Distributed Systems Algorithms - Distributed Ledger

Gonçalo Virgínia[*]
g.virginia@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Ricardo Rodrigues[†]
rf.rodrigues@campus.fct.unl.pt
MEI, DI, FCT, UNL

ABSTRACT

Distributed ledgers (DLTs) are decentralized databases maintained and updated across multiple nodes in a distributed system, relying on consensus algorithms to ensure agreement, ordering of events, and synchronization among participants - combating potential failures or inconsistencies in the network, and, most importantly, eliminating the need for centralized coordination.
This makes DLTs suitable for a wide range of applications, including data sharing, asset tracking, and collaborative systems. Nonetheless, although this approach offers robustness and fault tolerance, it naturally also presents its own challenges with respect to latency, throughput, and scalability, which can be mitigated through multiple approaches.
To this end, this project employs two different approaches consisting of mainstream consensus algorithms: the relatively simple ABD Quorum, and, a more complex State Machine + Multi-Paxos stack (with two Multi-Paxos variants: Classic, and Distinguished Learner).

ACM Reference Format:
Gonçalo Virgínia and Ricardo Rodrigues. 2024. Distributed Systems Algorithms - Distributed Ledger. In *The Projects of ASD - first delivery, 2024, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The following sub-sections describe the abstracted ideas for both previously mentioned approaches.

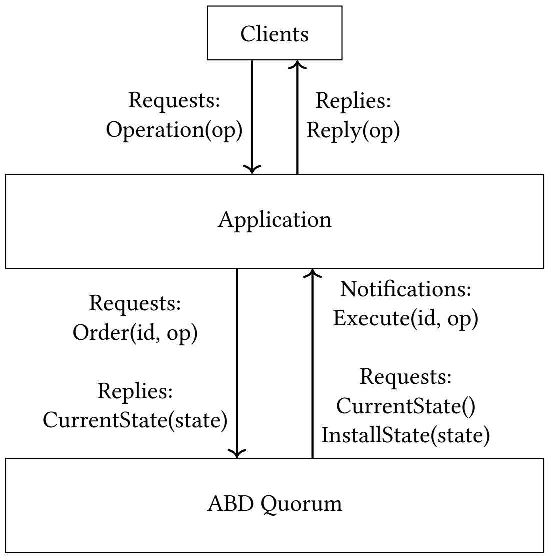
1.1 ABD Quorum

The ABD Quorum algorithm is a relatively protocol for maintaining consistency and fault tolerance, by ensuring that both read and write operations are performed reliably by requiring a majority quorum of replicas to participate. A write operation involves sending a value, along with a unique "timestamp" - or "tag" consisting of an (opSeq, processId) tuple - to all other replicas. Each replica updates its stored value and timestamp only if the incoming write is more recent. For a read operation, a client queries a quorum of replicas to determine the most up-to-date value based on the highest timestamp/tag. Finally, the replica originally contacted by the client then propagates this value to ensure it is written back to the replicas, maintaining consistency.

Operation "timestamp" (Tag) comparisons take the following form: <s1, i1> > <s2, i2> if s1 > s2 or (s1 = s2 and i1 > i2) - meaning, the operation sequence number always takes precedence in terms of importance, as would be expected, but, in the case of two operations have the same opSequence number, the highest process ID wins the transaction.

The algorithm relies on these intersections between read and write quorums to guarantee that the most recent data is always accessible, preventing stale reads. This method is ensured to work as long as a quorum receives replies from (numReplicas / 2) + 1 replicas, given that, at the bare minimum, a valid subsequent quorum will intersect at least 1 replica from the previous quorum, which will contain the most up-to-date value.

As expected, the protocol stack is fairly trivial - consisting solely of the ABD Quorum protocol for both replication and consistency of the Application's state:



Clients send their requests to the Application layer, whose sole job is to send operations down to the ABD Quorum algorithm, for both replicating writes, and maintaining the most up-to-date state for any read - consequently, after an operation passes a majority of replicas via the ABD protocol, a reply is sent upwards, confirming

[*]Student number 56773
[†]Student number 72054

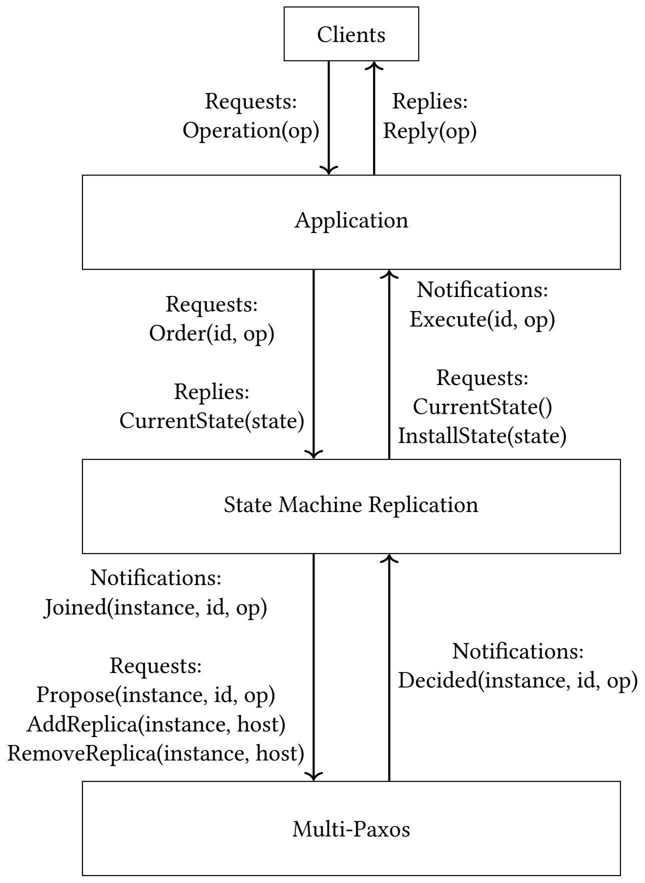
ASD24/25, 2024, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the execution of a given operation, and thus, updating the Application's state - which, finally, enables a correct reply to the original client.

1.2 State Machine + Multi-Paxos

The State Machine Replication (SMR) protocol is responsible for managing the execution and replication of client operations within the distributed system. It handles client requests by ensuring their operations are replicated and ordered consistently through the agreement protocol. The SMR tracks the sequence of commands, determines the decided operations for each position in the sequence, and notifies the application to execute these operations in the correct order.

Additionally, the protocol oversees the communication channel, including maintaining open TCP connections and managing membership changes in the system. This involves facilitating the addition of new replicas to the replica set, notifying the agreement protocol about membership updates, and detecting replica failures. Upon detecting a failure, the SMR protocol attempts to reconnect with the failed replica a configurable number of times. If the replica is deemed non-functional, the protocol issues a state machine operation to remove it from the replica set.


Clients
Requests: Operation(op)
Replies: Reply(op)
Application
Requests: Order(id, op)
Replies: CurrentState(state)
Notifications: Execute(id, op)
Requests: CurrentState() InstallState(state)
State Machine Replication
Notifications: Joined(instance, id, op)
Requests: Propose(instance, id, op) AddReplica(instance, host) RemoveReplica(instance, host)
Notifications: Decided(instance, id, op)
Multi-Paxos

State Transfer: When a new replica joins the system, it must synchronize with the current state of the system. This process, known as state transfer, is managed by the SMR protocol. The protocol requests the current application state from the application and, upon receiving it, transmits this state along with information about the current sequence position to the new replica. To optimize this process, instead of involving all replicas, the replica that received the join request can respond to the new replica with the necessary state information. This requires the SMR protocol to exchange messages with other processes over the network.

Independence from the Agreement Protocol below: The implementation of the SMR protocol is independent of the underlying agreement protocol, with a few exceptions. For instance, when using Multi-Paxos, the SMR protocol must be informed about the current leader since only the leader proposes commands. In such cases, client requests received by non-leader replicas must be forwarded to the leader. If the leader changes, any unprocessed requests must be resubmitted to the new leader. Conversely, when using ABD, each instance of the protocol may require the SMR protocol to provide the system membership information for that instance.

Multi-Paxos Agreement Protocol: The Multi-Paxos agreement protocol determines the order of operations to be executed in the system. This protocol does not run multiple instances concurrently, due to the complexity of ensuring correctness, instead, every call from the State Machine is done one at-a-time. In accordance to the Multi-Paxos specification, only the leader proposes commands for agreement. Other replicas forward client requests to the leader, and participate in instances as instructed by the leader's proposals. If a leader change occurs, pending client requests must be resubmitted to the new leader for ordering.

Careful management of system membership is critical in Multi-Paxos. Replicas must maintain an up-to-date view of the system membership to compute majorities accurately. Furthermore, a replica does not participate in instances from which it is excluded, either before joining the system, or after being removed from it. By adhering to these rules, the Multi-Paxos protocol ensures consistent and reliable agreement on the sequence of operations.

2 PSEUDO-CODE

The following sub-sections describe in detail the pseudo-code for the provided Application Layer, and, most importantly, our concrete implementations for the ABD Quorum, State Machine, Multi-Paxos (Classic), and Multi-Paxos (Distinguished Learner) algorithms.

2.1 Application

The top-level Application is relatively trivial, as it's simply responsible for direct request and response handling between a client and an abstracted interface of the implemented DLT protocol, in the following flow: Client -> App -> DLT -> App -> Client.
The Application can handle concurrent requests from clients, managing them via the data and clientIdMapper Maps, which await responses from the underlying DLT layer.
Some separate request and notification handlers are implemented purely for the ABD protocol, given that operations are not yet abstracted via the SMR protocol, these handlers are: WriteComplete,

ReadComplete, and UpdateValue.
To aid in understanding the implementation of this protocol, we present the pseudocode for this layer in the following block:

```
Interface:
    Requests:
        Operation(op, host)
        CurrentState()
        InstallState(state)
        Execute(opId, op)
    Indications:
        CurrentState(state)
        Read(opId, opKey)
        Write(opId, opKey, opData)

State:
    executedOps        ▷ Index for the most recently executed operation
    data               ▷ Map (ID -> data) for storing data sent in requests
    cumulativeHash
    clientIdMapper               ▷ Map (opID -> (cID, op)) containing client
operations
    strategy     ▷ Currently enabled replication strategy (ABD or SMR)

Upon Init(s) do:
    executedOps ⟵ 0
    data ⟵ {}
    clientIdMapper ⟵ {}
    cumulativeHash ⟵ {}
    strategy ⟵ s

Upon CurrentState() do:
    state ⟵ (executedOps, cumulativeHash, data)
    Trigger CurrentState(state)

Upon InstallState(state) do:
    Call InstallStateProcedure(state)

Upon Operation(op, host) do:
    opUUID ⟵ generateUUID()
    clientIdMapper[opId] ⟵ (host, op.opId,)
    if (strategy = SMR) then
        Trigger Order(opUUID, op.opType, op.key, op.data)
    else
        if (op.opType = READ) then
            Trigger Read(opUUID, op.key)
        else
            Trigger Write(opUUID, op.key, op.data)

Upon Execute(opId, op) do:
    cumulativeHash ⟵ cumulativeHash ∪ op.data
    if (op.type = WRITE) then
        data[op.key] ⟵ op.data
    executedOps ⟵ executedOps + 1
    (client, opId) ⟵ clientIdMapper[opId]
    clientIdMapper ⟵ clientIdMapper  opId
    if ((client, opId) = ⊥) then
        return
    if (op.type = WRITE) then
        Trigger Send(REPLY, (opId, ⊥), client)
    else
        Trigger Send(REPLY, (opId, op.data), client)

Upon WriteComplete(k, v, opId) do:
    data[k] ⟵ v
    (client, opId) ⟵ clientIdMapper[opId]
    clientIdMapper ⟵ clientIdMapper  opId
    Trigger Send(REPLY, (opId, ⊥), client)
    Call UpdateOperationCount()

Upon ReadComplete(k, v, opId) do:
    data[k] ⟵ v
    (client, opId) ⟵ clientIdMapper[opId]
    clientIdMapper ⟵ clientIdMapper  opId
    Trigger Send(REPLY, (opId, v), client)
    Call UpdateOperationCount()

Upon UpdateValue(k, v) do:
    data[k] ⟵ v
    Call UpdateOperationCount()

Upon UpdateOperationCount() do:
    executedOps ⟵ executedOps + 1
    if (executedOps % 10000 = 0) then
        cumulativeHash ⟵ ComputeDataHash() ▷ Computes hash of all
keys currently stored in data Map
```

2.2 ABD Quorum

2.2.1 Initialization. The ABD Quorum protocol, although straightforward, also maintains a decent amount of state.

Starting off, each node needs to maintain its own unique ID, as well as the membership of all active nodes currently in the system, in order to send quorum messsages, and check if the number of replies surpasses the majority threshold.

Next up, the protocol maintains the index for the most up-to-date executed operation, which serves two main purposes: discarding messages from older operations, and storing/sending Tags for a given operation/quorum.

Naturally, we also store a set of quorum replies for the current operation, which basically consist of (Tag, opValue) pairs, in order to obtain the most recent value at the end of a quorum.

For every operation, a map of opKey -> Tags is also used, in order to store the most recent Tag for a given operation, alongside a map of opKey -> opValues, and, a map of opKey -> OpId (opKey-opValue pairs are the actual operation, at the application level, while opId is merely an identifier for an operation between the App and ABD layers).

Finally, pending merely stores a write value pending the completion of the current majority quorum, and is discarded after the write completes.

```
Interface:
    Requests:
        CurrentState(state)
        Read(opId, opKey)
        Write(opId, opKey, opData)
    Indications:
        CurrentState()
        InstallState(state)
        Execute(opId, op)

State:
    thisHost                      ▷ This node's host information
    thisProcessId                 ▷ This node's ID
    membership                    ▷ Set of node's currently in membership
    opSeq                         ▷ Current operation index
    quorumReplies ▷ Set of quorum replies regarding current operation
    tags                          ▷ Map of key -> (opSeq, processId)
    values                        ▷ Map of key -> opData
    operations                    ▷ Map of key -> opID
    pending   ▷ Operation data pending to be written in current opSeq

Upon Init(h, initialMembership) do:
    thisHost ← h
    membership ← initialMembership
    thisProcessId ← membership.indexOf(h)
    opSeq ← 0
    quorumReplies ← {}
    tags ← {}
    values ← {}
    operations ← {}
    pending ← ⊥
```

2.2.2 *Read Operations.* Concerning read operations - the process is extremely streamlined: after the protocol receives a Read request from the App, it increments the current opSequence index, empties the quorumReplies set, and sets pending to null (given that it's a read operation, no value is pending to be written - this dynamic is subsequently used to facilitate some logic).

Furthermore, the opKey mappings are added to both the operations and tags maps, in order to use them for comparisons with quorum replies. Finally, the Read request handler terminates by sending a QUORUM_MSG to each peer in the membership. This message includes a ⊤ flag, signifying that it's a read operation, which enabled us to streamline both read and writes into one message handler.

Subsequently, peers then receive the QUORUM_MSG via the QuorumMsg handler, which takes care of fetching the local Tag for the opKey in the current message, and, seeing as the operation is a read, the execution goes into the if branch, also fetches the local value for the opKey, and sends both the local Tag and value inside the READ_REPLY to the original quorum sender.

Finally, the original quorum sender receives the aforementioned read replies via the ReadReplyMsg handler - which certifies that the message's opSequence is valid/current. Assuming it is, it checks if the pending value is null (which, as mentioned above, is used to facilitate some logic, including the reuse of quorumReplies for the next message's ACK's), and, if it is, adds the message to quorumReplies.

Once quorumReplies reaches the majority threshold, the maximum Tag is fetched from the set, pending is set to the value associated to the max Tag, opSeq is incremented, quorum replies emptied, and a WRITE_MSG is sent to all peers, starting a new quorum with the purpose of keeping every node with the most up-to-date value.

```
Upon Read(opId, opKey) do:
    opSeq ← opSeq + 1
    quorumReplies ← {}
    pending ← ⊥
    operations[opKey] ← (opId)
    tags[opKey] ← {(opSeq, thisProcessId)}
    foreach h ∈ membership do
        Trigger Send(QUORUM_MSG, (opSeq, opKey, ⊤), h)

Upon QuorumMsg((opSeq, opKey, isRead), h) do:
    tag ← tags[opKey]
    if (isRead = ⊤) then
        value ← values[opKey]
        Trigger Send(READ_REPLY_MSG, (opSeq, tag, opKey, value), h)
    else
        Trigger Send(READ_TAG_REPLY_MSG, (opSeq, tag, opKey), h)

Upon ReadReplyMsg((opSeq, tag, opKey, opValue), h) do:
    if (this.opSeq ≠ opSeq) then
        return
    if (pending = ⊥) then
        quorumReplies ← quorumReplies ∪ {(tag, opValue)}
    if (#quorumReplies = (#membership / 2) + 1) then
        maxTagQuorumReply ← MaxTagQuorumReply(quorumReplies)
        maxTag ← maxTagQuorumReply.tag
        pending ← maxTagQuorumReply.opValue
        opSeq ← opSeq + 1
        quorumReplies ← {}
        foreach h ∈ membership do
            Trigger Send(WRITE_MSG, (opSeq, opKey, maxTag, pending),
h)
```

2.2.3 *Write Operations.* Upon a write operation from the application, the Write handler is almost 1-to-1 compared to Read - it increments opSeq, empties quorumReplies, but - now pending is used to store the opData (whereas it was set to null on reads). Subsequently, the operations and tags mappings are exactly the same as on reads, and, the QUORUM_MSG's are sent with a ⊥ isRead flag.

Then, as mentioned previously, we reuse the QUORUM_MSG (and consequently **QuorumMsg** handler, on the peer's end, by simply using a boolean flag **isRead** to distinguish read and write quorums), which implies that the QuorumMsg handler goes into the else branch, sending a READ_TAG_REPLY_MSG with just the local Tag for the corresponding opKey.

This message is then handled by ReadTagReplyMsg, which, similarly to ReadReplyMsg, checks the validity of the message's opSeq, but, then checks if pending ≠ ⊥, basically checking if the majority quorum is still to be reached, and, if so, adds the quorum reply.
Eventually, the majority is reached, and the max Tag quorum reply is fetched (which, in this case is only used to derive the max Tag opSequence number), opSeq is incremented, quorumReplies emptied in preparation for ACK messages, and, a new Tag is created with the most recent opSequence number (higher than the max Tag), which basically means that all nodes will be updated with the current write value.
And, as anticipated, a WRITE_MSG is sent to every peer in a new quorum - setting pending to null, seeing as the write propagation is complete.

The other replicas then receive this write via the WriteMsg handler (which is used at the end of reads as mentioned prior), and, if the message's Tag for the corresponding opKey is superior to the local Tag for the same opKey (which, for original write operations, it's guaranteed to be, but, for originally read operations, it might not be), the tag and value relating to the opKey is replaced with the message's contents.
Subsequently, if the message came from another peer, the UpdateValue notification to the App layer is triggered. Then, an ACK_MSG is sent back to the original sender of the write quorum.

Upon receiving the ACK_MSG via the final AckMsg handler, it checks if the opSequence number is valid, adds the quorum reply to quorumReplies, and, when it reaches a majority, it empties the quorumReplies set, and verifies if pending = ⊥, which, by the logic explained prior, means the operation was originally a Write, or, a Read - triggering the corresponding operation completion to the Application layer.

```
Upon Write(opId, opKey, opData) do:
    opSeq ⟵ opSeq + 1
    quorumReplies ⟵ {}
    pending ⟵ opData
    operations[opKey] ⟵ (opId)
    tags[opKey] ⟵ {(opSeq, thisProcessId)}
    foreach h ∈ membership do
        Trigger Send(QUORUM_MSG, (opSeq, opKey, ⊥), h)

Upon ReadTagReplyMsg((opSeq, tag, opKey), h) do:
    if (this.opSeq ≠ opSeq) then
        return
    if (pending ≠ ⊥) then
        quorumReplies ⟵ quorumReplies ∪ {(tag, ⊥)}
    if (#quorumReplies = (#membership / 2) + 1) then
        maxTagOpSeq ⟵ MaxTagQuorumReply(quorumReplies).tag.opSeq
        opSeq ⟵ opSeq + 1
        quorumReplies ⟵ {}
        newTag ⟵ (maxTagOpSeq + 1, thisProcessId)
        foreach h ∈ membership do
            Trigger Send(WRITE_MSG, (opSeq, opKey, newTag, pending), h)
        pending ⟵ ⊥

Upon WriteMsg((opSeq, tag, opKey, opValue), h) do:
    thisTag ⟵ tags[opKey]
    if (tag > thisTag) then
        tags[opKey] ⟵ tag
        values[opKey] ⟵ opValue
        if (h ≠ thisHost) then
            Trigger UpdateValue(opSeq, opKey, opValue)
    Trigger Send(ACK_MSG, (opSeq, opKey), h)

Upon AckMsg((opSeq, opKey), h) do:
    if (this.opSeq ≠ opSeq) then
        return
    quorumReplies ⟵ quorumReplies ∪ {((opSeq, thisProcessId), pending)}
    if (#quorumReplies ≠ (#membership / 2) + 1) then
        return
    quorumReplies ⟵ {}
    if (pending = ⊥) then
        Trigger WriteComplete(opSeq, opKey, values[opKey], operations[opKey])
    else
        Trigger ReadComplete(opSeq, opKey, pending, operations[opKey])
```

2.3 State Machine

2.3.1 Initialization. There are two ways to join our system: either through the initial membership or by contacting a replica that is already part of the system. In our configuration, a replica that is not part of the system will request to join the system by contacting the first position of the initialMembership it receives as a parameter. The new replica will also have a timer, as it acts as a client during this process. If it is not added within a certain period, it will retry the connection. More details about this process will be elaborated in the AddReplica section of the state machine protocol.

On the other hand, during initialization, if a replica is part of the initialMembership, it will join the system immediately. The process with the highest processId (in our configuration, the process with the highest index in the initial membership list) will attempt to become the leader. As a result, even before receiving the first order from the HashApp, our system will already have a leader to propose requests to our agreement layer, which can be either the distinguished or Classic variant of Paxos.

2.3.2 Initialization and OrderRequest PseudoCode.

```
Interface:
    Requests:
        CurrentState(state)
        Read(opId, opKey)
        Write(opId, opKey, opData)
    Indications:
        CurrentState()
        InstallState(state)
        Execute(opId, op)

State:
    thisHost                ▷ This node's host information
    state                   ▷ JOINING or ACTIVE
    membership              ▷ Set of node's currently in membership
    nextInstance            ▷ Current operation index
    leader                  ▷ Leader node's host information
    previousLeader          ▷ Previous leader Candidate
    replcaIdSet             ▷ Hosts that contacted this replica to join
    pendingOrders           ▷ Pending Orders to be executed by leader
    pendingAddRemoves       ▷ Pending addRemove operations to be executed by leader, value pair of (host, boolean). True = Add ; False = Remove
    executedOperations      ▷ Orders to be executed by at the SMR Layer
    executedAddRemoves      ▷ Add/Remove operations executed at the SMR Layer
    hostTimers              ▷ (timerId, Host) to know the which Hosts timer triggered
    hostRetries             ▷ (Host, retriesNr) how many more connection retries for given Host

Upon Init(h, initialMembership) do:
    thisHost ⟵ h
    nextInstance ⟵ 1
    leader ← ⊥
    contact ← ⊥
    replicaIdSet ← {}
    pendingOrders ← {}
    pendingAddRemoves ← {}
    executedOperations ← {}
    executedAddRemoves ← {}
    hostTimers ← {}
    hostRetries ← {}
    **if** (thisHost ∈ initialMembership) **then**
        state ← ACTIVE
        membership ← initialMembership
        **Trigger Joined**(membership, membership.indexOf(thisHost), ⊤)
        firstLeader ← membership[#membership - 1]
        **if** (firstLeader = thisHost) **then**
            **Trigger Prepare**(nextInstance)
    **else**
        state ← JOINING
        membership ← initialMembership
        membership ← membership ∪ {(thisHost)}
        **Trigger Send**(ADD_REPLICA, (thisHost, 0, initialMembership[0]), initialMembership[0])
        **Setup Timer (AddReplicaTimer, 3)**

**Upon Order(opId, opKey, opData) do**:
    **if** (state = JOINING) **then**
        pendingOrders ← pendingOrders ∪ {(opId, ipData)}
        **return**
    **if** (leader = ⊥) **then**
        pendingOrders ← pendingOrders ∪ {(opId, ipData)}
    **else if** (thisHost = leader) **then**
        **Trigger Propose**(nextInstance, opId, opData)
        nextInstance ← nextInstance + 1
    **else**
        pendingOrders[opId] ← {(opData)}
        **Trigger Send**(LEADER_ORDER_MSG, (nextInstance, opId, opData), leader)

**Upon Decided(instance, opId, op) do**:
    **if** (thisHost ≠ leader) **then**
        nextInstance ← nextInstance + 1
    pendingOrders ← pendingOrders \ {(opId) }
    executedOperations ← executedOperations ∪ {(opId, op)}
    **Trigger Execute**(opId, op)

```

2.3.3 Order Request. In MultiPaxos, only the leader is responsible for proposing requests. Therefore, when a replica receives a request from its App Layer, it will forward the request to the leader if it is not the leader itself. If the replica is in the process of joining, the request will be buffered and held until the replica has fully joined the system, at which point it will be forwarded to the leader. Similarly, if no leader has been elected, requests are buffered and stored until a leader is chosen. Once a leader is in place, these pending requests are forwarded to the leader for processing. This mechanism ensures that requests are not lost and are eventually handled once a leader is available.

2.3.4 Decision. Once the Agreement Layer reaches consensus on an operation, it triggers the Decided Notification, which increments the state machine operation instance—except for the leader, whose instance is incremented when they propose the operation. The notification removes the operation from pendingOperations (if present) and places the decided operation in executedOperations. Finally, it propagates the operation to the Hash App Layer, which is responsible for maintaining the overall application state.

2.3.5 Add/Remove Replica. As mentioned in the initialization section, when a replica wants to join the system, it will contact an existing replica within the system. If the contacted replica is not the leader, it will forward the operation to the leader so that the leader can propose the AddReplica request. Once consensus is reached, each replica will add the new replica to their membership. The contacted replica, which will now the new replica in its replicaIdMapper upon contact, will request its own HashApp for the current state. This request will include the instance in which the new replica joined the system.

2.3.6 Add/Remove Replica PseudoCode.

```
**Upon AddReplica(newReplica, contact, instance) do**:
    **if** (leader = ⊥ ∨ (replica ∈ pendingAddRemoves)) **then**
        **return**
    **if** (thisHost = contact) **then**
        replicaIdSet ← replicaIdSet ∪ {newReplica}
    **foreach** $((k, v) \in$ executedAddRemoves$)$ **do**
        **if** $(v$.getLeft$()$ = newReplica $\wedge$ $v$.getRight$()$ = $\top)$ **then**
            **TriggerCurrentState**(instance)
            **return**
    **if** (thisHost ≠ leader) **then**
        **Trigger Send**(ADD_REPLICA_MSG, (newReplica, nextInstance, contact), leader)
        **return**
    **Trigger Connect**(newReplica)
    nextInstance ← nextInstance + 1
    **Trigger AddReplica**(nextInstance, newReplica)

**Upon MembershipChange(isAdding, replica, instance) do**:
    **if** (isAdding = ⊤) **then**
            **Trigger Connect**(replica)
            membership ← membership ∪ {replica}
        **if** (replica ∈ replicaIdSet) **then**
            replicaIdSet ← replicaIdSet \ {(replica)}
            **Trigger CurrentState**(instance)
    **else**
        **Trigger CloseConnection**(replica)
        membership ← membership \ {(replica)}
    **if** (thisHost ≠ leader) **then**
        nextInstance ← nextInstance + 1

**Upon CurrentState(instance, state) do**:
    **if** (leader = ⊥) **then**
        **return**
    newReplica ← executedAddRemoves[instance].host
    **Trigger Send**(REPLICA_ADDED_MSG, (instance, state, membership, leader), newReplica)

**Upon ReplicaAdded(newState, instance, newMembership, l) do**:
    leader ← l
    nextInstance ← instance
    membership ← newMembership
    **foreach** m ∈ membership **do**
```

```
        Trigger Connect(m)
    Trigger InstallState(newState)
    Trigger Joined(membership, nextInstance, leader)
    state ← ACTIVE
    foreach (orderId, value) ∈ pendingOrders do
        Trigger Send(LEADER_ORDER_MSG, (0, orderId, value), leader)

Upon Timer AddReplicaTimer() do:
    if (state = State.ACTIVE) then
        return
    if (membership.size() = 0) then
        Terminate Program
    contact ← membership.get(0)
    if (state = State.JOINING) then
        membership.remove(node)
    else if (leader = null) then
        pendingAddRemoves.put(node, false)
    else if (self.equals(leader)) then
        Trigger RemoveReplica(nextInstance ← nextInstance + 1, node)

```

Upon receiving the state in reply, the contacted replica will send the state to the new replica, along with the identity of the leader, the updated membership of the system, and the instance in which the new replica joined (as returned by the current state request). This joinedInstance is essential for the new replica to "catch up" to the current state of the leader. The new replica can do so either by making a Log Request to the leader with the snapshot of the system (Classic Paxos) or by receiving the undecided messages in the Paxos layer before joining(Distinguished Learner Paxos). Once the new replica has installed the state it received, it will transition from JOINING to ACTIVE and become ready to process requests from its HashApp layer. Additionally, it will flush any orders accumulated during the JOINING state at the state machine protocol level, trigger the JoinedNotification in the agreement layer, and fully catch up to the log state.

In addition to this, several fault tolerance mechanisms have been implemented to make the addition process more robust. Firstly, there is the AddReplicaTimer. If the new replica does not contact the designated replica or if the message fails to reach the leader (either due to failure of the contact replica or because the leader is down), the timer will simply retry the operation. Additionally, whenever a replica within the membership of the new replica fails, it will be removed from the membership. This mechanism rotates the contacts for the new replica and ensures termination. Specifically, if there are no remaining members in the membership of the new replica, the process will terminate, and all other replicas will initiate a RemoveReplica request if the AddReplica operation had received a majority but was not completed.

If the AddReplica operation has already been initiated, but is still pending (not yet decided), the timer will simply retry the operation. However, if the operation is already present in the executed messages, it means that the decision was made previously, but the new replica was not informed by the previous contact when the decision occurred. In this case, the contacted replica will immediately initiate the state transfer to the new replica. This ensures that the operation is not repeated unnecessarily. To further prevent duplication, the new replica will reject any state transfer if it does not come from its most recent contact, ensuring that the state is only accepted once and from the correct source.

A replica is removed when the membership detects it has failed, either by being unable to establish a connection or by an ongoing connection failing. In the case of a failed connection attempt, after a certain number of retries, the leader will propose a request to remove the failed replica. If no leader exists, the request is buffered in a list and will be processed once a new leader is elected. The process is similar when an established connection goes down, with the key difference that a new leader may be re-elected, as other replicas in the membership will detect the failure of their connection to any replica, including leader.

2.3.7 *Connection Failures and Leader Election PseudoCode.*

```
Upon LeaderMsgFail(opId, op, instance, host) do:
    if (leader = ⊥) then
        pendingOrders[opId] ← op
    else
        Send(LEADER_ORDER_MSG, (opId, op, instance), host)

Upon OutConnectionDown(host) do:
    if (leader = ⊥ or host = leader) then
        pendingAddRemoves[host] ← ⊥
    if (host = leader) then
        previousLeader ← leader
        leader ← ⊥
        Call NextLeaderCandidate
    Trigger CloseConnection(host)
    if (thisHost = leader) then
        nextInstance ← nextInstance + 1
        Trigger RemoveReplica(nextInstance, node)

Upon OutConnectionFailed(host) do:
    tid = Setup Timer ConnectionRetryTimer(0.05)
    hostTimers ← hostTimers ∪ {(tid, host)}
    if (host ∉ hostRetries) then
        hostRetries ← hostRetries ∪ {(host, MAXRETRIES)}

Upon Timer ConnectionRetryTimer(t, tid) do:
    node ← hostTimers.remove(tid)
    r ← hostRetries.computeIfPresent(node, (key, value) ⟶ value - 1)
    if (r ≠ null and r > 0 and membership.contains(node)) then
        Trigger Connect(node)
        return
    hostRetries.remove(node)
    if (state = State.JOINING) then
        membership.remove(node)
    else if (leader = null) then
        pendingAddRemoves.put(node, false)
    else if (self.equals(leader)) then
        Trigger RemoveReplica(nextInstance ← nextInstance + 1, node)

Procedure NextLeaderCandidate() do:
    foreach Descending (h ∈ membership) then
        if (pendingAddRemoves[h] = ⊥ or h ≠ previousLeader) then
            if (h = thisHost) then
                previousLeader = h
                Trigger Prepare(nextInstance)
            else
                break
    Setup Timer LeaderCandidateTimer(1000)
```

```
48:
49: Upon Timer LeaderCandidateTimer(t, tid) do:
50:     if (leader = ⊥) then
51:         Call NextLeaderCandidate
52:     else
53:         previousLeader ⟵ null
```

2.3.8 *Leader Election.*

At the state machine protocol layer, once the failure of the leader is detected, a recursive process is initiated to select a new leader. This process begins by selecting a replica (starting from the highest index) that is neither the previous leader, nor one marked for removal, nor in the joining state. The candidate replica will then issue a prepare request to the agreement layer. All replicas must agree on the new leader, or the timer will trigger a retry.

Once the leader initiates the prepareRequest, and the agreement layer reaches consensus, the new leader will receive all accepted messages that occurred past its own accepted instance. This allows the new leader to reissue these messages in the same order they were commited to all replicas. Additionally, any messages that were waiting for a leader election to be completed will be issued by the new leader in an arbitrary order, including Add and Remove Replica operations

2.3.9 *New Leader and Leader Orders PseudoCode.*

```
1: Upon NewLeader(newLeader, prepareOkMsgs) do:
2:     leader ⟵ newLeader
3:     if (thisHost = leader) then
4:         foreach (msg ∈ prepareOkMsgs) do
5:             pendingOrders ⟵ pendingOrders \ {(msg.id)}
6:             nextInstance ⟵ nextInstance + 1
7:             Trigger Propose(nextInstance, msg.id, msg.value)
8:         foreach ((host, isPending) ∈ pendingAddRemoves) do
9:             nextInstance ⟵ nextInstance + 1
10:            if (isPending = ⊤) then
11:                Trigger AddReplica(nextInstance, host)
12:            else
13:                Trigger RemoveReplica(nextInstance, host)
14:        foreach ((oId, value) ∈ pendingOrders) do
15:            nextInstance ⟵ nextInstance + 1
16:            Trigger Propose(nextInstance, oId, value)
17:    else
18:        foreach (order ∈ pendingOrders) do
19:            Trigger Send(LEADER_ORDER_MSG, (0, order.key, order.value), leader)
20:    pendingOrders ⟵ {}
21:    pendingAddRemoves ⟵ {}
22:
23: Upon LeaderOrder(instance, opId, op) do:
24:    if (leader = ⊥) then
25:        pendingOrders[opId] ⟵ op
26:        return
27:    nextInstance ⟵ nextInstance + 1
28:    Trigger Propose(nextInstance, opId, op)
29:
```

2.4 Multi-Paxos (Classic)

```
1: Interface:
2:     Requests:
3:         Propose(instance, id, op)
4:         AddReplica(instance, host)
5:         RemoveReplica(instance, host)
6:     Indications:
7:         Decided(instance, id, op)
8:
9: State:
10:    thisHost                ▷ This node's host information
11:    joinedInstance          ▷ This node's index in the membership
12:    prepareOkCount          ▷ Number of PrepareOk messages received for the current operation
13:    highestPrepare          ▷ Highest node index of received PrepareOk messages
14:    membership              ▷ Set of node's currently in membership
15:    toBeDecidedIndex        ▷ To be decided index
16:    instanceStateMap        ▷ Map of each instance's: index -> (acceptOkCount, isDecided)
17:    toBeDecidedMessages     ▷ Map of index -> (opId, opValue)
18:    acceptedMessages        ▷ Map of index -> (opId, opValue)
19:    addReplicaInstances     ▷ Map of index -> (host, isAdding)
20:
21: Upon Init(h, initialMembership) do:
22:    thisHost ⟵ h
23:    joinedInstance ⟵ -1
24:    membership ⟵ ⊥
25:    prepareOkCount ⟵ 0
26:    highestPrepare ⟵ 0
27:    toBeDecidedIndex ⟵ 1
28:    toBeDecidedMessages ⟵ {}
29:    acceptedMessages ⟵ {}
30:    addReplicaInstances ⟵ {}
31:    prepareOkMessages ⟵ {}
32:    instanceStateMap ⟵ {}
33:
34: Upon Prepare(instance) do:
35:    prepareOkCount ⟵ 0
36:    highestPrepare ⟵ highestPrepare + 1
37:    foreach (h ∈ membership) do
38:        Trigger Send(PREPARE_MSG, (highestPrepare, instance, ⊥), h)
39:
40: Upon PrepareMsg((seqNumber, instance, isOk), h) do:
41:    if (seqNumber < highestPrepare) then
42:        return
43:    highestPrepare ⟵ seqNumber
44:    relevantMessages ⟵ {}
45:    if (thisHost ≠ h ∧ joinedInstance ≥ 0) then
46:        foreach (m ∈ acceptedMessages) do
47:            if (m.instance ≥ instance) then
48:                relevantMessages ⟵ relevantMessages ∪ {m}
49:        Trigger NewLeader(h)
50:        toBeDecidedMessages ⟵ {}
51:    Trigger Send(PREPARE_OK_MSG, (highestPrepare, relevantMessages), h)
52:
53: Upon PrepareOkMsg((seqNumber, relevantMessages), h) do:
54:    if (seqNumber < highestPrepare) then
55:        return
56:    prepareOkCount ⟵ prepareOkCount + 1
57:    if (#relevantMessages > #prepareOkMessages) then
58:        prepareOkMessages ⟵ relevantMessages
59:    if (prepareOkCount ≥ (#membership / 2) + 1) then
60:        prepareOkCount ⟵ -1
61:        Trigger NewLeader(thisHost, prepareOkMessages)
62:        prepareOkMessages ⟵ {}
```

```

Upon Joined(jMembership, instance, leader) do:
    membership ⟵ jMembership
    toBeDecidedIndex ⟵ joinedInstance
    if (notification.getContact() ≠ null and toBeDecidedIndex > 1) then
        Trigger Send(LOGREAD_MSG, (toBeDecidedIndex), leader)
    else
        joinedInstance ⟵ toBeDecidedIndex

Upon LogReadMsg((instance), h) do:
    accMsgs ⟵ acceptedMessages.tailMap[msg.getInstance(), ∞]
    Trigger Send(LOGWRITE_MSG, (t, hp, a, tbd, ri, leader))

Upon LogWriteMsg((m), h) do:
    toBeDecidedIndex ⟵ m.getInstance()
    joinedInstance ⟵ toBeDecidedIndex
    highest_prepare ⟵ m.getHighestPrepare()
    acceptedMessages ⟵ m.getAcceptedMessages()
    toBeDecidedMessages.putAll(m.getToBeDecidedMessages())
    addReplicaInstances ⟵ m.getAddReplicaInstances()
    for each (k, v) in acceptedMessages do
        if (addReplicaInstances.containsKey(k)) then
            r ⟵ addReplicaInstances.get(k)
            Trigger Send(CHANGE_MEMBERSHIP_MSG, (k, r.getLeft(), r.getRight()))
        else
            Trigger Decided(k, v.getLeft, v.getRight)

Upon AddReplica(instance, host) do:
    foreach (h ∈ membership) do
        Trigger Send(CHANGE_MEMBERSHIP_MSG, (host, instance, highestPrepare, ⊥, ⊤), h)

Upon RemoveReplica(instance, host) do:
    foreach (h ∈ membership) do
        if (h ≠ host) then
            Trigger Send(CHANGE_MEMBERSHIP_MSG, (host, instance, highestPrepare, ⊥, ⊥), h)

Upon Propose(instance, opId, op) do:
    foreach (h ∈ membership) do
        Trigger Send(ACCEPT_MSG, (instance, highestPrepare, opId, op), h)

Upon ChangeMembershipMsg((host, instance, seqNumber, isAdding, isOk, mToBeDecidedMsgs, mAddReplicaInstances), h) do:
    if (seqNumber < highestPrepare) then
        return
    highestPrepare ⟵ seqNumber
    if (isAdding = ⊥) then
        membership ⟵ membership,remove(host)
    instanceStateMap[instance] ⟵ (0, ⊥)
    acceptOkMsg ⟵ (instance, seqNumber, GenerateUUID(), )
    toBeDecidedMessages[instance] ⟵ (acceptOkMsg.opId, acceptOkMsg.op)
    addReplicaInstances[instance] ⟵ (host, isAdding)
    foreach (h ∈ membership) do
        Trigger Send(ACCEPT_OK_MSG, acceptOkMsg, h)

Upon AcceptMsg((opId, instance, op, seqNumber, lastChosen), h) do:
    if (seqNumber < highestPrepare) then
        return
    highestPrepare ⟵ seqNumber
    toBeDecidedMessages[instance] ⟵ (opId, op)
    if (joinedInstance ≥ 0) then
        instanceStateMap[instance] ⟵ (0, ⊥)
        foreach (h ∈ membership) do
            Trigger Send(ACCEPT_OK_MSG, (opId, instance, op, seqNumber), h)

Upon AcceptOkMsg((opId, instance, op, seqNumber), h) do:
    if (instanceStateMap[instance] = ⊥) then
        return
    if (seqNumber = > highestPrepare) then
        state.acceptOkCount ⟵ 0
    state.acceptOkCount ⟵ state.acceptOkCount + 1
    if (state.acceptOkCount ≥ (#membership / 2) + 1 and state.isDecided = ⊥) then
        state.isDecided ⟵ ⊤
        while (toBeDecidedIndex ≤ instance) do
            pair ⟵ toBeDecidedMessages[toBeDecidedIndex]
            toBeDecidedMessages ⟵ toBeDecidedMessages.remove(toBeDecidedIndex)
            if (pair ≠ ⊥) then
                break
            acceptedMessages[toBeDecidedIndex] ⟵ pair
            if (addReplicaInstance[toBeDecidedIndex] ≠ ⊥) then
                Trigger Decided(toBeDecidedIndex, pair.opId, pair.op)
            else
                rPair ⟵ addReplicaInstances[toBeDecidedIndex]
            if (rPair.isAdding = ⊤) then
                    membership ⟵ membership ∪ {rPair.host}
            Trigger MembershipChanged(rPair.host, rPair.isAdding, toBeDecidedIndex)
            toBeDecidedIndex ⟵ toBeDecidedIndex + 1

```

In Paxos Classic, unlike the Distinguished Learner variant, replicas do not wait for the leader to commit before triggering the new leader notification and updating their state machine layers. Instead, once a new leader is elected, replicas immediately add the appointed leader to their state machine and flush any pending messages while proceeding with the next steps. Like the Distinguished Learner variant, all replicas send their accepted or committed messages past instance n, where n is the instance of the last accepted message from the new leader. This allows the new leader to re-execute these messages (in the order they were committed) to all the replicas, ensuring that they all synchronize the state upon re-election. The key challenges, leader re-election, adding/removing replicas, and normal synchronized state maintenance—have, been addressed and fully tested in both Paxos variants. However, the way these challenges were resolved differs between the two variants. In Paxos Classic, for example, after a new replica joins the system, it requests the current log state snapshot from the leader. This snapshot allows the new replica to execute the leader's messages and synchronize the toBeDecidedMessages list, highest-prepare and toBecidedIndex. This approach ensures that even if a new leader is re-elected in the meantime, the snapshot remains accurate, as the state machine layer guarantees that the new replica will not proceed without a valid leader.

All requests (add, remove, and propose) are processed in the same way. The leader proposes a value, and all replicas accept the request from the leader and send AcceptOK messages to all other replicas.

Once each replica receives a majority of AcceptOK responses, they can decide on the proposed value. To avoid concurrency issues, we use a toBeDecidedList. When the leader and replicas receive an Accept message, they place the operation, along with the instance number, in this list. Upon receiving AcceptOK messages from a majority of replicas, the operation is removed from the list and added to the acceptedMessages. This ensures that the system can safely loop from the current toBeDecidedIndex to the instance number obtained from the message, executing all messages in the correct order.

Replicas that have not yet joined the system will not have entries in their InstanceStateMap, and they will not send AcceptOK messages to other replicas, nor will they be able to process them. The replica will update its JoinedInstance and become a normal replica in the consensus once it has caught up with its logState, or if no operations have been executed, in which case the JoinedInstance will be set to 1. Additionally, in the Accept phase, since Add and Remove replica operations have no data and are not initiated by the YCSB client, there is a list that details which instances correspond to normal operations or AddRemove operations. In the AcceptOK phase, once we receive a majority, if the operation is present on this list, we execute a MembershipChangedNotification. This notification will either add a replica to the membership, and the contacted replica will transfer its state to the new replica (as detailed in the SMR section), or if it is a RemoveReplica operation, the replica will be removed from the membership. Since the membership list is used to deliver messages between replicas in both layers, the replica must be either removed or added in both layers.

If the operation is a normal read/write operation, we will execute the DecidedNotification and perform the operation at the HashApp layer level.

2.5 Multi-Paxos (Distinguished Learner)

```
Interface:
  Requests:
    Propose(instance, id, op)
    AddReplica(instance, host)
    RemoveReplica(instance, host)
  Indications:
    Decided(instance, id, op)


State:
  thisHost                  ▷ This node's host information
  joinedInstance            ▷ This node's index in the membership
  prepareOkCount ▷ Number of PrepareOk messages received for the
current operation
  highestPrepare            ▷ Highest node index of received PrepareOk
messages
  membership                ▷ Set of node's currently in membership
  toBeDecidedIndex          ▷ To be decided index
  instanceStateMap          ▷ Map of each instance's: index ->
(acceptOkCount, isDecided)
  toBeDecidedMessages       ▷ Map of index -> (opId, opValue)
  acceptedMessages          ▷ Map of index -> (opId, opValue)

Upon Init(h, initialMembership) do:
  thisHost ⟵ h
  joinedInstance ⟵ -1
  membership ⟵ ⊥
  prepareOkCount ⟵ 0
  highestPrepare ⟵ 0
  toBeDecidedIndex ⟵ 1
  toBeDecidedMessages ⟵ {}
  acceptedMessages ⟵ {}
  prepareOkMessages ⟵ {}
  instanceStateMap ⟵ {}

Upon Prepare(instance) do:
  prepareOkCount ⟵ 0
  highestPrepare ⟵ highestPrepare + 1
  foreach (h ∈ membership) do
    Trigger Send(PREPARE_MSG, (highestPrepare, instance, ⊥), h)

Upon PrepareMsg((seqNumber, instance, isOk), h) do:
  if (seqNumber < highestPrepare or joinedInstance < 0) then
    return
  highestPrepare ⟵ seqNumber
  if (isOk) then
    Trigger NewLeader(instance)
  relevantMessages ⟵ {}
  if (thisHost ≠ h) then
    foreach (m ∈ acceptedMessages) do
      if (m.instance ≥ instance) then
        relevantMessages ⟵ relevantMessages ∪ {(m)}
    foreach (m ∈ toBeDecidedMessages) do
      if (m.instance ≥ instance + #relevantMessages) then
        relevantMessages ⟵ relevantMessages ∪ {(m)}
    toBeDecidedMessages ⟵ {}
  Trigger Send(PREPARE_OK_MSG, (highestPrepare, relevantMes-
sages), h)

Upon PrepareOkMsg((seqNumber, relevantMessages), h) do:
  if (seqNumber < highestPrepare) then
    return
  prepareOkCount ⟵ prepareOkCount + 1
  if (#relevantMessages > #prepareOkMessages) then
    prepareOkMessages ⟵ relevantMessages
  if (prepareOkCount ≥ (#membership / 2) + 1) then
    prepareOkCount ⟵ -1
    Trigger NewLeader(thisHost, prepareOkMessages)
    foreach (h ∈ membership) do
      if (h ≠ thisHost) then
        Trigger Send(PREPARE_MSG, (seqNumber, -1, ⊤), h)
    prepareOkMessages ⟵ {}

Upon Joined(jMembership, instance, isInitial) do:
  joinedInstance ⟵ instance
  membership ⟵ jMembership
  toBeDecidedIndex ⟵ joinedInstance

Upon AddReplica(instance, host) do:
  instanceStateMap[instance] ⟵ (0, ⊥)
  foreach (h ∈ membership) do
    Trigger Send(CHANGE_MEMBERSHIP_MSG, (host, instance,
highestPrepare, ⊥, ⊤), h)

Upon RemoveReplica(instance, host) do:
  membership ⟵ membership  {(host)}
  instanceStateMap[instance] ⟵ (0, ⊥)
  foreach (h ∈ membership) do
```

```
        Trigger Send(CHANGE_MEMBERSHIP_MSG, (host, instance,
    highestPrepare, ⊥, ⊥), h)

Upon Propose(instance, opId, op) do:
    instanceStateMap[instance] ⟵ (0, ⊥)
    foreach (h ∈ membership) do
        Trigger Send(ACCEPT_MSG, (instance, highestPrepare, opId,
    op), h)

Upon ChangeMembershipMsg((host, instance, seqNumber, isAdding,
isOk, mToBeDecidedMsgs, mAddReplicaInstances), h) do:
    if (seqNumber < highestPrepare) then
        return
    highestPrepare ⟵ seqNumber
    if (isOk = ⊤) then
        if (#addReplicaInstances < #mAddReplicaInstances) then
            addReplicaInstances ⟵ mAddReplicaInstances
        foreach (m ∈ mToBeDecidedMessages) do
            toBeDecidedMessages ⟵ toBeDecidedMessages ∪ {(m)}
        return
    if (isAdding = ⊥) then
        membership ⟵ membership  host
    instanceStateMap[instance] ⟵ (0, ⊥)
    acceptOkMsg ⟵ (instance, seqNumber, GenerateUUID(), )
    toBeDecidedMessages[instance] ⟵ (acceptOkMsg.opId, acceptOkMsg.op)
    addReplicaInstances[instance] ⟵ (host, isAdding)
    foreach (h ∈ membership) do
        Trigger Send(ACCEPT_OK_MSG, acceptOkMsg, h)

Upon AcceptMsg((opId, instance, op, seqNumber, lastChosen),
h) do:
    if (seqNumber < highestPrepare) then
        return
    highestPrepare ⟵ seqNumber
    toBeDecidedMessages[instance] ⟵ (opId, op)
    if (joinedInstance ≥ 0) then
        instanceStateMap[instance] ⟵ (0, ⊥)
        foreach (h ∈ membership) do
            Trigger Send(ACCEPT_OK_MSG, (opId, instance, op, seqNum-
    ber), h)

Upon AcceptOkMsg((opId, instance, op, seqNumber), h) do:
    if (instanceStateMap[instance] = ⊥) then
        return
    if (seqNumber = > highestPrepare) then
        state.acceptOkCount ⟵ 0
    state.acceptOkCount ⟵ state.acceptOkCount + 1
    if (state.acceptOkCount ≥ (#membership / 2) + 1 and state.isDecided
    = ⊥) then
        state.isDecided ⟵ ⊤
        while (toBeDecidedIndex ≤ instance) do
            pair ⟵ toBeDecidedMessages[toBeDecidedIndex]
            toBeDecidedMessages ⟵ toBeDecidedMessages {toBeDecidedIn-
    dex}
            if (pair ≠ ⊥) then
                acceptedMessages[toBeDecidedIndex] ⟵ pair
                if (addReplicaInstance[toBeDecidedIndex] ≠ ⊥) then
                    Trigger Decided(toBeDecidedIndex, pair.opId, pair.op)
                else
                    rPair ⟵ addReplicaInstances[toBeDecidedIndex]
                    if (rPair.isAdding = ⊤) then
                        relevantMsgs ⟵
                        foreach (m ∈ toBeDecidedMessages) do
                            if (m.index ≥ toBeDecidedIndex + 1) then
                                relevantMsgs[m.index] ⟵ (m.opId, m.op)
                        Trigger Send(CHANGE_MEMBERSHIP_MSG, (rPair.host,
    toBeDecidedIndex, seqNumber, ⊤, ⊤, relevantMessages, addReplicaIn-
    stances), rPair.host)
                        membership ⟵ membership ∪ {rPair.host}
                    Trigger MembershipChanged(rPair.host, rPair.isAdding,
    toBeDecidedIndex)
            toBeDecidedIndex ⟵ toBeDecidedIndex + 1

```

The Distinguishing Learner variant of Paxos operates differently from the Classic Paxos, for instance, in the way it handles the prepare phase. In the Classic Paxos, a new replica must first synchronize with the leader to catch up on the log before it can participate in the consensus process. In contrast, the Distinguishing Learner variant does not require an explicit log catch-up process. Instead, synchronization happens during the prepare phase, where the replica that is aiming to become the leader receives all messages, including toBeDecidedMessages and acceptedMessages, starting from the instance it is trying to become the leader (denoted as instance n) and continuing until the end. These messages are then re-executed by the new leader in the exact order they were received. As a result, when a new replica joins the Agreement Layer, after having the state transferred and installed, it can immediately update its joinedInstance and begin executing its toBeDecidedMessages, without needing to synchronize with the current leader. This is possible because the new replica will have the messages it needs to decide (that it received while joining) synchronized with the current leader.

In the Distinguishing Learner variant, the leader proposes an operation and sends an accept message to all replicas, each of which responds with an accept OK message. Once the proposer collects a majority of accept OKs, it commits the operation. The key part here is how the leader's accept messages work and how the replicas process them, especially with respect to the toBeDecidedMessages list and the lastChosen value.

When the leader sends the first accept message, it sets the lastChosen value to be instance - 1. So, if the instance is 1, the leader will set lastChosen to 0 in the message. When a replica receives this message, it checks the instance number against its toBeDecidedIndex, which is the index of the instance that has yet to be decided. If the toBeDecidedIndex is greater than the lastChosen value (which, in this case, is 0), the replica skips the for loop designed to execute messages. This is because the toBeDecidedIndex is ahead of the lastChosen, and thus the replica has already processed messages beyond the point the leader wants to decide, and this will be the case the first time the leader sends the message, since it has not been commited yet.

Since the loop is skipped, the replica places the received message in its toBeDecidedMessages list, but it does not execute it yet. There is a check to see if there was a double insertion, it is there to stop replicas that are joining(joinedInstance < 0) to not send an extra message to the leader.

Once the proposer has received a majority of Accept OK messages, it commits the message and sends another Accept message, but this time with the lastChosen value updated to match the instance number of the message that has been decided. For example, if the current instance is 1, the lastChosen value will be set to 1.

The key logic here is that the toBeDecidedIndex always points to the instance that is about to be decided when the leader proposes it. During the first round of Accept messages, the toBeDecidedIndex will not be updated, as the lastChosen value will be less than the instance number of the message, and so, all replicas will buffer this message in toBeDecided. In the second round, when the proposer sends the commit, the lastChosen value will match the toBeDecidedIndex. At this point, the committed message, present in toBeDecidedMessages can be executed by all replicas and added to their acceptedMessages.

The Add and Remove operations function similarly to the already described Prepare and Accept requests. In both cases, the operation will be issued by the Distinguished Learner, all replicas will send AcceptOKs to him, and once a majority/consensus is reached, the learner will commit the message to be executed by all replicas. In the case of Add and Remove, it will either be a membership addition or removal.

3 TESTING

We conducted independent tests on networks including 3 and 5 - using, for each, two payload size scenarios: 1 KB, and 10 KB.

For each replica, a single client is assigned to it, sending 100 000 messages, with a 50/50 distribution reads and writes. Accordingly, for 3 replicas, 300 000 messages are sent, for 5 replicas, 500 000 messages, etc.

Each test case was repeated 2 times, in order to establish a reliable base, as a single run could be misleading and might not accurately reflect our system's performance.

3.1 Testing Outcomes

The following graphs compile the main takeaways from the experiments, comparing each algorithm stack directly via the corresponding color-codes.

If interested - all raw test outputs are available via the following Google Sheets Doc URL: https://docs.google.com/spreadsheets/d/1nIZhW3woIoNpFzNoptmrHNKo2inFsOQI2j3e-vgm2J4

ABD Quorum (1 KB payload)
ABD Quorum (10 KB payload)
Paxos Classic (1 KB payload)
Paxos Classic (10 KB payload)
Paxos Distinguished (1 KB payload)
Paxos Distinguished (10 KB payload)

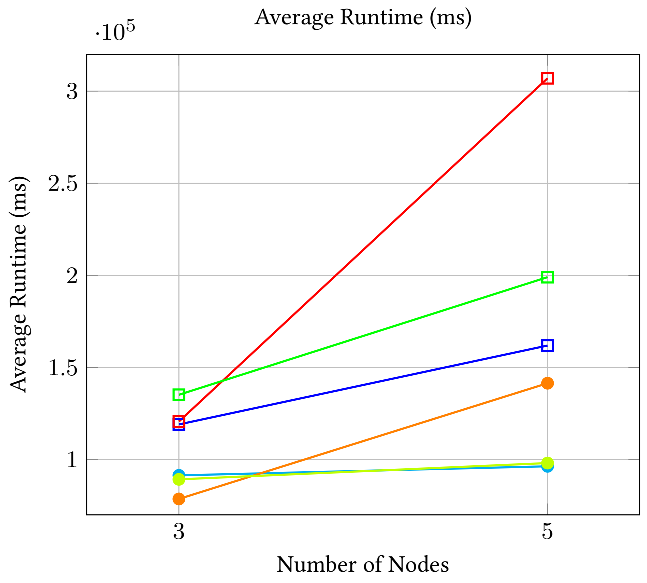


Figure 1: Average Runtime (ms)

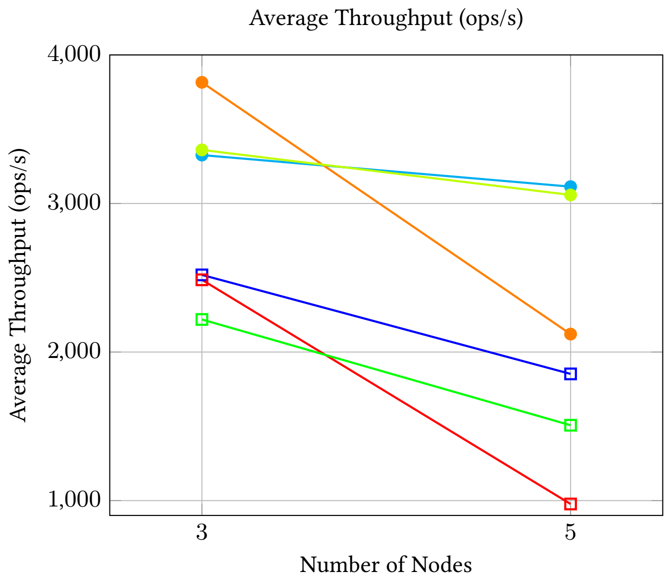


Figure 2: Average Throughput (ops/s)

4 CONCLUSIONS

We are pleased with our 3 implementations of the DLT/Replication stacks, as they address most of the challenges presented to us, with a blend of readable, extensible, and efficient code. In every experiment, the protocols performed robustly to any operations in the system, as expected, although performance suffered in some local tests, due to relatively low-powered hardware.

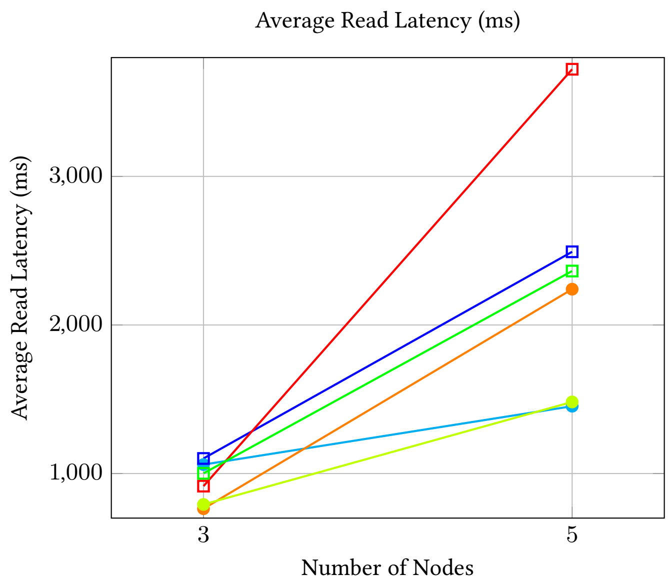


Figure 3: Average Read Latency (ms)

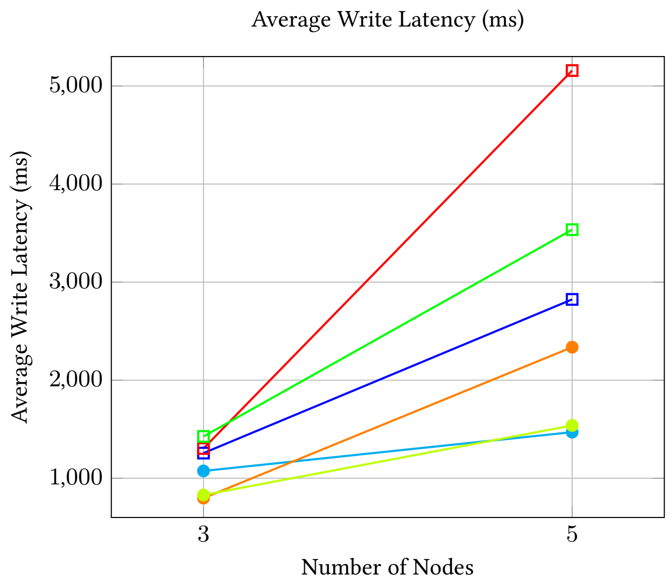


Figure 4: Average Write Latency (ms)

Not so surprisingly, the ABD Quorum algorithm performed rather well in these smaller scale, local tests, although, we suspect that with much higher node counts, the performance would rapidly decrease across all graphs.

As for the Paxos implementations, the Classic version functioned as expected during 3 node tests, but, quickly gave rise to performance problems when run with 5 nodes locally on a lower-powered laptop, as can be observed by every graph, Paxos Classic with a 10 KB Payload is unequivocally the worst outlier in every comparison.

Meanwhile, the Distinguished Learner version behaved quite well under the 5 node experiments, due to the much lower volume of messages across every transaction, so we suspect it can easily perform at a much higher node count.

ACKNOWLEDGMENTS

We would like to thank both our Distributed Systems Algorithms course professors, João Leitão and Alexander Davidson, for their support throughout these projects, both during and outside of classes. Their teaching styles were excellent throughout the semester, as professors who always pointed us in the right direction, without giving away too much, as to leave us room to develop our critical thinking.

Of course, we also relied on provided lecture materials, along with some foundational research papers, which served as blueprints for our work, mainly regarding the Paxos Distinguished Learner protocol.

REFERENCES

[1] Pedro Fouto, Nuno Preguiça, and Joao Leitão. 2022. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 575–592. https://www.usenix.org/conference/atc22/presentation/fouto