Kademlia and Chord: DES Performance Evaluation

Gabriele Masina
University of Trento
220496

Alex Zanetti
University of Trento
220225

Giovanni Zotta
University of Trento
223898

***Abstract*—Kademlia and Chord are peer-to-peer Distributed Hash Table (DHT) protocols. Both promise to distribute the load and storage of data among many hosts, scaling horizontally. In this assignment we explore their differences and similarities, implementing the protocols according to their original papers and conducting a basic performance evaluation analysis using a Discrete Event Simulation framework.**

I. Introduction

The goal of this work is to implement the Kademlia [1] protocol and simulate its behaviour to conduct a performance evaluation analysis on it. In order to have a protocol to which compare Kademlia to, we also implement the Chord DHT [2]. This assignment focuses on the implementation of the two DHTs in a Discrete Event Simulation environment. We also provide a basic performance evaluation analysis on some of the most important metrics like message delay and network load. Further performance evaluation studies will be carried out as a project for the course on Simulation and Performance Evaluation. The codebase for the project is available on GitHub at https://github.com/GiovanniZotta/kademlia.

II. Theory background

Kademlia and Chord can seem to be quite different at a first glance, but the reality is that they share many similarities. Both protocols have similar qualities: each node is assigned an ID in a 160-bit space and it is responsible for a fair proportion of the key space (depending on how redundant the storage of keys is). $O(\log n)$ messages are expected for key lookup and store operations. In Kademlia each node stores routing information about $O(k \cdot \log n)$ nodes, even though this is an upper bound since most of the k-buckets will not be filled to the brim. In Chord each node stores routing information about $O(\log n)$ other nodes in the network. The differences start to arise when we look at how the routing tables are maintained: an advantage of Kademlia is that its nodes can exploit normal messages to refresh the k-buckets while Chord nodes need to regularly update their finger tables. Kademlia can also exploit multiple parallel queries to find the value associated to a key thanks to the structure of the tree, providing better resistance to node crashes and potential misbehaviour of nodes. The advantages of the redundancy provided by the Kademlia protocol can be a double-edged sword, since it is also possible to have consistency problems since keys are stored in many nodes and therefore some key republishing algorithm has to be employed. Our Chord implementation does not provide any data redundancy by default and each key lookup follows the finger table in an iterative manner; this could be improved by using multiple hash functions to give each node multiple IDs and by querying the routes to them in parallel. This would bring the amount of routing information stored by each Chord node to $O(r \cdot \log n)$ where r is the number of hash functions used.

III. Simulation framework

We based our simulation framework on the SimPy Python library [3], which provides an intuitive interface for discrete-event simulations where different processes can interact. On top of it, we built a framework that allows us to think and write the code at an higher abstraction level, very similarly to what other agent-based frameworks provide. Differently from that, our simulation is discrete event-based, meaning that each process is just a generator of events, and the SimPy library manages the events queue to orchestrate the whole simulation.

A. General infrastructure

The system is composed of a (fixed) set of DHT nodes (Chord or Kademlia) and of clients arriving at random time instants. Clients are processes spawned by the simulator, which performs either a `FIND_VALUE` or a `STORE_VALUE` request to a random DHT node. When performing such a request, they spawn a new process on the chosen node with the corresponding node's method and start waiting for a response. The DHT node performs some actions to satisfy the request, and finally responds to the received request.

As in a realistic scenario, each DHT node can serve just a limited number c of packets in parallel, and processes that require complex exchanges of packets can be carried out in a parallel manner, as long as no more than c packets are served simultaneously. This behaviour can be modeled by a $M/M/c$ queue with infinite buffer (although it is possible to have at most K packets in queue, turning the model into a $M/M/c/K$ queue), where the service time of the queue varies according to the job the node is performing. We simulated this scenario by using SimPy Resources, as we explain in the next sections.

B. Message layer

We implemented a class `Node` with some primitives to abstract the message layer.

```
class Node(Loggable):
    ...
```

`Node` has a `send_req` method that creates a `sent_req` event that will be triggered when the recipient has served the request. This process spawns an asynchronous[1] process that first waits a random transmission time and then starts a new process on the recipient with the given answer callback `answer_method`.

Each message received by a `Node` first gets inserted in the queue of the recipient, which processes them in parallel with capacity C.

```
def _transmit(self) -> SimpyProcess[None]:
    # wait some random transmission time
    transmission_time = self.rbg.get_exponential(
        self.mean_transmission_delay)
    transmission_delay = self.env.timeout(
        transmission_time)
    yield transmission_delay


def _req(
    self,
    answer_method: Method[SimpyProcess[None]],
    packet: Packet,
    sent_req: Request
) -> SimpyProcess[None]:
    packet.sender = self
    # wait a random time
    yield from self._transmit()
    # then spawn recipient callback
    yield self.env.process(
        answer_method(packet, sent_req))


def send_req(
    self,
    answer_method: Method[SimpyProcess[None]],
    packet: Packet
) -> Request:
    sent_req = self.new_req()
    # simulate asynchronous send
    self.env.process(
        self._req(answer_method, packet, sent_req))
    return sent_req
```

Once the recipient has served the request, it can call the `send_resp` method that spawns an asynchronous process that will trigger the response event after waiting a transmission time drawn from some distribution (exponential distribution by default).

```
def _resp(
    self,
    recv_req: Request,
    packet: Packet
) -> SimpyProcess[None]:
    packet.sender = self
    yield from self._transmit()
    recv_req.succeed(value=packet)


def send_resp(
    self,
    recv_req: Request,
    packet: Packet
) -> None:
    self.env.process(self._resp(recv_req, packet))
```

[1]It is a separate process that advances independently from the current one, but they share the same temporal line.

In the meanwhile, the sending process can wait for the response to arrive by calling the `wait_resp` method, which will return the received response packet or raise an exception if the request times out. There is also the possibility to wait for a list of sent requests, and in this case a list is filled with the packets received within the timeout.

```
def wait_resps(
    self,
    sent_reqs: Sequence[Request],
    packets: List[Packet]
) -> SimpyProcess[None]:
    # wait for all the responses, but set a timeout
    sent_req = self.env.all_of(sent_reqs)
    timeout = self.env.timeout(self.max_timeout)
    wait_event = self.env.any_of((timeout,
                                  sent_req))
    ans = yield wait_event
    # collect packets received within the timeout
    timeout_found = False
    for event, ret_val in ans.items():
        if event is timeout:
            timeout_found = True
        else:
            packets.append(ret_val)
    if timeout_found:
        raise DHTTimeoutError()


def wait_resp(
    self,
    sent_req: Request
) -> SimpyProcess[Packet]:
    ans: List[Packet] = []
    yield from self.wait_resps([sent_req], ans)
    return ans.pop()
```

We also have a `@packet_service` decorator to decorate methods that serve packets. We wrap such methods because we can serve just a limited number of requests concurrently. To do so, the method first has to acquire the node's resource, which is our way to implement the $M/M/c$ queue [4]. Once the resource has been reserved it can perform the operations and finally, after some random service time, the resource is released.

```
def packet_service(
    operation: Method[T]
) -> Method[SimpyProcess[T]]:
    def wrapper(
        self: DHTNode,
        *args: Any
    ) -> SimpyProcess[T]:
        with self.in_queue.request() as res:
            # wait for the queue to be free
            yield res
            # do what we need to do
            ans = operation(self, *args)
            # wait some service time
            service_time = self.rbg.get_exponential(
                self.mean_service_time)
            yield self.env.timeout(service_time)
        return ans
    return wrapper
```

C. An example

```
@packet_service
def recv_callback(
    self,
    packet: Packet,
    recv_req: Request,
) -> None:
    # echo
    self.send_resp(recv_req, packet)

@packet_service
def action1(
    self,
    other: Node
) -> Request:
    # send empty packet
    packet = Packet()
    sent_req = self.send_req(other.recv_callback,
                             packet)
    return sent_req

@packet_service
def action2(
    self,
    packet: Packet
) -> None:
    self.log(f"Received answer {packet}")

def exchange(
    self,
    other: Node
) -> SimpyProcess[None]:
    # send req
    sent_req = yield from self.action1(other)
    # wait for response
    packet = yield from self.wait_resp(sent_req)
    # serve response
    yield from self.action2(packet)
```

This simulation framework allows us to have complete control over the experiment environment: we can decide the statistical distribution of client requests as well as the distribution of service time for each type of action a node has to perform, while still having realistic delay and load simulations thanks to the $M/M/c$ communication model. The fact that the simulator is pretty lightweight allows us to run simulations with a large number of nodes, which would otherwise be unfeasible with an emulated approach.

D. Kademlia protocol

The main procedure of the Kademlia protocol is the `FIND_NODE` algorithm which has been implemented according to the specification of the original paper [1]. Algorithm 1 shows the pseudocode of our basic implementation, which lacks some of the proposed optimizations explained in the next Section.

When a node receives a `FIND_NODE` request, it will respond with the k nodes closest to the key (k being the system-wide parameter describing at most how many nodes should be in a k-bucket). To get such information, it follows an algorithm that asks α nodes for the k nodes they know that are closest to the the required key. The answers are collected and put in a set of nodes C, together with the content of S. From the set C we keep only the k nodes closest to key. Finally, C becomes S for the next round. The algorithm stops at the round where the set S has not changed with respect to the previous round of α requests.

Algorithm 1: Kademlia find_node iterative procedure

Input : KEY key
1 SET $S \leftarrow k$ nodes from buckets closest to key
2 BOOL $found \leftarrow$ **false**
3 **while not** $found$ **do**
4 SET $askto \leftarrow \alpha$ nodes not contacted from S
5 **send** `ASK_CLOSEST`(key) request to $askto$
6 SET $packets \leftarrow$ **wait** for $askto$ to respond
7 SET $C \leftarrow S \cup \bigcup_{p \in packets} p.nodes$
8 $C \leftarrow k$ nodes from C closest to key
9 $found \leftarrow C = S$
10 $S \leftarrow C$
11 **end**
12 **return** S

The join procedure is straightforward: some node v that wants to join the network has to be aware of at least one node u already participating. The node v performs a `FIND_NODE` for the key corresponding to its own ID. Since v knows only about u, the latter will be the only entry in v's buckets, thus the only node contacted in the first round. There is no need for an explicit update procedure like in Chord, since the Kademlia protocol takes care of it via normal traffic.

E. Kademlia missing features

For the sake of simplicity, we have chosen to omit some Kademlia features that we think are either not relevant or not interesting for our performance evaluation studies:

- Key republishing: since we do not plan to use our implementation for production use, we preferred to keep things simple without adopting any periodic time-based key republishing algorithm, even though this could hamper data consistency in a real setting;
- Key caching: we do not cache keys in nodes that should not be responsible for them upon a successful lookup;
- Ping of the least recently seen node in the k-bucket upon reception of a message from a node which would have fitted in that k-bucket: this is useful in case of DoS attacks that try to flush routing tables but this is outside the scope of our project. Instead, we always refresh the buckets with the new node contacting us;
- Accelerated lookups: we always consider the IDs to query one bit at a time instead of b bits at a time as is suggested in section 4.2 of the Kademlia paper;
- In `FIND_NODE` we do not start round $i+1$ if we have not completed round i. The Kademlia paper states that round $i+1$ can begin as soon as one response from round i comes in. In our case, we wait for all α responses from round i before proceeding to round $i+1$.

F. Chord protocol

Our implementation of the Chord protocol is similar to the one we have seen in class: each node has one ID and is

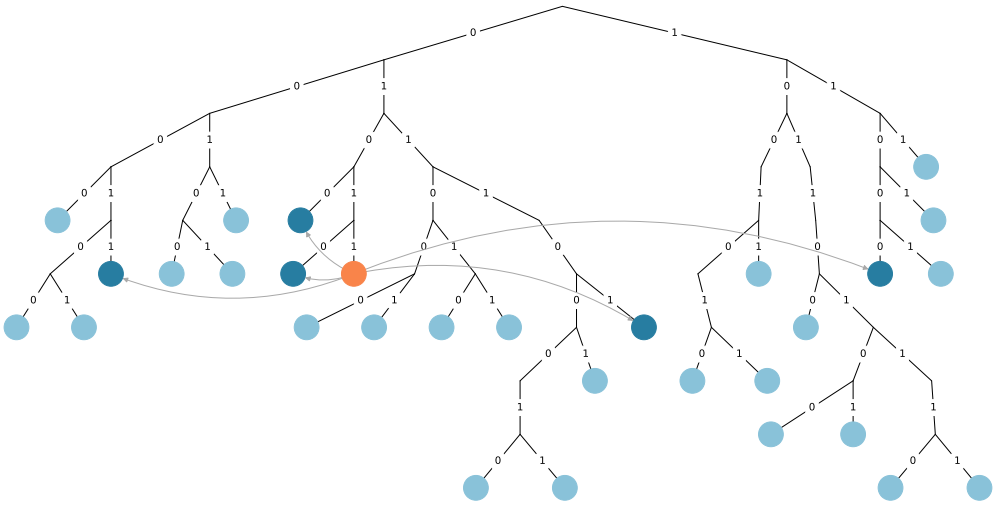

Figure 1. Kademlia tree from the perspective of the orange node. Each blue node represents the first node of each k-bucket of the orange node. This plot has been obtained during our simulations thanks to NetworkX.

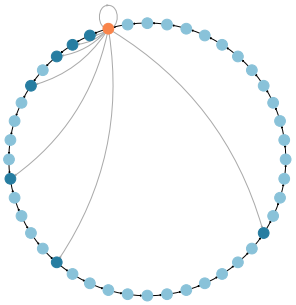

Figure 2. Chord ring from the perspective of the orange node. Each blue node represents a member of the finger table of the orange node. This plot has been obtained during our simulations thanks to NetworkX.

responsible for the set of keys between its ID and the ID of its successor. The routing of requests happens through a finger table that is updated before the start of the experiment.

G. Chord missing features

- Multiple node IDs: each node could have multiple IDs and this would lead to better redundancy and parallel queries for lookups, making Chord more similar to Kademlia. This, of course, brings the same challenges of data consistency;
- Periodic table updates: in order to be perfectly accurate, every node should update its routing table as soon as a new node joins ore leaves the network. This would of course result in a large amount of traffic and this is solved by performing periodic time-based updates, which we are not interested in;
- Runtime join and leave: we do not support joining the network at runtime. This feature will most likely be implemented in the future for the project of the course on Simulation and Performance Evaluation to evaluate the behaviour of Chord in a situation of high churn.

IV. Experiment setup

The setting of our experiments is the following:

- Static set of nodes: no nodes join or leave during the experiments;
- Clients perform read/write requests to a random node;
- Clients arrival rate follows an exponential distribution;
- Service time for packets follows an exponential distribution;
- Transmission time follows an exponential distribution;
- DHT nodes can fulfill one request at a time.

Since we are interested in analysing the client waiting times and the load of the network, we collect the following data:

- For each successful client request, we collect the elapsed time between issuing the request and receiving the response, along with the number of hops the network took to deliver the answer and the number of unsuccessful client requests that timed out;
- The DHT node's queue load over time.

Initially, for both Kademlia and Chord the network is composed of two hardwired DHT nodes and the rest of the nodes join by asking one of the already joined nodes. The whole process is simulated like the rest of the experiment, but we do not collect data during this phase to avoid polluting our analysis.

In this phase, Chord has to run an `update` procedure for each node after wiring the ring to fill the finger tables, which takes a large amount of time. On the other hand, in Kademlia the buckets are updated on-the-fly by the `process_sender`

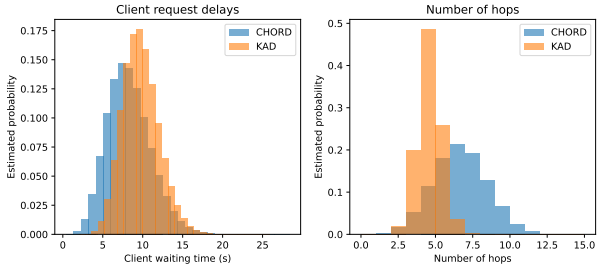

Figure 3. Distribution of waiting time and number of hops needed to fulfill client requests, 10000 nodes.

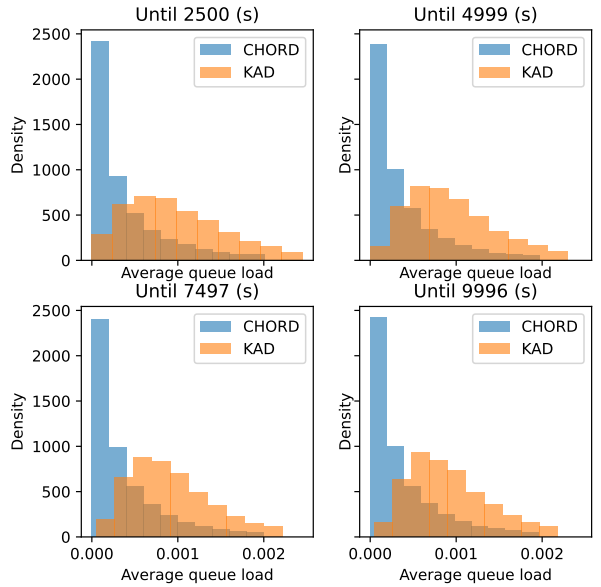

Figure 4. Distribution of average load over time, 10000 nodes.

procedure as messages flow through the network, thus no explicit updates are required.

V. Results

We start by looking at the delay experienced by the clients while waiting for a response.

From Figure 3 we can draw some initial considerations: in a stable network, Kademlia and Chord perform quite similarly in terms of delay, with Chord having a slight edge over Kademlia. However, Kademlia is able to fulfill client requests in less hops than Chord due to the fact that its nodes make α requests at a time and each knows up to $160 \cdot k$ other nodes.

One thing to note is that despite the lower number of hops needed by Kademlia to deliver the response, the mean waiting time is higher than Chord's. This is because each hop in the Kademlia protocol has to wait for all the α responses before proceeding to the next hop. This also explains the lower number of hops required, since each hop brings a lot of information: each of the α nodes we query responds with k nodes, while in Chord we query a single node which responds with information related to only one node.

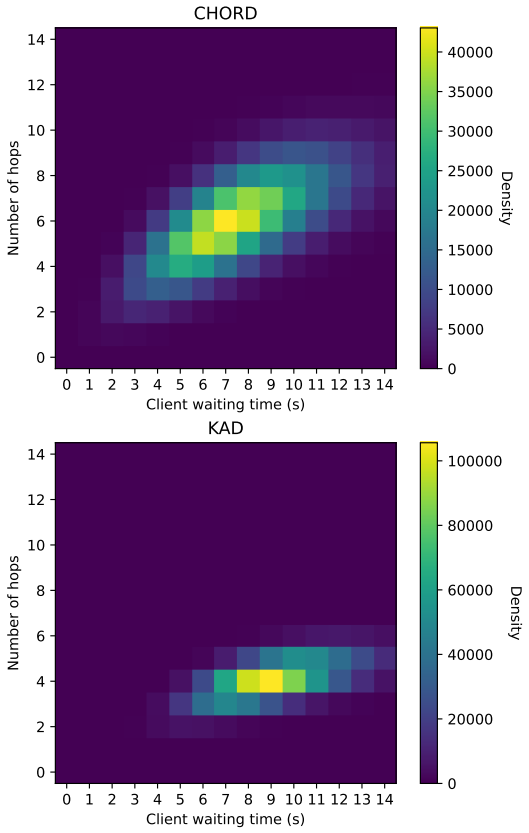

Figure 5. Correlation of message delay and number of hops, 10000 nodes.

From Figure 5 we can clearly see the correlation between the message delay and the number of hops needed by the network to fulfill the request. We can indeed spot a linear trend in the Chord chart, although in the Kademlia heatmap the observed trend is a bit more flattish due to the lower number of hops.

A. Queue load

As we can see from Figure 4, the average load of the queues of the nodes is quite different between Kademlia and Chord. As expected, Kademlia generates a lot more traffic and this is reflected in a larger resource usage by the nodes of the network. However, we would like to remark the fact that the traffic generated by Kademlia helps the nodes to keep their k-buckets up-to-date, while Chord does not have a way of doing this without explicit updates.

B. Impact of network congestion

As we can see from Figure 6, the number of clients coming in the system can greatly affect the performance of our protocols. In a stressful situation such as a client arrival rate of 66.7, Kademlia experiences a considerable performance degradation. Chord does not shine either, but keeps up fairly well with the load of requests. This is also due to the fact that Chord nodes' queues are less crowded since the network traffic in Chord is considerably smaller.

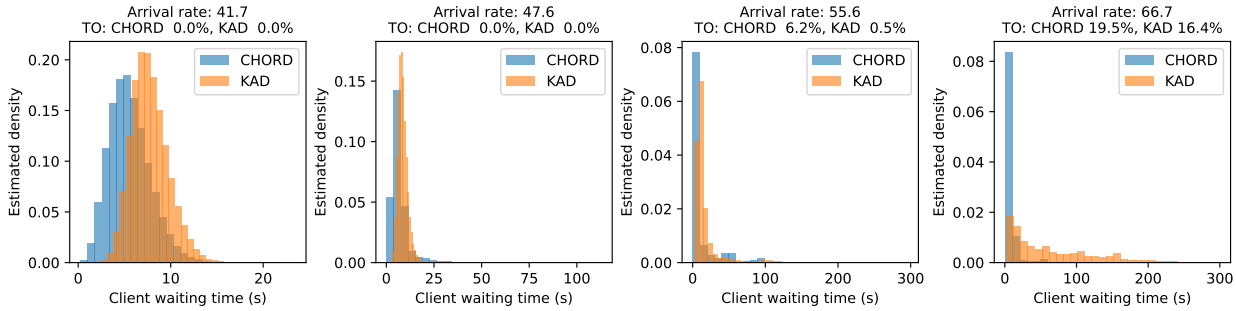

Figure 6. Distribution of waiting time and number of hops with different client arrival rates, 100 nodes. TO represents the number of timed out requests as a percentage of the total number of client requests for each DHT.

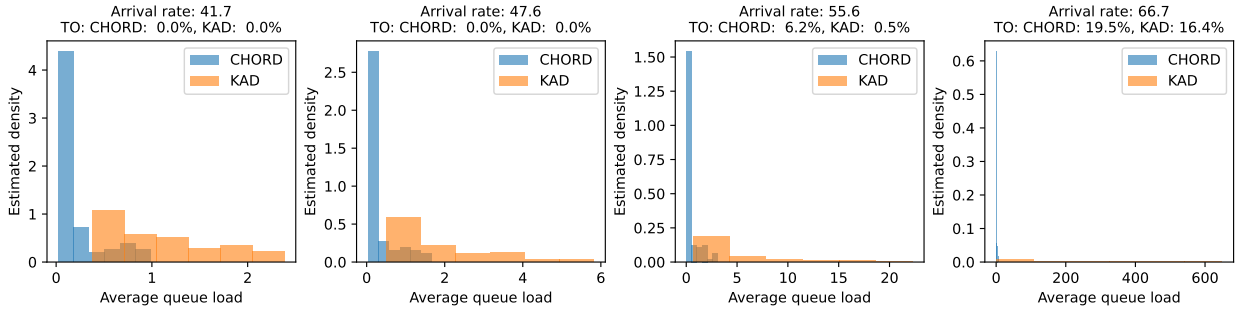

Figure 7. Distribution of average queue load with different client arrival rates, 100 nodes. TO represents the number of timed out requests as a percentage of the total number of client requests for each DHT.

Indeed, the same holds for the load of the nodes' queues: Figure 7 shows that Kademlia nodes suffer from the overwhelming load that is being generated by the network while Chord can still somewhat keep up. Although the timeout rate may seem lower for Kademlia, it gets worse in a non-linear way with respect to Chord's: at an arrival rate of 100, Chord experiences 70% timeout rate while Kademlia 98%.

VI. Conclusions

Comparing Kademlia and Chord on a completely fair ground is not straightforward: the protocols serve the same purpose but they do it by offering different trade-offs, at least in the versions we have implemented. Chord offers low network traffic overhead and low waiting delays at the price of robustness and redundancy, while Kademlia offers more robustness to failures and key redundancy at the price of higher resource demand.

Chord may figure as a clear winner in the department of latency and load, but it is important to keep in mind that our analysis assumes that no nodes join or leave during the simulation: if this was the case, every Chord node would have to periodically update its finger tables, which is a heavy task in terms of network load. Also, the more nodes are in the network the heavier the procedure is, as every single `FIND_NODE` becomes more expensive in terms of network load, since the number of hops grows logarithmically with respect to the number of nodes in the network. Furthermore, if every Chord node had more than one ID, the update procedure would have to be carried out for each and every ID.

In order to compare the protocols on a fair basis it would be interesting to implement multiple IDs and parallel queries in Chord. This would be a way to offer just about the same features of Kademlia while stressing the load of Chord nodes to an extent that would be similar to the load of Kademlia nodes.

In the future we would like to perform some analysis of the network load in case nodes join and leave during the experiment: Kademlia should fare fairly well with high churn given the fact that the k-buckets of each node update passively thanks to normal traffic, while Chord nodes would have to find a trade-off between finger table accuracy and update frequency. Indeed, if a Chord node does not update for a long time, its finger table will likely be outdated in the case of a very dynamic network, causing the overall performance of the system to degrade due to the larger number of hops needed to fulfill `FIND_NODE` requests.

References

[1] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International Workshop on Peer-to-Peer Systems*, pp. 53–65, Springer, 2002.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[3] "SimPy 4.0.1 - Discrete event simulation in Python." https://simpy.readthedocs.io/en/latest/index.html.

[4] "SimPy Shared Resources." https://simpy.readthedocs.io/en/latest/topical_guides/resources.html#res-type-resource.