Distributed Systems Algorithms - Distributed Hash Table

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

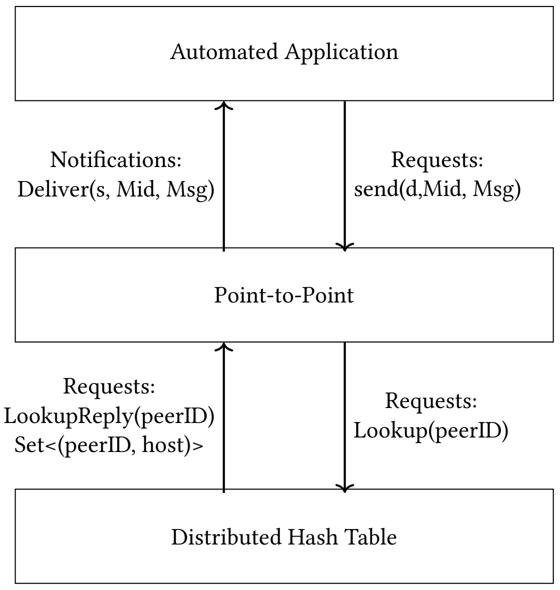
Peer-to-peer applications face challenges such as load balancing, efficient peer lookup, and ensuring that peers can reliably find all relevant resources in the system. To address these issues, we developed a scalable, robust, and efficient peer-to-peer point-to-point communication system. Our system utilizes a Distributed Hash Table (DHT) based on the Chord protocol, which facilitates efficient peer lookups by translating unique peer identifiers into the corresponding IP addresses and ports, enabling seamless direct communication between nodes. The Chord protocol is known for its scalability, with both communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes, as supported by extensive experiments and analysis. To tackle reliability, we introduced the concept of a "helper node", that steps in on behalf of the sender if the sender becomes unavailable to help the sender in case he fails to send the message, This contribution boosts message delivery success and system robustness, and will be explored in greater detail later in the report.

ACM Reference Format:
Gonçalo Virgínia and Ricardo Rodrigues. 2024. Distributed Systems Algorithms - Distributed Hash Table. In *The Projects of ASD - first delivery, 2024, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Peer-to-peer systems and applications operate as decentralized overlay networks. In such systems, all nodes are equal, running identical software that allows them to function both as clients and servers, enabling seamless data sharing across the network. However, despite these strengths, and more, Peer-to-peer systems face inherent challenges. Issues such as load balancing, peer lookup efficiency, and ensuring resource availability across the system have traditionally posed difficulties.

This project was developed to address these challenges by creating a scalable and robust peer-to-peer point-to-point communication system. The system leverages a Distributed Hash Table (DHT), implemented following the Chord protocol. It consists of three main layers: the Application Layer, responsible for sending and receiving payloads; the Peer-to-Peer Layer, which facilitates direct communication between peers; and the DHT Layer, which handles the mapping of messages to nodes using Chord's efficient lookup mechanism. In essence, when a node in our system wants to send a message, it first contacts the DHT Layer with a lookup request through the Peer-to-Peer layer. The DHT layer identifies the target peer by mapping the message's ID to the appropriate node in the Chord ring and returns the target node information to the Peer-to-Peer layer. The Peer-to-Peer layer then uses this information to send the message to the target node, where it is received by that node's own Peer-to-Peer layer and propagated to its Application Layer for final delivery.


Automated Application
Notifications: Deliver(s, Mid, Msg)
Requests: send(d,Mid, Msg)
Point-to-Point
Requests: LookupReply(peerID) Set<(peerID, host)>
Requests: Lookup(peerID)
Distributed Hash Table

To address robustness in our peer-to-peer system, we introduced the concept of a Helper Node. In our implementation, the message sender continuously attempts to resend the message to the designated peer until a connection is established. However, the scenario where the sender fails needed to be addressed. To do this, an additional node is returned during the message lookup, specifically designated to resend the message if the original sender becomes unavailable. To prevent unnecessary flooding of the network, the Helper Node only sends the original message when the sender disconnects. If the Helper Node also goes offline, its predecessor in the Chord Ring takes on the role of the new Helper Node.

To ensure our Distributed Hash Table (DHT) is both efficient and scalable, as well as capable of handling the dynamic nature of nodes joining and leaving the network, we implemented it using

[*]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 Chord protocol. Chord enables efficient peer lookups by employing a consistent hashing mechanism that maps message IDs to specific nodes within the network. Consistent hashing tends to balance the load, as each node is assigned roughly the same number of keys. This distribution changes very little when nodes join or leave the system, helping maintain a stable and even workload across the network. These nodes are organized in a logical ring structure, known as the "Chord ring." This design allows the system to quickly determine which node is responsible for a particular message, ensuring that peer lookup requests are processed efficiently, even as the network scales. In a Chord system, each node maintains routing information of $O(\log N)$ other nodes and resolves lookups using $O(\log N)$ messages. When nodes join or leave, the system efficiently updates its routing with at most $O(\log N^2)$ messages.

The remainder of this paper is organized as follows: Section 2 discusses related work, while Section 3 details the Application Protocol responsible for delivering and receiving payloads. Section 4 presents the pseudocode and explanation of the peer-to-peer layer, and Section 5 describes the Chord Protocol, which maps messages to nodes for delivery. Finally, Section 6 showcases the results of our experiments conducted at the DI and NOVA LINCS research cluster, and we conclude with our insights and a summary of what we discussed in the paper in section 7.

2 RELATED WORKS

This first project, part of the Distributed Systems Algorithms specialization course of the Computer Science and Engineering Master's Degree, serves to expand our experience with developing more complex and layered algorithms, as compared to the BSc. Distributed Systems projects, which only dealt with the implementation of relatively simpler algorithms, which utilized pre-existing centralized servers for event ordering, such as Apache Kafka, or only applying the Apache ZooKeeper component for primary-secondary server replication and management.
On the other hand, the current course focuses on a bottom-up approach of developing increasingly abstracted layers of algorithms, each with their separate function in the system - which also includes the production of language abstracted pseudo-code.
As for professional experience, both group members interned at a company whose main focus revolved around managing and implementing a distributed system of products on client's servers. Gonçalo, in particular, worked on the back-end, which included developing a new product for remotely viewing and managing multiple machine's hardware and software usage statistics and configurations, which, although implemented in a more standard centralized "master node" architecture, still served to refine aspects which are used in this project, such as: handling a non-responsive or downed peer, etc.

3 AUTOMATED APPLICATION PROTOCOL

The AutomatedApp (Application Layer), facilitates message exchange peers. Each peer is uniquely identified by a sequence number, configured through the processSequence property, ensuring messages are sent to other processes(not including the creator of the message). Each peer is assigned a unique ID, generated by hashing a combination of the peer's sequence number and random seed, ensuring unique identifiers across the network. The Application Layer is regulated by four distinct timers. These timers orchestrate the application's lifecycle, from startup, to periodic message broadcasting and eventually shutting down. The first timer, StartTimer, initiates after a specified prepareTime to allow time for other layers to setup. When triggered, it calls the uponStartTimer method, which starts periodic broadcasting of messages. This setup includes a secondary timer to stop the broadcast phase after the configured runtime. The second timer, SendMessageTimer, is responsible for periodic message sending at fixed intervals. Upon each trigger, a message is generated, this payload, is sent to peer-to-peer layer, which will be responsible for looking up the DHT layer to find the designated peer for that messaged(based on the message ID and the Chord Protocol, which will be discussed in section 5) . The third timer, StopTimer, stops the periodic message broadcast once the application's runtime has elapsed. It also sets up the ExitTimer to facilitate a graceful shutdown after a cooldown period. Finally, the ExitTimer calls the uponExitTimer method, allowing the application to exit cleanly after the specified cooldownTime. To aid in understanding the implementation of this protocol, we present the pseudocode for this layer in the next sub-section.

3.0.1 *Automated Application Protocol Pseudocode.*

```
State:
    payloadSize              ▷ Size of the payload of each message (in bytes)
    prepareTime              ▷ Time to wait until starting sending messages
    runTime                  ▷ Time to run before shutting down
    cooldownTime             ▷ Time to wait before shutting down
    broadcastInterval        ▷ Interval between each broadcast
    randomSeed               ▷ Random seed for topic generation
    nPeers                   ▷ Number of different peers to be considered
    idBits                   ▷ Number of bits used for peer IDs
    processSequenceNumber    ▷ Sequence Number of this process
    sendMessageTimer         ▷ Timer for sending messages
    myPeerID                 ▷ This host's peerID
    myPeerIDHex              ▷ This host's peerID in hexadecimal format
    peerIDsHex               ▷ Hexadecimal peerIDs of known peers
    peerIDs                  ▷ PeerIDs of known peers

Upon Init() do:
    payloadSize ⟵ ps
    prepareTime ⟵ pt
    cooldownTime ⟵ ct
    runTime ⟵ rt
    broadcastInterval ⟵ bi
    randomSeed ⟵ rs
    nPeers ⟵ n_peers
    idBits ⟵ id_bits
    peerIDsHex ⟵ {}
    peerIDs ⟵ {}
    Setup Timer StartTimer(prepareTime)

Upon Timer StartTimer() do:
    sendMessageTimer ⟵ Setup Periodic Timer SendMessageTimer(0,
broadcastInterval)
    Setup Timer StopTimer(runTime)

Upon Timer SendMessageTimer() do:
    toSend ⟵ generatePayload(0, payloadSize)
    payload ← toSend.getBytes()
    destination ← ⊥
    while (destination = ⊥) do
        d ← r.nextInt(peerIDsize)
        if (d ≠ processSequenceNumber) then
            destination ← peerIDs[d]
    mid ← generateMid()
    send ← Send(peerID, dest, mid, payload)
    Trigger p2pRequest(send)

Upon p2pDeliver(msg) do:
    Call Log(msg)

Upon Timer StopTimer() do:
    Cancel Timer SendMessageTimer
    Setup Timer ExitTimer(cooldownTime)

Upon Timer ExitTimer() do:
    Call Exit()
```

4 POINT-TO-POINT PROTOCOL

4.1 Initialization

The Point-to-point Protocol facilitates message exchange between peers in a decentralized network. This protocol is structured to manage various states and requests associated with sending and receiving messages.

The Point-to-point layer begins with an initialization phase, where it sets up the protocol's initial state, including peer information, message queues for pending and received messages, and a timer for resending messages.

4.2 Message Received from App Layer

When a message sending request is received (sendRequest), from the Application Layer, the protocol first checks whether the Distributed Hash Table (DHT) is initialized. If the DHT is not yet ready, the request is stored in a queue for later processing. Once initialized, the protocol triggers a lookup to identify the destination peer for the message.

The Peer-to-peer and the DHT Layer share a tcp channel, since it is pointless and redundant to have two separate channels for their communication. When the DHT layer is setup, it triggers the notificaiotn channelCreated, , and the message handlers for this protocol are set.

Not to confuse this concept with the DHT Initiliaziation, which triggers the dhtInitiliazed in the peer-to-peer layer, in this case, this notification gets triggered when the Chord structure is ready to receive and look up messages, which only happens when there is at least 2 nodes. Otherwise, the messages get stored in sendRequest as mentioned before. When the DHT is initialized, all the messages that were pendling, are finally processed one by one.

4.2.1 *Point-to-point Pseudocode.*

```
Interface:
    requests:
        p2pSend(s)
    indications:
        p2pDeliver(d)

State:
    thisHost                    ▷ This peer host information
    tcpChannelId                ▷ ID of the TCP channel
    msgsPendingLookup           ▷ Messages queueing for lookup until DHT layer is initialized
    msgsPendingLookupReply      ▷ Messages awaiting reply from DHT lookups
    receivedMsgs                ▷ Received and delivered Point2Point messages
    p2pMessagesPendingAck       ▷ Messages sent and awaiting Ack & their corresponding backup helper node
    helperMsgsToSend            ▷ Messages stored to eventually send as a helper
    helperMsgsSending           ▷ Messages I am currently sending as a helper
    myHelpersMsgs               ▷ Messages I sent to my helpers
    isDHTInitialized            ▷ Initialization state of the DHT layer

Upon Init() do:
    thisHost ← h
    msgsPendingLookup ← {}
    msgsPendingLookupReply ← {}
    receivedMsgs ← {}
    p2pMsgsPendingAck ← {}
    helperMsgsToSend ← {}
    helperMsgsSending ← {}
    myHelpersMsgs ← {}
    isDHTInitialized ← ⊥
    Setup Periodic Timer ResendMessagesTimer(3, 3)

Upon p2pSend(r) do:
    if (isDHTInitialized = ⊥) then
        msgsPendingLookup ← msgsPendingLookup ∪ {r}
        return
    lookUpRequest ← r.getDestPid(), r.getMessageId()
    Trigger dhtLookupRequest(lookUpRequest)
    msgsPendingLookupReply[r.getMessageId()] ← r

Upon channelCreated(c) do:
    tcpChannelId ← c.getChannelId()
    registerSharedChannel(tcpChannelId)
     // Register Message Handlers
     // p2pMsg, helperMsg, p2pAckMsg

Upon dhtInitialized(n) do:
    isDHTInitialized ← n.isInitialized()
    if (isDHTInitialized = ⊥) then
        return
    foreach m ∈ msgsPendingLookup do:
        Trigger p2pSend(m)
    msgsPendingLookup ← {}
```

4.3 P2P Message Handling

When the DHT lookup responses are received, the dhtLookupReply processes the response, identifying the target and helper peers before sending messages accordingly.

The P2P protocol incorporates a helper node that is returned along with the designated target for the message. The primary function of this helper node is straightforward: it sends the message on behalf of the sender.

To minimize the amount of messages in the network, this only happens in event of a failure from the sender (as indicated by p2pMessageFail) - more precisely, the original sender attempts to send the payload to the target node, and, only in the event that first message fails, the payload is then shared with the designated helper node (usually the direct predecessor of the target) who stores the message(s) until (if) the original sender goes down. After that moment, the helper node starts transmitting the original sender's message. This minimizes redundant messages over the network, which can have especially heavy load on larger payload sizes.

More precisely, if the original sender does fail, the helper node adds the message that the sender was supposed to transmit to its own messages to send, which are sent periodically when the timer ResendMessages gets triggered. The helper sends all the messages to the nodes he is helping when the ResendMessageTimer gets triggered, including all his other messages, that are not send to nodes he is helping but to be delivered to the App Layer directly. Since we can't have duplicate messages, in this case, the first message that gets through is delivered to the App Layer, while the second triggers the p2pAckMessages, which will remove this message from the helper or the original sender, as it is duplicated, and there is no point in continuing to resend it.

In the case of the destination being unavailable or the Chord Ring not being initialized, the message is retried upon the Timer ResendMessages.

With these two cases, we manage to mask failures, from the sender and the receiver, and in doing so increase the robustness of our Peer-to-Peer point-to-point communication, as it is important that all peers have access to all resources in our system, which only happens if messages don't get lost.

Upon receiving a message through the p2pMessage(m, h) function, the protocol first checks whether the message has already been received. If it has, an acknowledgment is sent back to the sender, allowing the sender to remove the message from their queue of messages to send, thereby preventing duplicated deliveries. If the message is new, it is delivered to the application layer for further processing. The handling of acknowledgments is managed by the p2pAckMessage function, which removes the acknowledged message from the pending acknowledgment queue, ensuring efficient message management.

In summary, this protocol provides a robust mechanism for peer-to-peer communication, enabling efficient message delivery while handling scenarios like message acknowledgment, failures, and dynamic peer management within a decentralized system.

Next, we present the pseudocode for this protocol, which outlines the key functions and their interactions within the point-to-point communication layer.

```
Upon p2pMessage(m, h) do:
  if (m.getMid() ∈ receivedMsgs) then
    Trigger Send(ACK_MSG, m, from)
    return
  Trigger p2pDeliver(m)
  receivedMsgs ⟵ receivedMsgs ∪ m.getMid()
  Trigger Send(ACK_MSG, m, from)

Upon p2pAckMessage(m, h) do:
  p2pMsgsPendingAck ⟵ p2pMessagesPendingAck \ {m}
  helperMsgsSending ⟵ helperMsgsSending \ {m}

Upon helperMsg(m, h) do:
  helperMsgsToSend[h] ⟵ helperMsgsToSend[h] ∪ {m}

Upon p2pMessageFail(m, h) do:
  helperHost ⟵ p2pMessagesPendingAck[m]
  helperMsg ⟵ helperMsg(m)
  if (helperHost = ⊥) then
    return
  if (helperMsg ∈ myHelperMsgs[helperHost]) then
    return
  if ((helperHost ≠ m.getDestination()) and helperHost ≠ thisHost) then
    myHelpersMsgs[helperHost] ⟵ myHelpersMsgs[helperHost] ∪ helperMsg
    Trigger Send(HELPER_MSG, helperMsg, helperHost)

Upon dhtLookupReply(r) do:
  send ⟵ msgsPendingLookupReply[r.getMid()]
  msgsPendingLookupReply \ {r.getMid()}
  if (send = ⊥) then
    return
  h ⟵ r.peers[0]
  t ⟵ r.peers[1]
  p2pMsg ⟵ p2pMessage(send, thisHost, t.host)
  if (t.host = thisHost) then
    Trigger p2pMessage(p2pMsg, t.host)
    return
  p2pMsgsPendingAck[p2pMsg] ⟵ {h.host}
  Trigger Send(P2P_MSG, p2pMsg, t.host)

Upon peerDown(n) do:
  msgs ⟵ helperMsgsToSend[n.getPeer()]
  if (msgs = ⊥) then
    return
  helperMsgsSending ⟵ helperMsgsSending ∪ msgs
  msgs ⟵ {}

Upon Timer ResendMessagesTimer() do:
  if (isDHTInitialized = ⊥) then
    return
  foreach m ∈ p2pMsgsPendingAck do:
    Trigger Send(P2P_MSG, m, m.getDestination())
  foreach m ∈ helperMsgsSending do:
    Trigger Send(P2P_MSG, m, m.getDestination())
```

5 CHORD DHT PROTOCOL

5.1 Initialization

The initialization process sets up the initial state of a new Chord node. A unique identifier for the node, denoted as n, is generated, typically based on a hash of the node's IP address and port. At this stage, the node's predecessor, the successor and the successor of its successor are both set to point to itself, in Chord, a node must always be aware of it's predecessor and successor, so this step is crucial. Additionally, the isInit flag indicates whether a node in the DHT Layer has been successfully initialized, which only happens, when it connects to another node in the chord ring.

Next, the finger table is created based on the number of bits specified in the property idBits. Each entry in this finger table points to another node, is successor. This table allows the node to skip over several other nodes in a single step, resulting in logarithmic(O(n)) search times. The initialization phase also involves setting up various timers to handle periodic tasks. The RetryLookupsTimer is responsible for retrying any lookup requests that haven't been resolved, while the StabilizeTimer ensures that the node's successor information remains current. Additionally, the FixFingersTimer is set to periodically update the finger table to maintain accurate routing information. Finally, for every node except the first, a contact property must be provided, the newly initialized node sends a message to locate its successor, thereby integrating itself into the existing network.

5.2 Stabilization

The Stabilize process is employed To facilitate proper functioning within the network. This process is called periodically to ensure the node's successor information is accurate. During each stabilization call, the node contacts its successor to confirm that it is still active and reachable. If the successor identifies a new successor, the node updates its own successor reference accordingly. Furthermore, the node checks its predecessor to see if it is still valid and updates its predecessor pointer if necessary.

5.3 Update Fingers

The protocol also incorporates a mechanism to keep the finger table up-to-date, managed through the FixFingers timer. The purpose of this function is to optimize the finger table. When a new node joins the Chord Ring, it initially only knows its immediate successor and predecessor, and they, in turn, are aware of the new node's existence, however, the rest of the nodes in the network are not. To enhance look up process efficiency, FixFingers updates the finger table entries with information about other nodes in the network. We have set it to run every three seconds to avoid overwhelming the network, as this process can be resource-intensive.

5.3.1 Chord DHT Pseudocode.

```
Interface:
  Requests:
    dhtLookupReq(r)
  Indications:
    dhtLookupReply(r)
    dhtInitialized(isInit)
    peerDown(dc)

State:
  pre                       ▹ Predecessor of this node
  sucSuc                    ▹ Successor of the successor of this node
  n                         ▹ The chord node of this host/process
  fingers                   ▹ Finger table of this node
  lookupsPending            ▹ Lookup requests pending response
  fixFingersPending         ▹ Nodes pending fix fingers update
  isInit                    ▹ If this node is initialized or not
  tcpId                     ▹ TCP ID of the channel

Upon Init(props) do:
  lookupsPending ⟵ {}
  fixFingersPending ⟵ {}
  isInit ⟵ $\perp$
  pid ⟵ props.nodeId
  n ⟵ {pid, h}              ▹ h = host of this node
  pre ⟵ n
  sucSuc ⟵ n
  fingers ⟵ {}
  fingersNr ⟵ props.idBits
  end ⟵ $(n.pid + 2^{fingersNr}) \bmod 2^{\text{fingersNr}}$
  For $i$ = fingersNr $- 1$ to 0
    start ⟵ $(n.\text{pid} + 2^{i}) \bmod 2^{\text{fingersNr}}$
    fingers[$i$] ⟵ {start, end, $n$}
    end ⟵ start
  EndFor
  Trigger dhtChannelCreated
  Setup Timer RetryLookupsTimer(3, 3)
  Setup Timer StabilizeTimer(1, 3)
  Setup Timer FixFingersTimer(3, 3)
  if ({c} ∈ props.contact) then
    m ⟵ {random.uuid, n.pid}
    Trigger Send(FIND_SUCC, m, c)

Upon LookupRequest(r) do:
  m ⟵ {r, n}
  lookupsPending ⟵ lookupsPending ∪ {r}
  Trigger dhtFindSuccessorMsg(m, m.host)

▹ OpenInterval is similar to belongsTo, but with an open interval (s, e) instead of (s, e]
belongsTo(s, e, k):         ▹ start, end, key
  return $k \in (s, e]$ or $(s > e$ and $k \in (s, e])$

Upon Timer StabilizeTimer(t, tid):
  if init = $\perp$ then
    return
  m ⟵ { uuid.random, n }
  Trigger Send(GET_PRED, m, fingers[0].node.host)

Upon Timer FixFingersTimer(t, tid):
  if isInit = $\perp$ then
    return
  i ⟵ random index $> 1$ into fingers
  u ⟵ uuid.random
  fixFingersPending[u] ⟵ {fingers[i]}
  m ⟵ {u, n.host, n.host, fingers[0].start}
  Trigger dhtFindSuccessorMsg(m, n.host)

Upon Timer RetryLookupsTimer(t, tid):
  foreach p ∈ lookupsPending do:
    Trigger dhtFindSuccessorMsg(p, p.originalSender)
```

5.4 Scalable Key Lookup

Following initialization, the protocol handles incoming lookup requests. When a lookup request is received, it is recorded in the lookupsPending list. The FindSuccessorMessage then determines whether the key being requested lies between the node's identifier and its successor's. If it does, the node is able to return the requested value directly to the sender(in the peer-to-peer layer), concluding the lookup process. In cases where the key does not lie within this range, the node uses its finger table to select the closest finger preceding the ID of the message key and triggers FindSuccessor, this is a recursive process. Upon finding the successor, it triggers the foundSuccessor function

The found successor has three separate cases to consider. If it's from a node that is not initialized, in the process of joining, it initializes itself by setting up its predecessor and successor, which it gets from the FoundSuccessorMsg sent by the closest predecessor found in FindSuccessorMessage. Afterwards, it notifies the Peer-to-Peer layer that the DHT layer of this node is ready and can start receiving lookup requests.

If the message was sent by FixFingers, then it updates the node's fingers to optimize the routing and efficiency of DHT lookups.

Finally, if we are not in either case, then we have found the successor of the message with the target ID, requested in a Look up request by the Peer-to-peer Layer, and so, we reply to that request.

The outConnectionDown, event triggered when a node disconnects, executes several important steps to ensure the integrity and efficiency of the Chord network. First, the function iterates through the fingers, if a finger node matches the disconnected node, the finger is updated to point to the current node (n). This step is crucial for maintaining the accuracy of the finger table, ensuring that the node can still efficiently route messages through the network. Next, if the successor node corresponds to the current node's host, which happens in the previous step if it was the disconnect node, it is updated to point to the successor of the successor node. This is important because the successor of the successor node acts as a backup helper node for the current sender, providing redundancy in case of failure during communication. This ensures that the P2P layer remains functional, because the helper node will deliver the message in case the original sender of the message fails to. Additionally, if the successor matches the current node's host, the predecessor node is set to the current node. This is an edge case that occurs only when that is the only node in the Chord ring, in which case the DHT is not initialized, and isInit is set to false. Afterwards, we signal the P2P layer by triggering the dhtInitialized(isInit) function, allowing it to stop sending messages until a second node joins the Chord ring, at which point the transmission of messages can resume.

In summary, the Chord DHT Protocol establishes a scalable framework for efficiently processing lookup requests. Its initialization process sets up the necessary structures and relationships among nodes, while its lookup, stabilization, and finger table management functions facilitate the efficient resolution of these requests in a dynamic environment

```
Upon dhtGetPredecessorMsg(m, h) do:
    pm ⟵ {m.pid, pre, n, fingers[0].node}
    Trigger Send(RETURN_PRE, pm, m.sender)

Upon dhtFindSuccessorMsg(m) do:
    if isInit = ⊥ or belongsTo($n.pid, fingers[0].node.pid, m.k$) then
        sm ⟵ {m, n, fingers[0].node}
        Trigger Send(FOUND_SUCC, sm, sm.originalHost)
        return
    if belongsTo($pre.pid, n.pid, m.k$) then
        fm ⟵ {m, pre, n}
        if (m.originalSender = n.host) then
            Trigger dhtFoundSuccessorMsg(fm, n.host)
            return
        Trigger Send(FOUND_SUCC, fm, fm.host)
        return
    c ⟵ closestPreNode(m.k)
    fi ⟵ {m, n.host}
    Trigger Send(FIND_SUCC, fi, c.host)

Upon FoundSuccessorMsg(m, f) do:
    if isInit = ⊥ then
        pre ⟵ {m.senderPid, m.senderHost}
        fingers[0].node ⟵ {m.successorPid, m.successorHost}
        isInit ⟵ ⊤
        Trigger dhtInitialized(isInit)
        return
    if {m.mid} ∈ fixFingersPending then
        Call fixFinger(m)
        return
    lookup ⟵ {m}
    if m.senderPid = n.pid then
        lookup ⟵ lookup ∪ {pre.senderPid, pre.senderHost}
    else
        lookup ⟵ lookup ∪ {m.senderPid, m.senderHost}
    lookup ⟵ lookup ∪ {m.succPid, m.succHost}
    Trigger dhtLookupReply(lookup)
    lookupsPending ⟵ lookupsPending \ {m.mid}

Upon dhtReturnPreMsg(m, h) do:
    if inOpenInterval($n.pid, fingers[0].node.pid, m.prePid$) then
        fingers[0] ⟵ {m.prePid, m.preHost}
    sucSuc ⟵ {m.sucPid, m.sucHost}
    m ⟵ {generateMid(), n}
    Trigger Send(NOTIFY_SUCC, m, fingers[0].node.host)

Upon dhtNotifySuccessorMsg(m, h):
    if isInit = ⊥ or inOpenInterval($pre.pid, n.pid, m.senderPid$) then
        pre ⟵ {m.senderPid, m.sender}
        if isInit = ⊥ then
            fingers[0].node ⟵ pre
            isInit ⟵ ⊤
            Trigger dhtInitialized(isInit)

Upon outConnectionDown(dc):
    foreach f ∈ fingers do:
        if finger.node.host = dc then
            finger.node ⟵ n
    if fingers[0].node.host = n.host then
        fingers[0].node ⟵ sucSuc
    if fingers[0].node.host = n.host then
        pre ⟵ n
        isInit ⟵ ⊥
        Trigger dhtInitialized(isInit)
    Trigger peerDown(dc)
```

6 TESTING

We conducted independent tests on networks with 10, 50, 100, 150, and 200 nodes. From the beginning, one of the main goals of our project was for our system to comfortably handle 150 nodes.

We tested two scenarios by adjusting payload sizes and message broadcasting intervals. Except for the 200-node case, which was only tested for payloads with 100bytes, mainly to confirm that our system could handle 200 nodes as effectively as it did 150, and our results suggest that it indeed performed as expected.

Each test case was repeated three times to establish a reliable baseline, as a single run could be misleading and might not accurately reflect our system's performance

6.1 Testing Scenarios

- **Scenario 1**
  - **Payload Size:** 100 bytes
  - **Broadcast Interval:** 1 second
- **Scenario 2**
  - **Payload Size:** 1024 bytes
  - **Broadcast Interval:** 10 seconds

- **Scenario 1(100 bytes payload)**
- **Scenario 2 (1024 bytes payload)**:

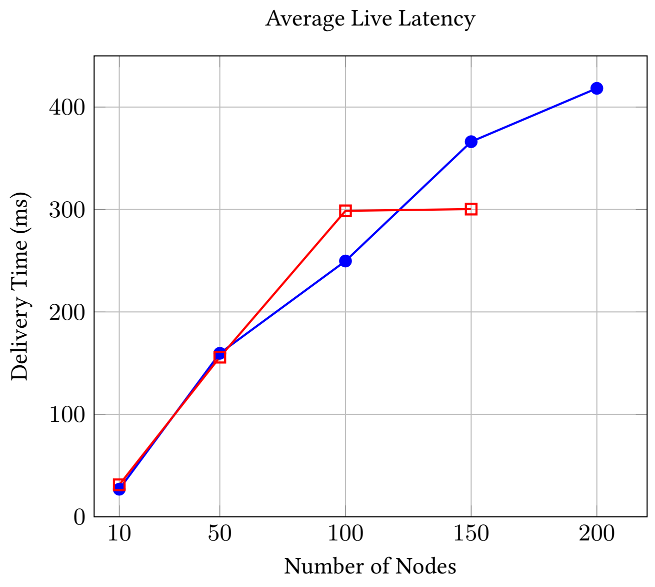


Figure 1: Time for delivering a message when both peers are online

Table 1: Scneario 1 Network Statistics

Nodes	Sent P2P	Received P2P	Total Messages	P2P Redundancy
10	6000	5964	84947	0.99
50	27440	27342	473206	0.999
100	56349	56149	1004610	0.999
150	76572	76260	1480380	0.999
200	94458	94049	1824929	0.99

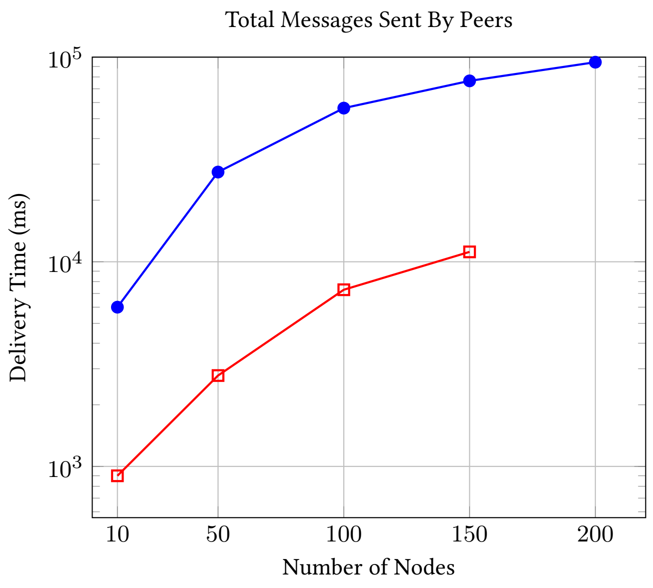


Figure 2: Time for delivering a message when both peers are online

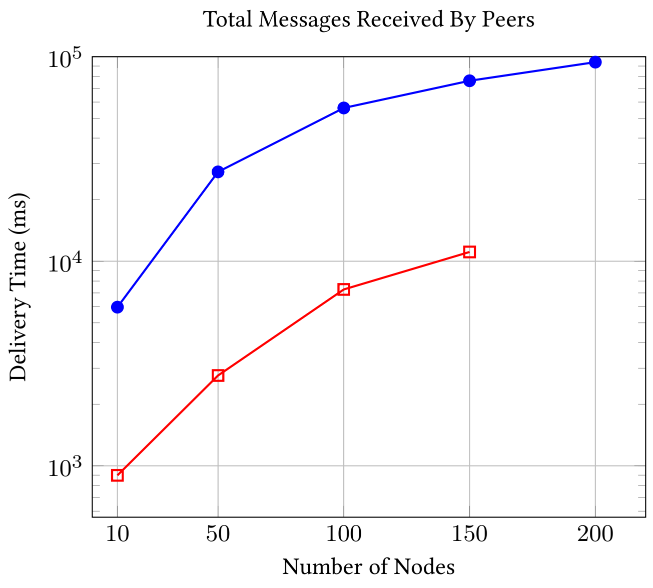


Figure 3: Time for delivering a message when both peers are online

Table 2: Scneario 2 Network Statistics

Nodes	Sent P2P	Received P2P	Total Messages	P2P Redundancy
10	900	898	27275	0.999
50	2780	2763	94149	0.99
100	7301	7281	259911	0.999
150	11178	11100	409369	0.99

7 CONCLUSIONS

We are extremely pleased with our implementation of the Peer-to-Peer DHT Communication System, as it addresses most of the challenges presented to us, with a blend of readable, extensible, and efficient code. Our program demonstrated full scalability when run locally, and on the cluster. In every experiment, the protocols performed and acted robustly to any changes in the system, as expected.

Our implementation was finalized on the 24th, though we could only submit this report, concluded with our experiments, today due to certain complications we encountered within the cluster. In hindsight, for future collaborations, we will aim to begin testing our scripts and configurations well in advance rather than days before the deadline. Due to the time limitations in setting up our scripts and configurations, we conducted shorter tests to ensure we captured a large range of results to present. This approach led to somewhat unrealistic testing conditions—primarily because most of our tests had only a 1-second interval between processes starting, which does not accurately represent real-world scenarios. As a result, our finger tables could not keep up with the rapid starting and stopping of processes, with each process starting and terminating within one second of the previous one. The finger table only updates and stabilizes every 3 seconds, which meant it could not adapt quickly enough, leading to some lost messages. This is reflected in the Average Live Latency plot, which ideally should exhibit a logarithmic pattern. However, due to the suboptimal testing conditions we established, primarily, the only 10 total minutes of the tests, plus, the 1 second timing between processes starting (and eventually stopping) - the finger tables lacked sufficient nodes in order to be full, and naturally, were quite far from being optimal - resulting in shorter, less optimal jumps between nodes.

Had the experiment been conducted over a longer duration (at least 30 minutes), we would expect to see a logarithmic progression of live latency - as the finger tables would have had adequate time to stabilize and fill with optimal and "fresh" nodes at each interval. In fact, the bare minimum amount of time required to fill out the 256-long finger table - if all "random" indexes were uniquely chosen, only once, for every 256 iterations - would be around $3 \times 256 = 768$ seconds (3 seconds between each stabilization iteration), or around 12.8 minutes. Which still does not take into account the freshness of the entries, since most of the fingers would/could be relatively sub-optimal or outdated by a couple of minutes, as nodes entered and left the network.

Under more realistic conditions, with a 15/30-second interval between node joins and leaves, these issues would not occur, and our tests would yield exactly 1.0 redundancy. However, as described, the high frequency of process terminations in our setup distorted what could have been optimal test results.

ACKNOWLEDGMENTS

We would like to thank our Distributed Systems Algorithms course professor, João Leitão, for his support throughout this project, both during and outside of classes. His teaching style was excellent throughout this first half of the semester, as a professor who pointed us in the right directions, 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 the foundational research for the project, such as the Chord and Kademlia papers, which served as a blueprint for our work.

REFERENCES

[1] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. (2002).

[2] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Paper. *Digital Libraries* (Aug. 2001), 12. https://doi.org/1-58113-411-8/01/0008

[3] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. 2005. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management* 13, 2 (June 2005), 197–217.