Article

Not peer-reviewed version

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

Agbaeze Ejem [*] , Cosmas Ifeanyi Nwakanma , Ejem Agwu Ejem , Juliet Nnenna Odii

Posted Date: 4 September 2025

doi: 10.20944/preprints202509.0458.v1

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

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, WR1 3AS, United Kingdom
[2] Computer Science and Electrical Engineering, West Virginia University, 26506, USA
[3] Department of Logistics and Supply Chain Management, Federal University of Technology Owerri, Nigeria
[4] Department of Computer Science, Federal University of Technology Owerri, Nigeria
* Correspondence: a.ejem@worc.ac.uk; Tel.: +447879178581

Abstract

Server replication ensures crash tolerance by enforcing a total order of input requests across multiple servers. The Logical Clock and Ring (LCR) protocol, a ring-based leaderless total order protocol, achieves high throughput by arranging processes in a logical ring with unidirectional message flow. However, LCR design assumption may not produce optimal latency under high message concurrency due to its use of vector clocks as vector timestamps for sequencing messages and a fixed "last" process for ordering concurrent messages. To improve latency, we propose using Lamport's logical clock as a message timestamp for sequencing messages and redefining the "last" process as the nearest process in the opposite direction of message flow, ensuring a unique last process for each message sender. Fairness is preserved using a modified fairness control algorithm from the Fixed Sequencer and Ring (FSR) protocol. Our evaluation shows that the proposed protocol offers latency improvement better than LCR across all considered configurations. Additionally, fairness among process replicas was maintained, evidenced by an even distribution of message sending responsibilities, with each process contributing approximately equally to total message output.

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

---

1. Introduction

A crash-tolerant system is designed to continue functioning despite a threshold number of crashes occurring and is crucial to maintaining 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. 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 different order, but despite this, all servers must process client requests in the same order [5,6]. 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 [7]. A logical lock 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 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.

Total order protocols in the literature broadly fall into two categories: Leader-based and leaderless. In the leader-based protocol, every client request is routed through the leader which coordinates the request replication and responds to the clients with the results of execution. Examples include Apache Zookeeper [10,11], Chubby [12,14], Paxos [15], View-stamp replication [25] and Raft [9,16,18]. Ring-based protocols are a class of leaderless protocols whose nodes are arranged in a logical ring. They ensure that all messages are delivered by all processes in the same order, regardless of how they were generated or sent by the sender. An example includes LCR [19], FSR [22], E-Paxos [20], and S-Paxos [21]. Our study focuses specifically on the ring-based leaderless protocols. Total order protocols are widely applicable to distributed systems, especially in applications requiring strong consistency and high throughput. For instance, they are utilised to coordinate transactions in massive in-memory database systems [17,23] where achieving minimal latencies despite heavy load is critical.

However, despite the progress made in ring-based protocols like LCR, certain design choices may lead to increased latency. The Logical Clock and Ring (LCR) protocol utilizes *vector clocks* where each process, denoted as P_i, maintains its own clock as $\mathrm{VC_i} = \mathrm{vc}_{\mathrm{k,k=0,\ldots N-1}}$. 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 [28], 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.

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 $\mathrm{N = f + 1}$ implies that $\mathrm{f = N - 1}$, 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 $\mathrm{N = f + 1}$ 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, $\mathrm{N} > 2$, 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 $\mathrm{P_1}$ until it gets to $\mathrm{P_3}$ which is the last process for $\mathrm{P_0}$. The study aims to achieve the following objectives: (i) Optimize message timestamping with Lamport logical clocks, which uses a single integer to represent message timestamps. This approach is independent of N, the number of processes in the communication cluster, unlike the vector timestamping used in LCR, which is dependent on N. In LCR, as N increases, the size of the vector timestamp grows, leading to information overhead. By contrast, Lamport's clock maintains a constant timestamp size, reducing complexity and improving efficiency

(ii) Dynamically determines 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 as a means to minimize 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 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 a group communication can crash. 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 [8]. 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 [9]**.** 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.

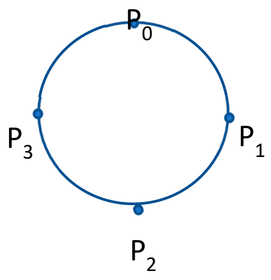


Figure 1. Last Process Concept.

In addition, a new concept of "deliverability requirements" was introduced to guarantee the delivery of only crash-proof 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) [13]. 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 sending of messages.

1.1. Contributions

The contributions of this paper can be summarized as follows:

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

   a. Lamport Logical Clock used for message timestamping which achieves efficient concurrent message ordering, reducing latency and improving fairness.

   b. Dynamic Last-Process Identification to replace LCR's globally fixed last process assumption, accelerating message stabilization and accelerates delivery.

(ii) Relaxed Failure Assumption: DCTOP reduces the fault tolerance threshold from N = f + 1 to N = 2f + 1, enabling faster message delivery with fewer failures.

(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 DCTOP under practical conditions.

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

2. System Model

The ring-based protocols are modelled as a group of N processes represented by $\Pi = \{P_0, P_1, P_2, \ldots, P_{N-1}\}$ which are linked together in a circular structure (see Figure 1) with varying cluster sizes with an asynchronous-based communication framework, with no constraints on communication delays and exponentially distributed intervals between message transmissions. This model supports first in first out (FIFO), and thus messages sent are received in the order sent. The system model restricts a process, P_i, to only send messages to its clockwise neighbour and receive from its anticlockwise neighbour.

Thus, for each process P_i, where $0 \le i \le N-1$ and n is the number of processes in the cluster, the clockwise neighbour (CN_i) is defined as the process immediately following P_i, $CN_i = P_{i+1}$ or $CN_i = P_0$ if $i = \mathrm{N} - 1$. Conversely, the anticlockwise neighbour of P_i (ACN_i) is defined as the process immediately preceding P_i, $ACN_i = P_{i-1}$ or $ACN_i = P_{N-1}$ if $i = 0$. Therefore, messages are transmitted exclusively in the clockwise direction, with P_i receiving from ACN_i and transmitting to CN_i in a daisy chain framework. We also defined the Stability clock of any process (SC_i) as the largest timestamp, ts, known to any process P_i as stable. When a message m becomes stable, SC_i is updated as follows: $SC_i = \max\{SC_i, m\_ts\}$. Additionally, we introduce the definition of $Hops_{i,j}$ which is defined as the number of hops between any two processes from $\mathrm{P_i}$ to $\mathrm{P_j}$ in the clockwise direction: (i) $\mathrm{Hops_{i,j}} = 0$, if $\mathrm{i} = \mathrm{j}$. (ii) $\mathrm{Hops_{i,j}} = (\mathrm{j} - \mathrm{i})$, if $\mathrm{j} > \mathrm{i}$, and (iii) $\mathrm{Hops_{i,j}} = (\mathrm{j} + \mathrm{N} - \mathrm{i})$ if $\mathrm{j} < \mathrm{i}$.

3. Daisy Chain Total Order Protocol- DCTOP

The DCTOP system employs a group of interconnected process replicas, with a group size of $\boldsymbol{N}$, where $\boldsymbol{N}$ is an odd integer, $\boldsymbol{N} \ge \mathbf{3}$ 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 utilizing 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 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 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 $\mathrm{P_i}$ as stable. Initially, SC_i is zero.
3. **Message Buffer** $(mBuffer_i)$: This field holds the sent or received messages by $\mathrm{P_i}$.
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.

M is used to denote all types of messages used by the protocol. Usually there are 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 μ(m) when it is bound to m. μ(m) is used to announce that m has been received by all processes in Π. The relationship between m and its counterpart μ(m) is shown in Figure 2.

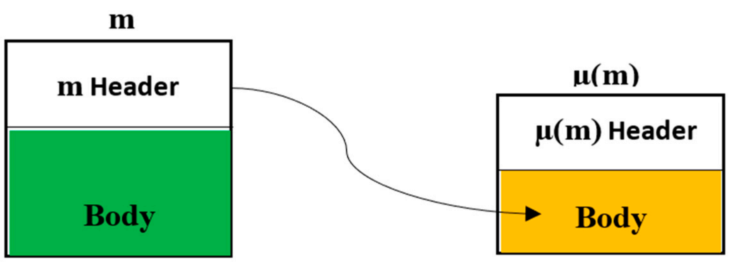


Figure 2. Relationship between m and μ(m).

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 μ(m), which contains the information from m's header. This is why we refer to μ(m) instead of just μ. μ(m) has m header information as its main information and does not contain its own data; therefore, the body of μ(m) is essentially m's header (see Figure 2).

A message M has at least the following data structures:

1. **Message origin** (M_origin) field shows the id of the process in $\Pi = \{\mathrm{P_0}, \mathrm{P_1}, \mathrm{P_2}, \ldots, \mathrm{P_{N-1}}\}$ 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.

3.2. DCTOP Principles

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: 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 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 $\mathrm{m\_ts} = \mathrm{LC_i}$, increments LC$_i$ by one afterwards, and then places the timestamped message into the OutgoingQueue$_i$ for transmission. A copy of the message is also stored in mBuffer$_i$ for local record.

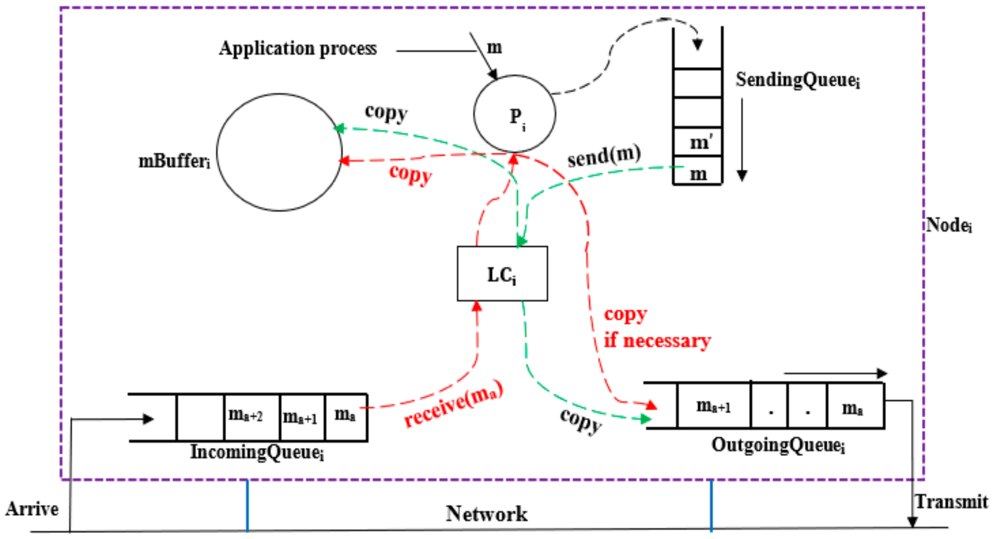


Figure 3. Message Sending, Receiving, Forwarding.

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 $\mathrm{LC_i}$ as $\mathrm{LC_i} = \max\{(\mathrm{m\_ts}+1), \mathrm{LC_i}\}$, and delivers the message to process P$_i$ for further handling. Typically, m is entered in $\mathrm{mBuffer_i}$ and may be forwarded if necessary to $\mathrm{CN_i}$ by entering a copy of m with destination set to $\mathrm{CN_i}$ into the $\mathrm{OutgoingQueue_i}$. If necessary, meaning the message has not yet completed a full cycle around the ring, it may be forwarded to the CN$_i$ by placing a copy of it in the OutgoingQueue$_i$ with its destination set to $\mathrm{CN_i}$. However, once the message has completed a full cycle within the ring network, it is no longer forwarded, and the forwarding process stops.

If two messages are received consecutively, they are sent in the same order but not necessarily immediately after each other, depending on the transmission control policy. As shown in Figure 3, 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.

(2) Timestamp Stability: A *message timestamp TS*, $TS \geq 0$, is said to be *stable* in a given process $\mathrm{P_i}$ if and only if the process $\mathrm{P_i}$ is guaranteed not to receive any m', $m'\_ts \leq \mathrm{TS}$ any longer.

Observations:

1) A timestamp $\mathrm{TS'} < \mathrm{TS}$ is also stable in $\mathrm{P_i}$ when TS becomes stable in $\mathrm{P_i}$.
2) The term "stable" is used to refer to the fact that once TS becomes stable in $\mathrm{P_i}$, it remains stable for ever. This usage corresponds to that of "stable" property used by Chandy and Lamport [24]. Therefore, the earliest instance when a given TS becomes stable in $\mathrm{P_i}$ will be the interest in the later discussions.
3) When TS becomes stable in $\mathrm{P_i}$, the process can potentially total order (TO) deliver all received but undelivered $m, m\_ts \leq$ TS, because stability of TS eliminates the possibility of $\mathrm{P_i}$ 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 $\mathrm{P_i}$ when $\mathrm{P_i}$ knows that m has been received by at least $(f+1)$ processes. 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.

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 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 destination, 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 ≤ SC_i to DQ_i. Then, it forms μ(m), sets the two header fields, μ(m)_origin=P_i, μ(m)_destn=CN_i and deposit 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 ≤ 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, μ(m)_destn = CN_i and deposit μ(m) in $OutgoingQueue_i$.
   - However, if *m in* μ(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 ≤ SC_i to DQ_i.
   - If $P_j = CN_i$, P_i *ignores* μ(m) otherwise, it sets μ(m)_destn = CN_i and deposit 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 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*': (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*').

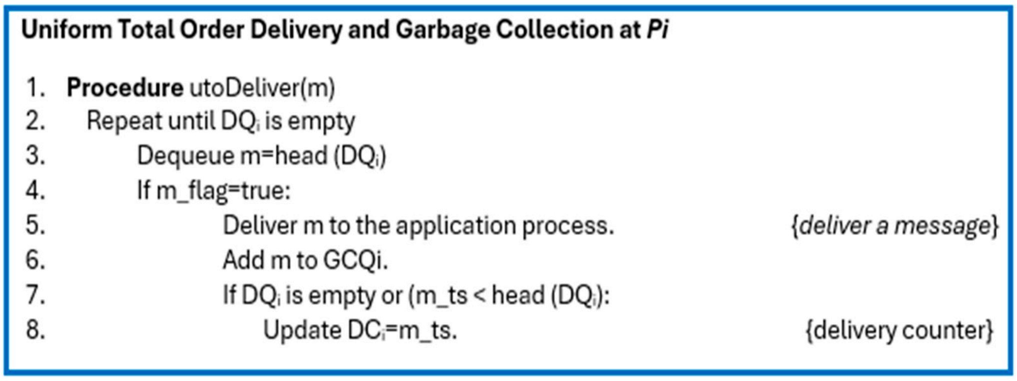
3.4. DCTOP Delivery Requirements

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 $N - 1 > i$.

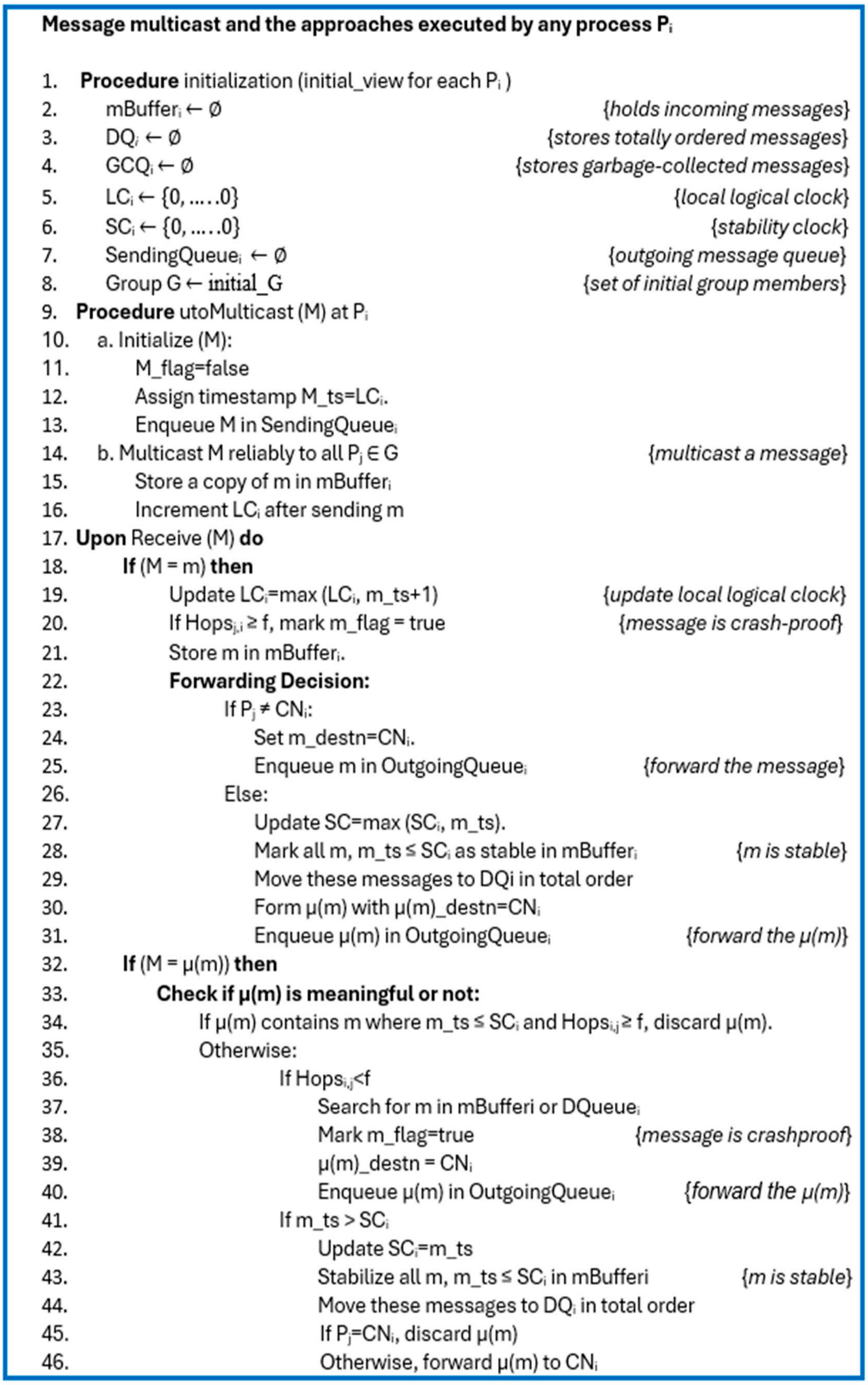
In summary, the pseudocode representations for total order message delivery, message communication, and membership changes are presented in Figures 4, 5, and 6 respectively.



```
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}
```

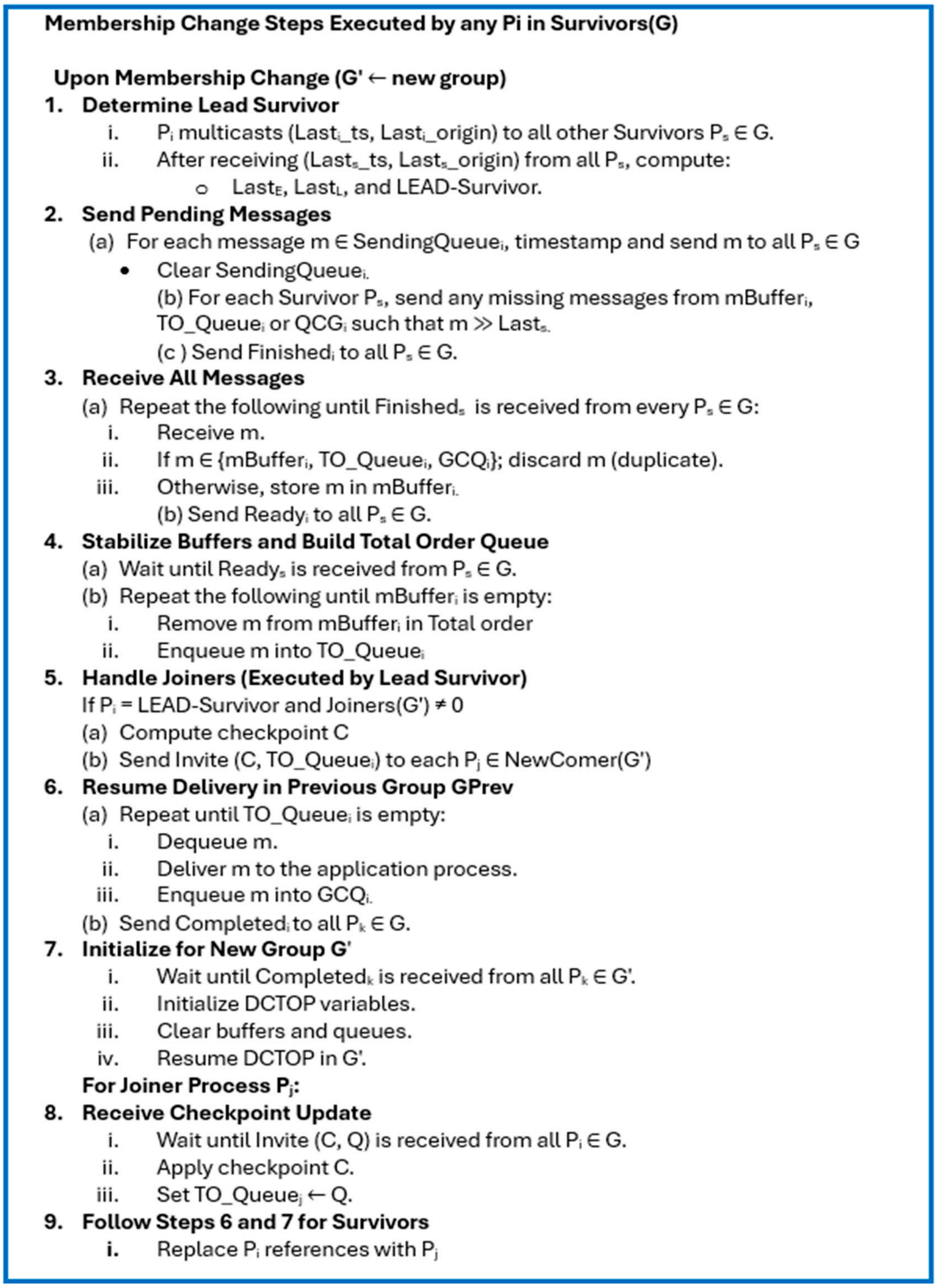
Figure 4. TO_Delivery Algorithm of DCTOP.



```
Message multicast and the approaches executed by any process P_i

1.  Procedure initialization (initial_view for each P_i )
2.      mBuffer_i ← Ø                                        {holds incoming messages}
3.      DQ_i ← Ø                                             {stores totally ordered messages}
4.      GCQ_i ← Ø                                            {stores garbage-collected messages}
5.      LC_i ← {0, ..…0}                                     {local logical clock}
6.      SC_i ← {0, ..…0}                                     {stability clock}
7.      SendingQueue_i ← Ø                                   {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 ∈ 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} ≥ f, mark m_flag = true            {message is crash-proof}
21.         Store m in mBuffer_i.
22.         Forwarding Decision:
23.             If P_j ≠ 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 ≤ SC_i as stable in mBuffer_i     {m is stable}
29.                 Move these messages to DQi in total order
30.                 Form μ(m) with μ(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 ≤ SC_i and Hops_{i,j} ≥ 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.                     μ(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 ≤ 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 of 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:
              o  $Last_E$, $Last_L$, and LEAD-Survivor.
2. Send Pending Messages
    (a) For each message m ∈ $SendingQueue_i$, 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 ∈ {$mBuffer_i$, $TO\_Queue_i$, $GCQ_i$}; 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 $TO\_Queue_j$ ← Q.
9. Follow Steps 6 and 7 for Survivors
      i.   Replace $P_i$ references with $P_j$
```

Figure 6. Membership Changes of DCTOP.

3.5. Group Membership Changes

The DCTOP protocol is built on top of a group communication system [26,27]. 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 always true. The *membership change procedure* is detail in Figure 6. 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 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 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 work 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 perform TO delivery of messages in Gprev and then resume operations in the new G' of 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.

3.6. 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 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 steps 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 stable (Line 28 of Figure 5). This happens because when 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 $\text{LC}_i > m_i\text{\_ts}$ afterward. 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 $\text{Hops}_{ij} \geq f$, then m_i is crashproof and it sets m_i_flag=true. The ACN_i sets m_i 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 utoDeliver m_i (Lines 1 to 8 of Figure 4) if m_i is at the head of DQ. ACN_i generates, timestamp μ(m_i) using its LC and then sends it to its own CN. Similarly, μ(m_i) is forwarded along the ring (Line 31 of Figure 5) until the ACN of μ(m_i)_origin receives μ(m_i). When any process receives μ(m_i) and $\text{Hops}_{ij} < f$, it knows that m_i is crash proof and stable but if $\text{Hops}_{ij} \geq f$, 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 μ(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 μ(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 μ(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 equal timestamp, message from a higher origin is ordered before message from a lower origin.). When P_i receives μ(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 μ(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 μ(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 is 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 crash proof and during 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 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 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, GCQ_i before executing line 6a(ii) of Figure 6. Since all correct processes exchange their $mBuffer_i$, TO_Queue_i, 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 crashproof and stable. Remember, when any process receives $\mu(m_i)$ and $Hops_{ij} < f$, it knows that m_i is crash proof and stable 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 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 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 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 $\mu(m)$ 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 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)

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

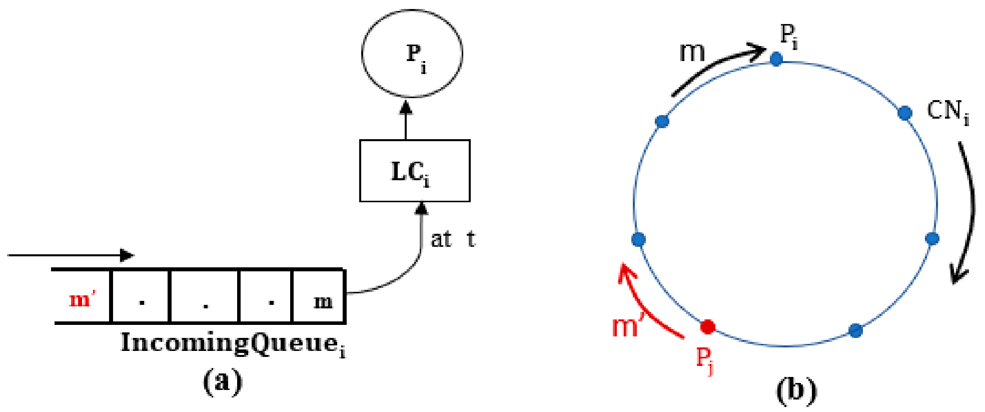


Figure 7. Example Contradicting Lemma 1.

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 7b. Given that m'_ts $\le$ 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 $P_{j+1}, P_{j+2}, \ldots P_{i-1}$ 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 6b. Since P_i is the last process to receive m in the system, P_j must have received m before t; since m'_ts $\le$ m_ts, P_j could not have sent m' after receiving m. So, the only possibility for m'_ts $\le$ 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.

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 $SendingQueue_i$ 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 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 $\mathrm{P_i}$ forwarded ever since it last sent its own message. Obviously by definition, as soon as the process $\mathrm{P_i}$ sends a message, the $forwardlist_i$ is empty. Therefore, if $\mathrm{P_i}$ forwards a message that originates from the process $\mathrm{P_{i-1}}, i > 0$, which was initially in its $IncomingQueue_i$, then process $\mathrm{P_i}$ will contain the process $\mathrm{P_{i-1}}$ in its forward list, and whenever it sends a message the process $\mathrm{P_{i-1}}$ 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 [19] protocol and Raft [9,18] a widely implemented, leader-based ordering protocol by evaluating latency and throughput across varying numbers of messages transmitted within the cluster environment. Java (OpenJDK-17, Java version 17.02) framework was used to run a discrete event simulation for the protocols with at most 9 processes, $N = 4,5,7, and\ 9$.

Every simulation method made use of a common PC with a 3.00GHz 11th Gen Intel(R) Core(TM) i7-1185G7 Processor and 16GB 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 a neighbour receives a message, it passes it on to another neighbour until all processes have done so. When the ACN of the message origin receives the message then it knows that it is stable and makes an attempt to deliver it in total order, a process known as *TO delivery*. This process then notifies all other processes which, up until this point, had no idea of the message's status by using an acknowledgement message known as μ-message to inform them of the message's stability. Other processes that get this acknowledgement are aware that the message is stable and make an effort to deliver it in total sequence. 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 varying numbers of process replicas, such as 4, 5, 7, and 9 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 seconds. 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 (1,000k) and twenty-five million (25,000k) 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, denoted as $TOdeliver(m)$, to the applications process. For example, let t_0 and t_1 be the time when P_0 sends a message to its $\mathrm{CN_0}$ and the time when the ACN($\mathrm{ACN_0}$) delivers that message in total order respectively. Then $\mathrm{t_1 - t_0}$ 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%. 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.

All experiments were done independently to prevent any inadvertent consequences of running multiple experiments simultaneously. Nevertheless, the execution ensured each of the experiments was staggered to cover approximately the same amount of simulation time. This was done to sustain a uniform load on the ring-based and leader-based network across all of the experiments.

5.1. Results and Discussion

The latency analysis of DCTOP, LCR, and RAFT (see Figure 8i-iv) across varying group sizes (N = 4, 5, 7, and 9) and increasing message volumes reveals that DCTOP consistently demonstrates the lowest latency. This performance advantage is likely attributed to its use of Lamport logical clocks for efficient sequencing of concurrent messages, the assignment of a unique last process for each message originator, and a relaxed crash-failure assumption that permits faster message delivery. LCR shows moderately increasing latency with larger group sizes and message volumes, primarily due to its reliance on vector clocks whose size grows with the number of processes and the use of a globally fixed last process for message ordering, both of which contribute to increased message size and coordination cost. RAFT, a leader-based protocol, exhibits the highest latency overall, especially under higher load conditions, underscoring the limitations of centralized coordination. However, under lower traffic conditions (e.g., 1 million messages per process), RAFT performs competitively and, in configurations with N = 7 and N = 9, even outperforms DCTOP and LCR. This suggests that RAFT may remain suitable in low-load or moderately scaled environments.

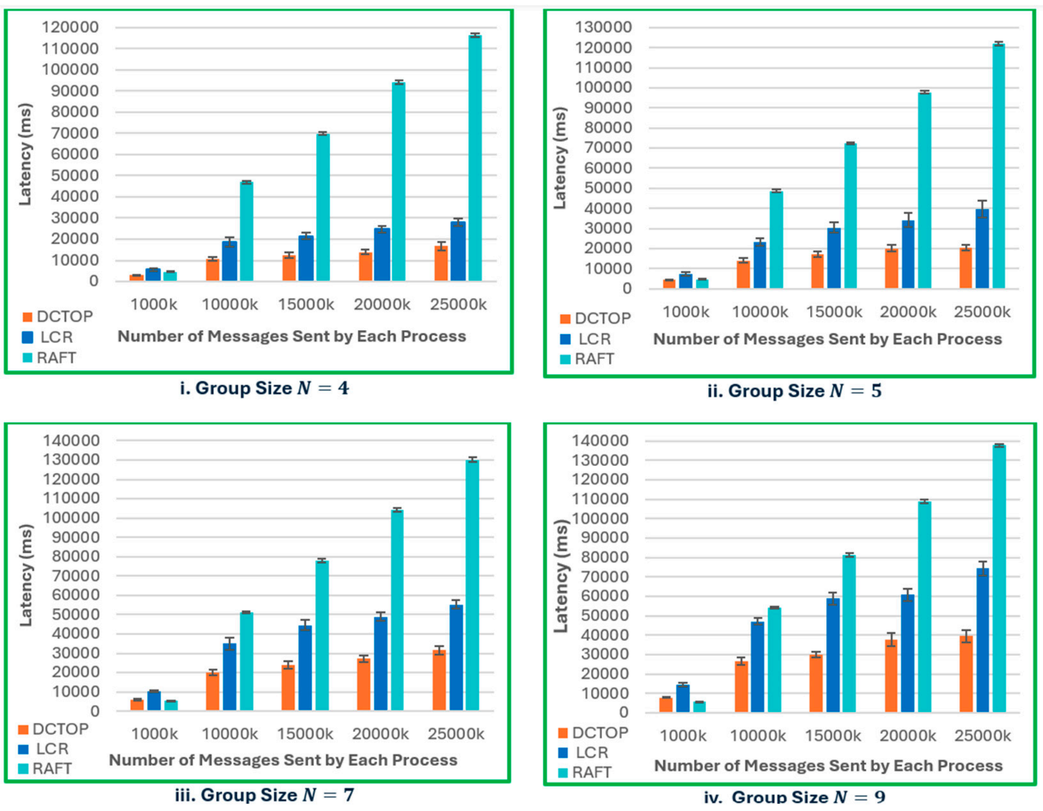


Figure 8. Latency Comparison.

On the other hand, in terms of throughput (Figure 9i-iv), DCTOP and LCR outperform RAFT across all group sizes and message volumes. Both are leaderless ring-based order protocols that benefit from decentralized execution, enabling all processes to independently receive and process client requests. This results in cumulative throughput, which scales linearly with group size. RAFT, by contrast, centralizes request handling at the leader, thereby limiting throughput to the leader's processing capacity. The results demonstrate that DCTOP consistently achieves the highest throughput, with LCR following closely despite its minor overhead from vector timestamps and centralized 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 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.

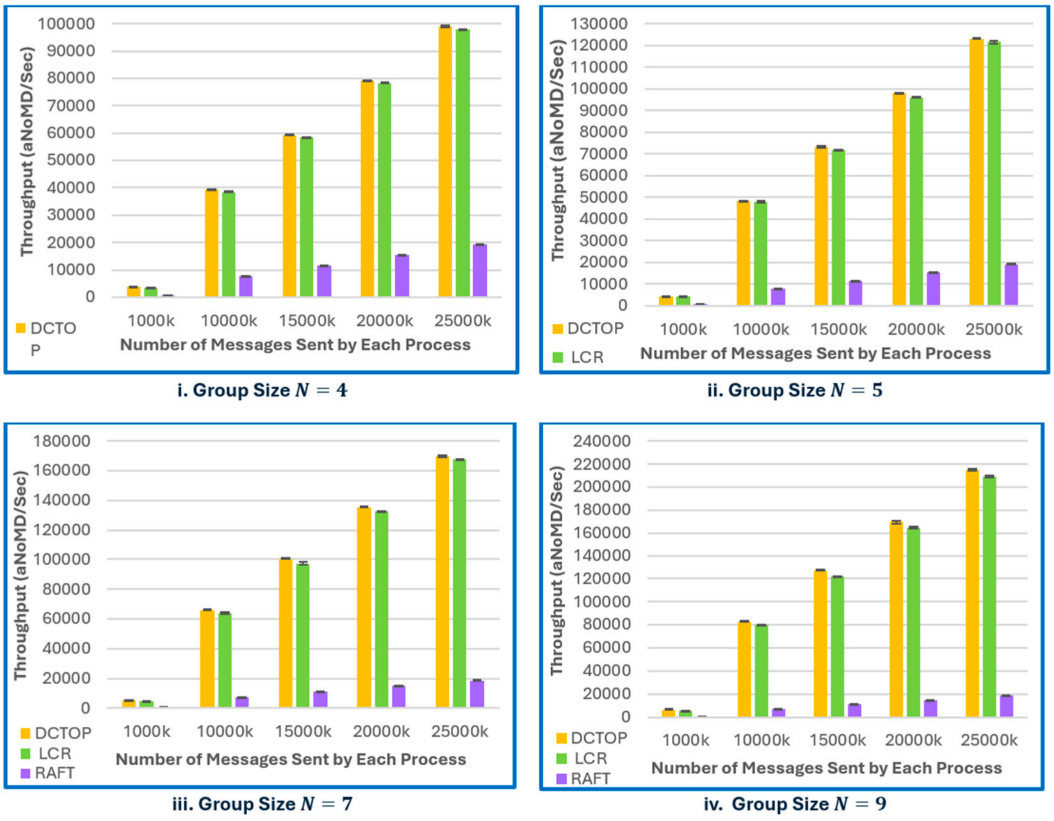


Figure 9. Throughput Comparison.

6. Conclusions and Future Work

In this work, DCTOP, a novel ring-based leaderless total order protocol that extends the traditional LCR approach was introduced through three key innovations: the integration of Lamport logical clocks for concurrent message sequencing, a new mechanism for dynamically identifying the last process per each message sender, and a relaxation of the traditional crash failure assumption. These modifications collectively contributed to significant latency reductions under varying system configurations. To promote fairness among process replicas, our simulation model incorporated control primitives to eliminate message-sending bias. A comparative performance evaluation of DCTOP and LCR using discrete-event simulation across group sizes of N = 4, 5, 7, and 9, and under concurrent message loads was conducted. The results yielded three major insights. First, DCTOP achieved over 43% latency improvement compared to LCR across all configurations, demonstrating the efficacy of Lamport logical clocks in this context. Second, the proposed dynamic last process mechanism proved to be an effective alternative to the globally fixed last process used in LCR, enabling faster message stabilization. Third, by relaxing the LCR crash tolerance condition from N =

f + 1 to N = 2f + 1, DCTOP is able to deliver messages more quickly while still tolerating failures, further contributing to latency reduction. While our primary focus was on evaluating DCTOP relative to LCR, we included RAFT, a widely adopted leader-based protocol as a benchmark to contextualize our results. RAFT demonstrated competitive performance under light message loads, but its throughput and latency plateaued with scale due to its centralized coordination model. The goal was not to critique RAFT, but to highlight architectural differences and situate DCTOP within the broader spectrum of total order protocols.

Given that the primary goal of this study was to investigate the initial performance characteristics of DCTOP, we deliberately limited our evaluation to small group sizes ($N \leq 9$) to enable controlled experimentation and isolate protocol-level behaviour. While this approach provides useful insight, it also introduces some limitations. The current implementation does not model process or communication failures, which are common in practical distributed systems. Furthermore, larger-scale deployments may exhibit additional performance dynamics not captured in this setting. As part of our ongoing work, we are developing a cloud-based, fault-tolerant implementation of DCTOP to validate these findings in more realistic environments, including under failure conditions and dynamic workloads.

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
CN	Clockwise Neigbhour
ACN	Anti-Clockwise Neigbhour
SC	Stability Clock
DQ	Delivery Queue
DCQ	Garbage Collection Queue
UTO	Uniform Total Order (uto)

References

1. A. Choudhury, G. Garimella, A. Patra, D. Ravi, and P. Sarkar, "Crash-tolerant consensus in directed graph revisited." In *International Colloquium on Structural Information and Communication Complexity* (pp. 55-71). Cham: Springer International Publishing.
2. M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *Journal of the ACM (JACM),* vol. 27, no. 2, 1980, pp. 228-234, https://doi.org/10.1145/322186.322188.
3. E. W. Vollset, and P. D. Ezhilchelvan, "Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in manets." In 24th IEEE Symposium on Reliable Distributed Systems, pp. 166-175.
4. M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems." In In Proceedings of the 2012 USENIX conference on Annual Technical Conference, pp. 453-466.
5. M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems." pp. 464-474, doi: 10.1109/ICDCS.2000.840959.
6. A. A. Helal, A. A. Heddaya, and B. B. Bhargava, *Replication techniques in distributed systems*: Springer Science & Business Media, 2006.
7. X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Computing Surveys (CSUR),* vol. 36, no. 4, pp. 372-421, 2004, https://doi.org/10.1145/1041680.1041682.

8. D. Ongaro, and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," Tech Report. May, 2014. http://ramcloud. stanford. edu/Raft. pdf, (Accessed on June 6, 2024).
9. F. Junqueira, and B. Reed, *ZooKeeper: distributed process coordination*: " O'Reilly Media, Inc.", 2013.
10. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free Coordination for Internet-scale Systems." In Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIXATC'10)
11. M. Burrows, "The Chubby lock service for loosely-coupled distributed systems." In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06), pp. 335-350.
12. A. Ejem, P. Ezhilchelvan,: Design and Performance Evaluation of High Throughput and Low Latency Total Order Protocol. In: 38th Annual UK Performance Engineering Workshop (2022)
13. J. Pu, M. Gao, and H. Qu, "SimpleChubby: a simple distributed lock service.", https://www.scs.stanford.edu/14au-cs244b/labs/projects/pu_gao_qu.pdf, (Accessed on May 31, 2024).
14. L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51-58, 2001.
15. D. Ongaro, and J. Ousterhout, "In search of an understandable consensus algorithm." In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14), pp. 305-319.
16. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system." *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, doi: 10.1109/MSST.2010.5496972, pp. 1-10.
17. A. Ejem, P. Ezhilchevan,: Design and performance evaluation of raft variations. In: 39th Annual UK Performance Engineering Workshop (2023)
18. R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Transactions on Computer Systems (TOCS),* vol. 28, no. 2, pp. 1-32, 2010, https://doi.org/10.1145/1813654.1813656.
19. Moraru, David G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, 2013, 358–372. https://doi.org/10.1145/2517349.2517350
20. M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication." *2012 IEEE 31st Symposium on Reliable Distributed Systems*, 2012, pp. 111-120, doi: 10.1109/SRDS.2012.66.pp. 111-120.
21. 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.
22. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, and J. Donham, "Storm@ twitter." pp. 147-156, In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14), https://doi.org/10.1145/2588555.2595641.
23. M. Chandy, and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS),* vol. 3, no. 1, pp. 63-75, 1985, https://doi.org/10.1145/214451.214456
24. 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, (Acessed online on 07/04/2024).
25. Birman, and T. Joseph, "Exploiting virtual synchrony in distributed systems." In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87), pp. 123–138. https://doi.org/10.1145/41457.37515.

26. Y. Amir, and J. Stanton, *The spread wide area group communication system*. Johns Hopkins University. Center for Networking and Distributed Systems:[Technical Report: CNDS 98-4]. 1998, (Accessed on April 28, 2024).
27. Ejem A., Njoku C. N., Uzoh O. F., Odii J. N, "Queue Control Model in a Clustered Computer Network using M/M/m Approach," *International Journal of Computer Trends and Technology (IJCTT)*, vol. 35, no. 1, pp. 12-20, 2016. https://doi.org/10.14445/22312803/ IJCTT-V35P103