the DCTOP protocol against the LCR [9] protocol and Raft [17,18], a widely implemented, leader-based ordering protocol, by evaluating the latency and throughput across varying numbers of messages transmitted within the cluster environment. A Java (OpenJDK-17, Java version 17.02) framework was used to run a discrete event simulation for the protocols with at most nine processes, $N = 4, 5, 7,$ and 9. Every simulation method made use of a common PC with a 3.00 GHz 11th Gen Intel(R) Core (TM) i7-1185G7 Processor and 16 GB of RAM. A request is received from the client by each process, which then sends the request as a message to its neighbour on the ring-based network. When one process receives a message, it forwards the message to another process, continuing this pattern until all processes have received the message. When the ACN of the message origin receives the message, then it knows that it is stable and tries to deliver it, a process known as uniform total order delivery [42]. The process then uses an acknowledgement called a μ-message to notify all other processes of the message's stability, which they were previously unaware of. Other processes that receive this acknowledgement recognise the message's stability and endeavour to ensure its delivery in total order. For a Raft cluster, when a client sends a request to the leader, the leader adds the command to its local log, then sends a message to follower processes to replicate the entry. Once a majority (including the leader) confirms replication, the entry is committed. The leader then applies the command to its state machine for execution, notifies followers to do the same, and responds to the client with the output of execution.

The time between successive message transmissions is modelled as an exponential distribution with a mean of 30 milliseconds, reflecting the memoryless property of this distribution, which is well-suited for representing independent transmission events. The delay between the end of one message transmission and the start of the next is also assumed to follow an exponential distribution, with a mean of 3 milliseconds, to realistically capture the stochastic nature of network delays. For the simulation, process replicas are assumed to have 100% uptime, as crash failure scenarios were not considered. Additionally, no message loss is assumed, meaning every message sent between processes is successfully delivered without failure. The simulations were conducted with a varying number of process replicas, such as four, five, seven, and nine processes. The arrival rate of messages follows a Poisson distribution with an average of 40 messages per second, modelling the randomness and variability commonly observed in real-world systems. The simulation duration ranges from 40,000 to 1,000,000 s. This extended period is chosen to ensure the system reaches a steady state and to collect sufficient data for a 95% confidence interval analysis. The long duration also guarantees that each process sends and delivers between one million (1 Mega ($\times 10^6$)) and twenty-five million (25 Mega ($\times 10^6$)) messages.

Latency: These order protocols calculate latency as the time difference between a process's initial transmission of a message m and the point at which all m destinations deliver m in total order to the applications process. For illustration, let t_0 represent the time when process P_0 sends a message to its CN_0, and let t_1 denote the time when the final process in the cluster (e.g., P_2 in a four-node setup) delivers that same message. The latency for that message is, therefore, defined as $(t_1 - t_0)$, representing the maximum delivery time observed. The average of such maximum latencies, computed across 1 to 25 million messages, was then calculated and repeated 10 times to obtain a 95% confidence interval. The average maximum latency was plotted against the number of messages sent by each process.

Throughput: The throughput is calculated 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. Similarly, to the latency, we also determined a 95%

confidence interval for the average maximum throughput. Additionally, we presented the latency enhancements offered by the proposed protocol in comparison to LCR, as well as the throughput similarities. Nevertheless, all experiments were carried out independently to prevent any inadvertent consequences of running multiple experiments simultaneously. All processes were configured to send and receive an equal number of messages at uniform transmission intervals, ensuring a balanced message distribution and adherence to the fairness control mechanism during simulation. For this preliminary validation, failure and fault models are left out to focus on verifying the DCTOP's key features and order consistency during controlled asynchronous communication. The Discrete Event Simulation framework monitors the flow of messages, their timestamps, and when stability is reached, all without external disruptions. Once the protocol's correctness is validated, future evaluations in cloud environments will explore how it handles failures, message loss, and scalability. In Figures 8 and 9, the Y axis indicates latency measured in seconds, and the X axis is rescaled to mega ($\times 10^6$) for clarity.

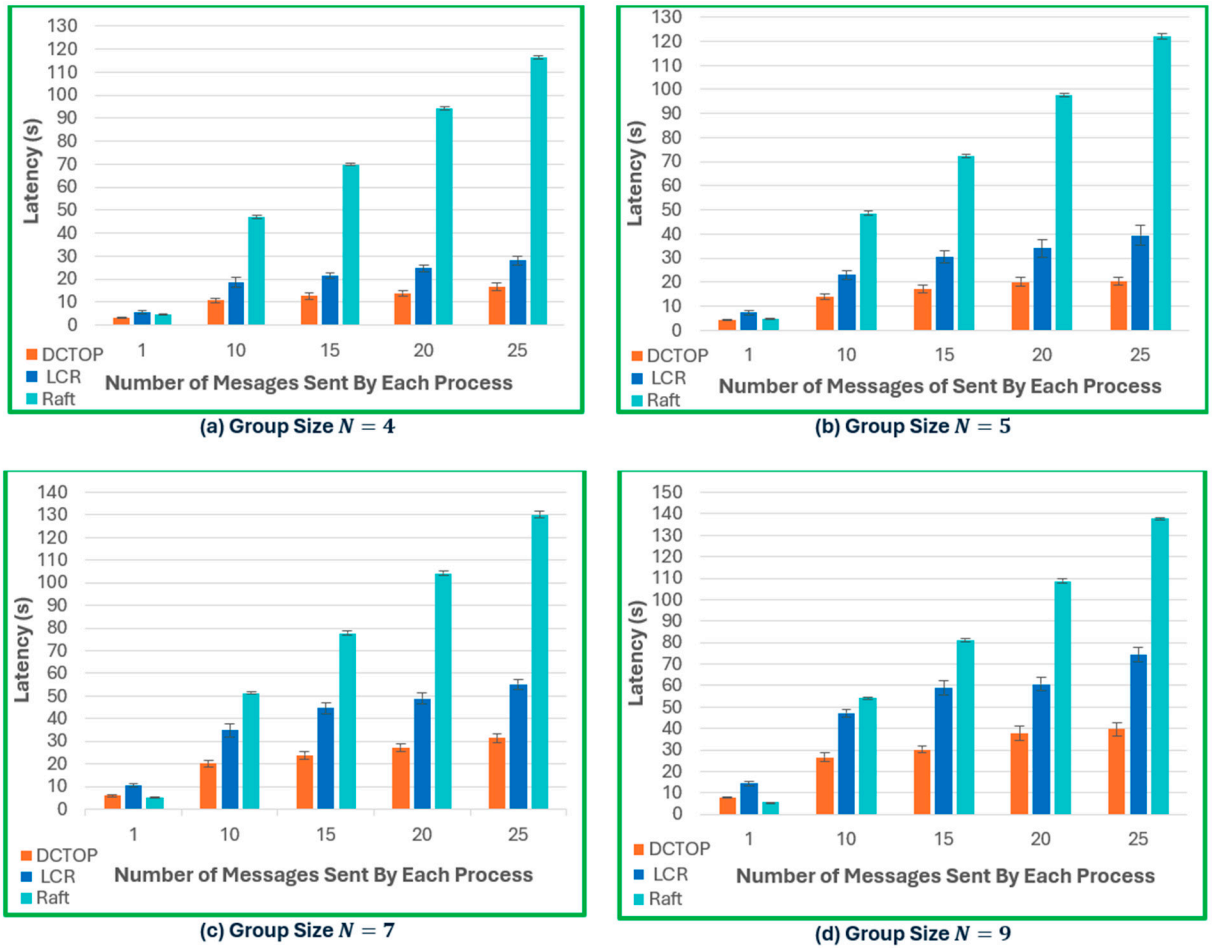


Figure 8. Latency comparison of Raft, LCR and DCTOP protocols under varying message load and group sizes (N = 4, 5, 7, and 9).

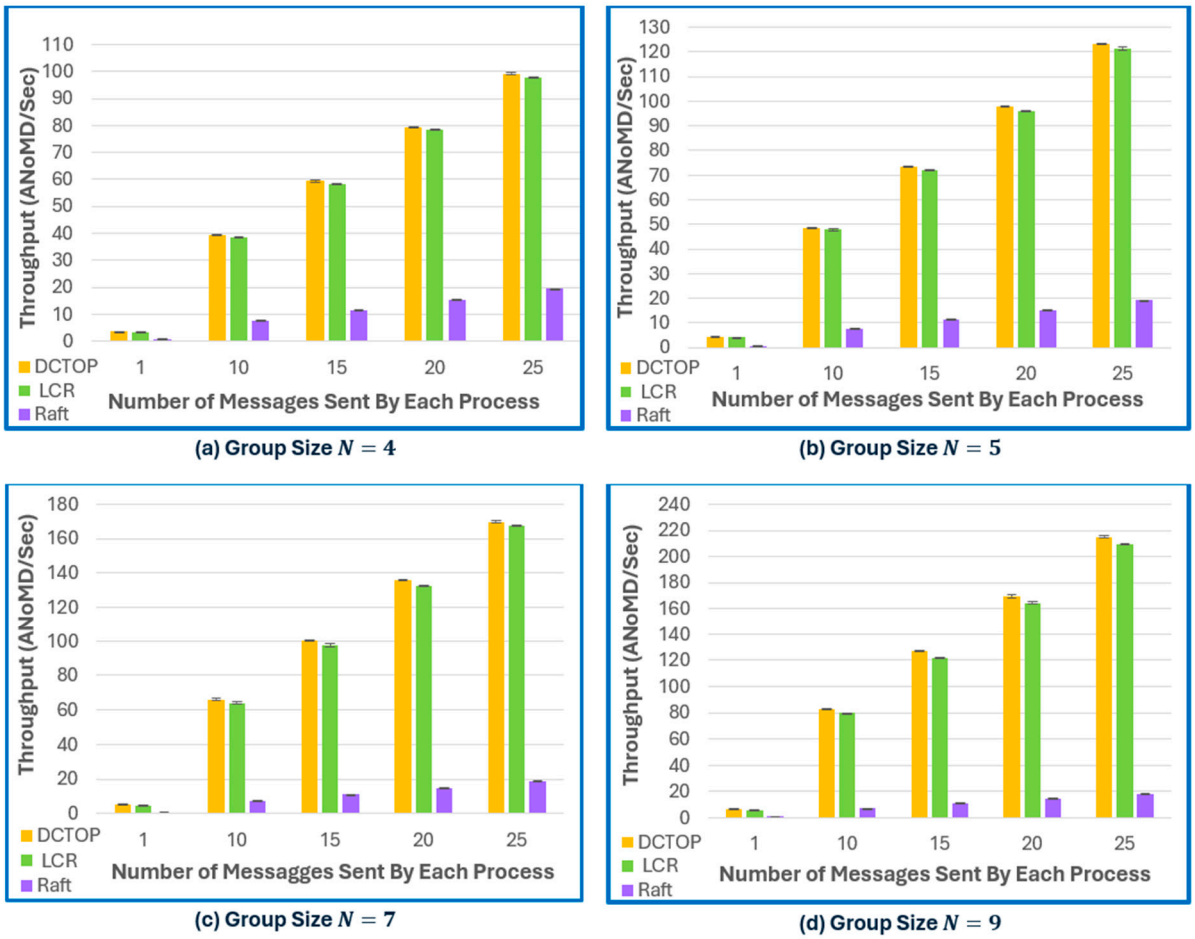


Figure 9. Throughput comparison of Raft, LCR, and DCTOP protocols under varying message load and group sizes (N = 4, 5, 7, and 9).

Results and Discussion

As shown in Figure 8a–d, the latency trends for the DCTOP, LCR, and Raft across increasing group sizes (N = 4, 5, 7, and 9) and message volumes reveal distinct performance characteristics. The DCTOP consistently exhibits the lowest latency across all configurations. This efficiency stems from its use of Lamport logical clocks for the lightweight sequencing of concurrent messages, the dynamic assignment of a unique last process for each message originator, and a relaxed crash-failure assumption that enables faster message stabilisation and delivery. LCR shows a moderately higher latency, which increases with both the group size and message load. This behaviour can be attributed to its reliance on the vector timestamp, whose dimensionality scales with the number of processes, and the use of a globally fixed last process for concurrent message ordering. Together, these design choices introduce larger message headers and a higher coordination overhead, particularly in larger groups. Raft, as a leader-based protocol, consistently incurs the highest latency, especially under heavier loads. The centralisation of log replication and client handling at the leader creates a sequential processing bottleneck, which limits the responsiveness as the system load grows. However, under lower traffic conditions (e.g., 1 million messages per process), Raft performs competitively and, in configurations with N = 7 (see Figure 8c) and N = 9 (see Figure 8d), even outperforms the DCTOP and LCR. This suggests that Raft may remain suitable in low-load or moderately scaled environments.

In terms of the throughput as illustrated in Figure 9a–d, the DCTOP and LCR outperform Raft across all group sizes and message volumes. The DCTOP and LCR are leaderless ring-based order protocols that benefit from a decentralised execution, enabling all pro-

cesses to concurrently receive and process client requests. This distributed handling results in a cumulative throughput that almost scales linearly with group size. In contrast, Raft centralises all client communication and log replication at a single leader. This architectural bottleneck constrains the throughput to the leader's processing capacity, causing performance to saturate under higher loads. Among the leaderless protocols, the DCTOP achieves the highest throughput, followed closely by LCR, which incurs a small overhead due to its vector timestamp management and centralised ordering logic. Raft consistently exhibits the lowest throughput, and its performance plateaus with increasing load, reinforcing the inherent scalability limitations of leader-based protocols in high-throughput environments. Notably, all three protocols—Raft, LCR, and the DCTOP—were implemented from a unified code base, differing only in protocol-specific logic. The experiments were conducted under identical evaluation setups and hardware configurations, ensuring a fair and unbiased comparison. In summary, these findings indicate that decentralised, leaderless architectures such as the DCTOP and LCR achieve a higher scalability and throughput efficiency relative to leader-based consensus protocols like Raft.

6. Conclusions

In this study, we introduced the DCTOP, a novel ring-based and leaderless total order protocol that extends the classical LCR approach through three innovations: the integration of Lamport logical clocks for concurrent message sequencing, a dynamic last-process identification mechanism, and a relaxed crash failure assumption. Together, these mechanisms significantly reduced the message latency and improved fairness in distributed message ordering. The simulation results across group sizes (N = 4, 5, 7, and 9) demonstrated that the DCTOP achieves an over 43% latency improvement compared to LCR under concurrent message loads, confirming the effectiveness of its design. The inclusion of Raft as a benchmark further emphasised the DCTOP's scalability advantage in decentralised coordination settings.

Overall, this work establishes the DCTOP as a viable foundation for efficient and fair total order delivery in distributed systems. Future work will extend this study to large-scale, fault-tolerant cloud environments to evaluate the performance under realistic failure conditions and dynamic workloads.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/digital5040064/s1.

Author Contributions: Conceptualization, A.E.; methodology, A.E.; software, A.E.; validation, A.E. and C.I.N.; formal analysis, A.E.; investigation, A.E., J.N.O.; resources, A.E.; data curation, A.E.; writing—original draft preparation, A.E.; writing—review and editing, C.I.N., E.A.E., J.N.O.; visualization, A.E.; supervision, C.I.N.; project administration, E.A.E. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article/Supplementary Materials. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LCR	Logical Ring and Ring Protocol
DCTOP	Daisy Chain Total Order Protocol
VC	Vector Clock

LC	Logical Clock
TO	Total Order
TS	Timestamp Stability
Π	The set of processes in a ring
CN	Clockwise Neighbour
ACN	Anti-Clockwise Neighbour
SC	Stability Clock
DQ	Delivery Queue
GCQ	Garbage Collection Queue
FIFO	First In First Out
BFT	Byzantine Fault Tolerance
FSR	Fixed Sequence Ring

References

1. Choudhury, A.; Garimella, G.; Patra, A.; Ravi, D.; Sarkar, P. Crash-tolerant consensus in directed graph revisited. In *International Colloquium on Structural Information and Communication Complexity*; Springer International Publishing: Cham, Switzerland, 2018; pp. 55–71. [CrossRef]
2. Pease, M.; Shostak, R.; Lamport, L. Reaching agreement in the presence of faults. *J. ACM (JACM)* **1980**, *27*, 228–234. [CrossRef]
3. Vollset, E.W.; Ezhilchelvan, P.D. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in manets. In Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems, Orlando, FL, USA, 26–28 October 2005; pp. 166–175.
4. Correia, M.; Ferro, D.G.; Junqueira, F.P.; Serafini, M. Practical hardening of crash-tolerant systems. In Proceedings of the 2012 USENIX Conference on Annual Technical Conference, Boston, MA, USA, 13–15 June 2012; pp. 453–466.
5. Ezhilchelvan, P.D.; Macedo, R.A.; Shrivastava, S.K. Newtop: A fault-tolerant group communication protocol. In Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, BC, Canada, 30 May–2 June 1995; pp. 296–306. [CrossRef]
6. Wiesmann, M.; Pedone, F.; Schiper, A.; Kemme, B.; Alonso, G. Understanding replication in databases and distributed systems. In Proceedings of the 20th IEEE International Conference on Distributed Computing Systems, Taipei, Taiwan, 10–13 April 2000; pp. 464–474. [CrossRef]
7. Helal, A.A.; Heddaya, A.A.; Bhargava, B.B. *Replication Techniques in Distributed Systems*; Springer Science & Business Media: Boston, MA, USA, 2006.
8. Défago, X.; Schiper, A.; Urbán, P. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. (CSUR)* **2004**, *36*, 372–421. [CrossRef]
9. Guerraoui, R.; Levy, R.R.; Pochon, B.; Quéma, V. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst. (TOCS)* **2010**, *28*, 1–32. [CrossRef]
10. Guerraoui, R.; Levy, R.R.; Pochon, B.; Quema, V. High Throughput Total Order Broadcast for Cluster Environments. In Proceedings of the International Conference on Dependable Systems and Networks (DNS'06), Philadelphia, PA, USA, 25–28 June 2006; pp. 549–557. [CrossRef]
11. Junqueira, F.; Reed, B. *ZooKeeper: Distributed Process Coordination*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2013.
12. Hunt, P.; Konar, M.; Junqueira, F.P.; Reed, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10), Boston, MA, USA, 23–25 June 2010.
13. Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, USA, 6–8 November 2006; pp. 335–350.
14. Pu, J.; Gao, M.; Qu, H. SimpleChubby: A Simple Distributed Lock Service. Available online: https://www.scs.stanford.edu/14au-cs244b/labs/projects/pu_gao_qu.pdf (accessed on 31 May 2024).
15. Lamport, L. Paxos made simple. In *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*; ACM: New York, NY, USA, 2001; pp. 51–58.
16. Liskov, B.; Cowling, J. Viewstamped Replication Revisited. MIT Technical Report MIT-CSAIL-TR-2012-021. 2012. Available online: https://dspace.mit.edu/handle/1721.1/71763 (accessed on 7 April 2024).
17. Ongaro, D.; Ousterhout, J. In Search of an Understandable Consensus Algorithm (Extended Version). Tech Report. May 2014. Available online: https://raft.github.io/raft.pdf (accessed on 6 June 2024).
18. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14), Philadelphia, PA, USA, 19–20 June 2014; pp. 305–319.

19. Ejem, A.; Ezhilchevan, P. Design and performance evaluation of raft variations. In Proceedings of the 39th Annual UK Performance Engineering Workshop (2023), Birmingham, UK, 7–8 September 2023.
20. Lamport, L. Fast Paxos. *Distrib. Comput.* **2006**, *19*, 79–103. [CrossRef]
21. Biely, M.; Milosevic, Z.; Santos, N.; Schiper, A. S-paxos: Offloading the leader for high throughput state machine replication. In Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems, Irvine, CA, USA, 8–11 October 2012; pp. 111–120. [CrossRef]
22. Moraru, I.; Andersen, D.G.; Kaminsky, M. There is more consensus in Egalitarian parliaments. In Proceedings of the 24th ACM Symposium on Operating Systems Principles, Farminton, PA, USA, 3–6 November 2013; pp. 358–372. [CrossRef]
23. Yin, M.; Malkhi, D.; Reiter, M.K.; Gueta, G.G.; Abraham, I. HotStuff: BFT consensus with linearity and responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19), Toronto, ON, Canada, 29 July–2 August 2019; pp. 347–356. [CrossRef]
24. Stathakopoulou, C.; David, T.; Pavlovic, M.; Vukolić, M. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *J. Syst. Res.* **2022**, *2*, 1–34. [CrossRef]
25. Guo, B.; Lu, Z.; Tang, Q.; Xu, J.; Zhang, Z. Dumbo: Faster asynchronous BFT protocols. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), Virtual Event, USA, 9–13 November 2020; pp. 803–818. [CrossRef]
26. Gao, Y.; Lu, Y.; Lu, Z.; Tang, Q.; Xu, J.; Zhang, Z. Dumbo-NG: Fast asynchronous BFT consensus with throughput-obliv ious latency. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), Los Angeles, CA, USA, 7–11 November 2022; pp. 1187–1201. [CrossRef]
27. Monti, M.; Camaioni, M.; Roman, P.-L. Fast Leaderless Byzantine Total Order Broadcast. *arXiv* **2024**, arXiv:2412.14061. Available online: https://arxiv.org/abs/2412.14061 (accessed on 17 October 2025). [CrossRef]
28. Mao, C.; Golab, W.; Wong, B. Antipaxos: Taking interactive consistency to the next level. *J. Parallel Distrib. Comput.* **2024**, *187*, 104839. [CrossRef]
29. Stathakopoulou, C.; Wei, M.; Yin, M.; Zhang, H.; Malkhi, D. BBCA-LEDGER: High throughput consensus meets low latency. *arXiv* **2023**, arXiv:2306.14757. Available online: https://arxiv.org/abs/2306.14757 (accessed on 17 October 2025). [CrossRef]
30. Marandi, P.J.; Primi, M.; Schiper, N.; Pedone, F. Ring Paxos: A high-throughput atomic broadcast protocol. In Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Chicago, IL, USA, 28 June–1 July 2010; pp. 527–536. [CrossRef]
31. van Renesse, R.; Schneider, F.B. Chain replication for supporting high throughput and availability. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04), San Francisco, CA, USA, 6–8 December 2004; p. 7.
32. Whittaker, M.; Giridharan, N.; Szekeres, A.; Hellerstein, J.M.; Stoica, I. Bipartisan Paxos: A Modular State Machine Replication Protocol. *arXiv* **2020**, arXiv:2003.00331. Available online: https://arxiv.org/abs/2003.00331 (accessed on 19 October 2025). [CrossRef]
33. Bianco, A.; Cuda, D.; Finochietto, J.M. Short-Term Fairness in Slotted WDM rings. *Comput. Netw.* **2015**, *83*, 235–248. [CrossRef]
34. Ejem, A.; Njoku, C.N.; Uzoh, O.F.; Odii, J.N. Queue Control Model in a Clustered Computer Network using M/M/m Approach. *Int. J. Comput. Trends Technol. (IJCTT)* **2016**, *35*, 12–20. [CrossRef]
35. Fischer, M.J.; Lynch, N.A.; Paterson, M.S. Impossibility of distributed consensus with one faulty process. *J. ACM* **1985**, *32*, 374–382. [CrossRef]
36. Lamport, L. The Part-Time Parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
37. Kreps, J.; Narkhede, N.; Jun, R. Kafka: A Distributed Messaging System for Log Processing. In Proceedings of the NetDB 2011, Athens, Greece, 12 June 2011; Volume 11, pp. 1–7. Available online: https://classpages.cselabs.umn.edu/Spring-2018/csci8980/Papers/PublishSubscribe/Kafka.pdf (accessed on 6 November 2025).
38. Zhang, I.; Sharma, N.K.; Szekeres, A.; Krishnamurthy, A.; Ports, D.R.K. Building Consistent Transactions witInconsistent Replication. *ACM Trans. Comput. Syst. (TOCS)* **2018**, *35*, 1–37. [CrossRef]
39. Chandy, K.M.; Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. (TOCS)* **1985**, *3*, 63–75. [CrossRef]
40. Birman, K.; Joseph, T. Exploiting virtual synchrony in distributed systems. In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, Austin, TX, USA, 8–11 November 1987; pp. 123–138. [CrossRef]

41. Amir, Y.; Stanton, J. *The Spread Wide Area Group Communication System*; Technical Report: CNDS 98-4; Johns Hopkins University, Center for Networking and Distributed Systems: Baltimore, MD, USA, 1998.
42. Vicente, P.; Rodrigues, L. An indulgent uniform total order algorithm with optimistic delivery. In Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings, Suita, Japan, 13–16 October 2002; pp. 92–101. [CrossRef]