Kademlia and Chord: DES and Performance Evaluation

Final report for the project of the Simulation and Performance Evaluation course @ University of Trento

Gabriele Masina
University of Trento, DISI
220496

Giovanni Zotta
University of Trento, DISI
223898

***Abstract*—In this project we implement a Discrete Event Simulation for evaluating the performance of the Chord and Kademlia DHT protocols. We collect and analyze multiple network metrics to compare the two protocols under different scenarios, observing how the network behaves when client requests arrive with an high rate and when nodes join and leave the DHT during the simulation. We provide a thorough performance evaluation that explores the drawbacks of the two protocols along with an interactive visualization of the key metrics. We show that it is important to analyze the whole context of the simulations, as one may draw wrong conclusions from a first-glance analysis.**

I. Introduction

The goal of this work is to implement the Kademlia [1] protocol and the Chord DHT [2] to simulate their behavior and conduct a performance evaluation analysis under different scenarios.

This is a continuation of the project that we carried out for the course on Distributed Systems 2, where we implemented the two DHTs in a Discrete Event Simulation environment. The goal of this project is to provide a thorough performance evaluation of the two protocols after enhancing the capabilities of Chord and adapting the simulator for dealing with nodes joining and leaving. We also provide a performance evaluation analysis on some of the most important metrics like client delay and network load.

In addition to the project for Distributed Systems 2, which code can be found at https://github.com/giovannizotta/kademlia/tree/v1.0.0, we did the following:

- **Refactor of the messaging framework**, in order to have more granularity and control on the simulation;
- **Multiple IDs for Chord nodes**: this also means parallel lookup queries and redundant storage;
- **Churn**: nodes are now able to join and crash during the simulation;
- **Chord updates**: Chord nodes periodically update their finger tables to deal with churn;
- **Better transmission delays**, which are now computed in a geographically consistent way;
- **Performance evaluation**: analyzed the behavior of the network under more scenarios in a more statistically meaningful way;
- **runexpy**: development of a Python package to easily run multiple simulations using different parameters combinations and retrieve data in a friendly format.

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 a rough 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 misbehavior of nodes.

The original Chord protocol does not provide data redundancy by default and each key lookup follows the finger table in an iterative manner; our simulator improves on this aspect by using multiple hash functions to give each node multiple IDs and by querying the routes to them in parallel. This brings 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, which makes our simulator a fair ground for comparing Chord with Kademlia in terms of feature set. This also has side-effects on the load experienced by Chord nodes, since they have to keep all their r finger tables updated.

III. Simulation framework

We implemented a network simulation framework based 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. However, 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 set of DHT nodes (Chord or Kademlia) and of clients arriving following a random process. 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 which handles the incoming message and start waiting for a response. The DHT node performs some actions to satisfy the request, possibly contacting other nodes in the DHT, and finally responds to the received request.

As in a realistic scenario, each DHT node has limited network buffer of size K and can serve only 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.

B. Message Layer

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

```
class Node:
    ...
```

The `recv_packet` method simulates the path that a network packet follows before reaching the application layer: we first check the current server status, meaning that if the network buffer is full we drop the packet, or we buffer it until it can be served.

Once the server is available, we wait for some random service time and call the `manage_packet` method, which is responsible for correctly handling the packet according to its type. In particular, the receipt of a `REPLY` triggers the response event that it stores, so that the process that was waiting for this response is awakened and can continue its procedure.

This method must be overridden by `Node`'s subclasses, to handle the other `REQUEST` packets in an appropriate way.

```
def recv_packet(
    self,
    packet: Packet,
) -> SimpyProcess[None]:
    # Drop packet if the node is crashed
    if self.crashed:
        return
    # or if the buffer is full
    if len(self.in_queue) == self.queue_capacity:
        return
    # Buffer the packet
    with self.in_queue.request() as res:
        # wait for it to be served
        yield res
        # wait a service time
        service_time = self.rbg.get_exponential(
                self.mean_service_time)
        yield self.env.timeout(service_time)
        # serve the packet
        self.manage_packet(packet)


@abstractmethod
def manage_packet(self, packet: Packet) -> None:
    msg = packet.message
    # replies trigger a response event
    if msg.ptype.is_reply():
        assert msg.event is not None
        msg.event.succeed(value=packet)
    # subclasses override this method
    # to handle other packet types
```

On the other hand, the sending of a message can be done with the `_send_msg` method, wrapped by `send_req` and `send_resp` which send a request and a response message respectively.

The main difference between them is that the former also creates a `Request` event that is stored inside the message and returned by the method, so that the sender can wait for it to be triggered once the corresponding reply is received.

```
def _transmit(
    self,
    dest: Node,
) -> SimpyProcess[None]:
    dist = distance(self.location, dest.location)
    # assume latency is 10ms every 1000km
    transmission_time = dist / 100
    transmission_delay = self.env.timeout(
        transmission_time)
    yield transmission_delay


def _send_msg(
    self,
    dest: Node,
    msg: Message,
) -> SimpyProcess[None]:
    packet = Packet(self, msg)
    yield from self._transmit(dest)
    yield self.env.process(dest.recv_packet(packet))


def send_req(
    self,
    dest: Node,
    msg: Message,
) -> Request:
    sent_req = self.new_req()
    msg.event = sent_req
    self.env.process(self._send_msg(dest, msg))
    return sent_req


def send_resp(
    self,
    dest: Node,
    msg: Message,
) -> None:
    self.env.process(self._send_msg(dest, msg))
```

The receipt of one or more responses can be waited for by calling the `wait_resps` method. This method fills the `packets` list with the packets received within some timeout, and raises a `DHTTimeoutError` if any of them times out.

```
def wait_resps(
    self,
```

```
    sent_reqs: Sequence[Request],
    packets: List[Packet],
) -> SimpyProcess[None]:
    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
    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()
    yield from self.wait_resps([sent_req], ans)
    return ans.pop()
```

C. An example

Let us consider the case when a Node wants to send a message to some other node; the message flow is shown by the `exchange` method. The sender starts the exchange by sending a packet, which after some time it is received by `other` and managed by its `manage_packet` method, while the sender waits for a response. Once `other` sends the response, the corresponding event is triggered after the transmission time and the original sender process can continue its `exchange` method, reading the content of the response.

```
from common.node import *

class EchoNode(Node):
    def manage_packet(
        self,
        packet: Packet,
    ) -> SimpyProcess[None]:
        super().manage_packet(packet)
        if packet.ptype == ECHO_MSG:
            self.echo_reply(packet)


    def echo_reply(
        self,
        packet: Packet,
    ) -> None:
        reply_msg = Message(
            ptype=MessageType.ECHO_REPLY,
            data=packet.msg.data.copy(),
            event=packet.msg.event)
        )
        self.send_resp(packet.sender, reply_msg)


    def echo_send(
        self,
        other: Node
    ) -> Request:
        # send empty packet
        msg = Message(ptype=ECHO_REQUEST,
            data={"m": "Hello"})
        return self.send_req(msg)


    def echo_back(
        self,
        packet: Packet
    ) -> None:
        message = packet.message.data["m"]
        self.log(f"Received answer {message}")

    def exchange(
        self,
        other: Node
    ) -> SimpyProcess[None]:
        """Simpy process that simulates the message
        exchange process"""
        # send req
        sent_req = yield from self.echo_send(other)
        # wait for response
        packet = yield from self.wait_resp(sent_req)
        # serve response
        yield from self.echo_back(packet)
```

IV. System setup

After building a framework that provides realistic network simulation, we needed to regulate the behavior of nodes in the network. Some areas needed particular attention as otherwise they would have skewed the simulation results, such as:

- **Transmission delay** – how long a message takes to transmit through the internet;
- **Client request distribution** – how popular each key is among clients;
- **Client arrival rate** – how often clients make requests and what distribution they follow;
- **Churn** – how often nodes join and leave the DHT.

A. Transmission delay

In our previous work we just picked a network delay from an exponential distribution, which does not take into account the actual geographical position of the nodes. We improve on this aspect by fetching a list of nodes of a popular peer-to-peer network (Bitcoin) and by assigning each of our nodes geographical coordinates picked from that list[1].

Every time a node has to send a message to someone else, we compute the distance between the two nodes and estimate the latency that would appear in the real world. As an heuristic, we decided to estimate the latency based on the distance by simply assuming that **a message takes 10ms to travel 1000km**.

B. Client request distribution

During the simulation, it is not always reasonable to assume each key is equally popular among clients. In the observations that have been carried out on applications that exploit DHTs such as file sharing [4], request distribution has not been uniform but instead followed a Zipf's law, which is one of a family of related discrete power law probability distributions[2].

This means that a handful of keys are likely to be extremely popular among clients, while the vast majority of them belongs to some user niche. Modern DHTs make use of load balancing

[1] supplied by https://bitnodes.io
[2] https://en.wikipedia.org/wiki/Zipf%27s_law

algorithms to mitigate this issue, while our DHTs do not have load balancing capabilities outside of those provided by their hash functions, which means that each key is stored on a fixed number of nodes regardless of its popularity.

After briefly evaluating the results of applying a Zipfian distribution to our system, we decided that it was not fit for our analysis, as the load was too concentrated on a handful of nodes and this would have made the analysis pretty dull. Instead, we decided to **assume that the client requests were being load balanced to a uniform distribution**, which made our analysis more interesting and independent from the real client request distribution.

C. Client arrival rate

The **rate** at which clients make requests is also important: we model it as an **exponential distribution** and simulate various combinations.

D. Churn

Churn is the measure of the change in the set of participating nodes due to joins and failures. In our system, nodes join the network and then crash after a set amount of time. We manipulate churn by acting on the knobs of the rate of nodes joining and crashing:

1) Join: The rate at which nodes join the network is modeled via a **second-order hyper-exponential distribution**, which has been observed by the authors of [4]. In our system, the `joinrate` parameter is used to manipulate this distribution to make nodes join more or less frequently. The `joinrate` is the factor by which the *lambda* parameters of both the exponential distributions are multiplied, which means that it is a division factor for the mean time between two nodes joining the DHT.

2) Crash: The parameter `crashrate` of our system is not really a rate, instead it is a measure of how long each node stays up. This is modeled via a **Lognormal distribution**, which has once again been observed by the authors of [4]. We divide the mean of the Lognormal distribution by the value of `crashrate`. Thus, it is a division factor for the mean lifetime of a node.

We adopted this configuration because it allowed us, if we assume no timeouts, to have a stable number of nodes after a period of warmup. This is because the `crashrate` parameter defines the distribution of **session duration for one node**, while `joinrate` models the distribution of **delay between two nodes joining**.

E. Default configuration

The main metrics that we wanted to analyze were the **latency** experienced by the clients and the **load** of the servers participating in the DHTs. **Accuracy** of DHT lookups also plays an important role in the clients' experience, so we observe the rate at which clients get the correct response when they ask for a key. Many variables play a role in such metrics and we needed a way to analyze their effect thoroughly. We were also interested in offering a view of how many nodes were active in the network at any given time in order to understand how the two DHTs can keep up with the churn.

Making a plot for every possible combination of the parameters would have been impractical and thus we built an interactive visualization tool using the Streamlit[3] framework to help us plot the results. We made a time slider to cut out part of the simulations that are being analyzed, allowing any user visualizing the results to zoom in on a specific timeframe. This is especially useful if one wants to observe the state of the network during the warmup phase, or for cutting out the warmup phase.

Table I summarizes the network variables and their default values, while Tables II and III contain the configuration parameters for Chord and Kademlia respectively. These values have been tuned in order to make the network as stable as possible – meaning that the number of nodes stabilizes to a constant value after some warm-up time – and to not overload the network buffers of the nodes.

Once we fixed the simulation constants, we decided to analyze the influence of the following variables:

- Client arrival rate
- Join rate
- Crash rate

The set of parameters that we chose to analyze is displayed in Table IV in the Appendix.

Furthermore, in order to have more confidence about the results of our simulations, we performed a number of runs for each configuration of the system. We started with the idea of having 30 runs for each configuration with different random seeds, but we quickly figured out that our simulations were quite cumbersome in terms of resource usage. Considering the large amount of data that we had at hand, we decided to run 10 runs for each configuration, allowing us to have a better experience analyzing the final result. This still resulted in an overall amount of data of 174 GB, with some plots taking up to 120 GB of RAM to compute.

In order to manage this large number of simulations with different parameters, we also implemented a Python package named *runexpy*[4]. It is highly inspired by the ns-3 Simulation Execution Manager [5], and allows us to run simulations with different seeds and parameter configurations and later retrieve the data by means of a handy interface.

V. Results

Before getting to the analysis of the results, we explain the kind of plots that we made:

- **Latency ECDF**: for each DHT, we plot an Empirical Cumulative Distribution Function of the latency experienced by the clients with 95% confidence intervals. This allows the reader to understand which DHT has a faster response time;
- **Latency over time**: for each DHT, we plot the average latency experienced by clients over time;

[3] https://streamlit.io/

[4] https://pypi.org/project/runexpy/

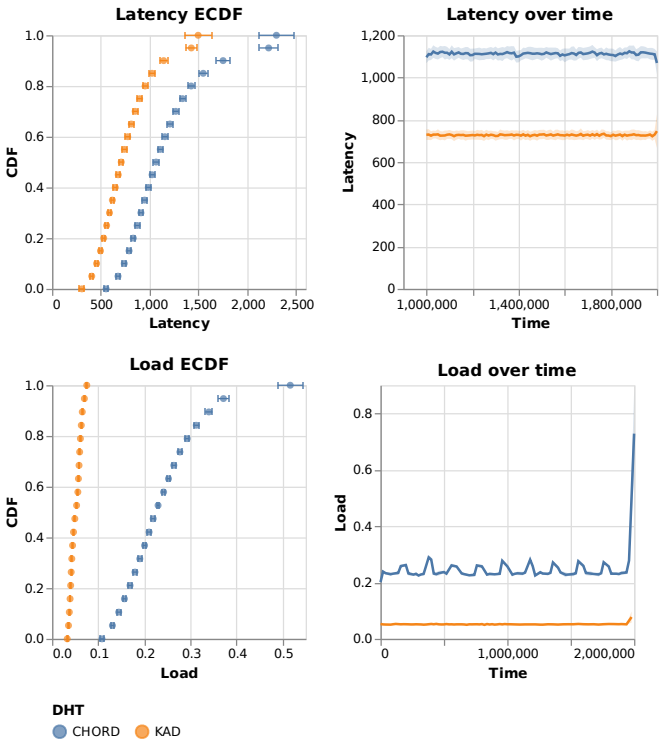

Figure 1: Low-stress scenario, no-churn with 0.1 client rate. Plots show 95% confidence intervals.

- **Load ECDF**: similar to latency ECDF, but for load. We define the load of a node as the number of packets that are in its network queue. A detail is particularly significant: we need to take time into account. For example, if a node stays idle for a period of time of 800 and then has a load of 1 for 200 units of time, its average load is 0.2 and not 0.5. Therefore, we compute an average of the load weighted by the period of time where a certain load was sustained;
- **Load over time**: for each DHT, we plot the average load experienced by nodes over time.

A. Stable network, no churn

The first thing that we wanted to analyze was the behavior of our DHTs when the network is stable and without nodes joining and leaving. Starting with a low-stress scenario in Figure 1, we see what we expected: Kademlia outperforms Chord both in terms of latency and load. In the load over time chart, it is interesting to note that Chord has some load spikes every about 200000 time units: those are caused by the periodic updates of the finger tables.

B. Introducing churn

Next, we wanted to see how the system behaved in presence of churn. We were surprised to see the results of Figure 2, as they did not match our expectations. After a warmup time (which ends at about time 1000000), **Chord outperforms Kademlia in terms of client latency, which is weird** given how the protocols work. In this case, since there is churn, we also plotted a chart showing the number of active nodes, which are nodes that did not crash yet and are able to respond to queries. To have a better view of the data, we implemented a slider in Streamlit that allows us to select a specific time window so that we can analyze the statistics cutting off the warmup period. If we zoom in on the time interval 0-1000000, we notice a trend similar to the no-churn scenario of Figure 1. However, as soon as we look at the period where nodes start crashing, the behavior of the system gets weird and incomprehensible to us. Zooming on the period from 1000000 to 2000000 as in Figure 3, we can see that the number of **Chord** nodes **is slowly collapsing, but that seems to not affect the latency** of clients. That is definitely weird, since the

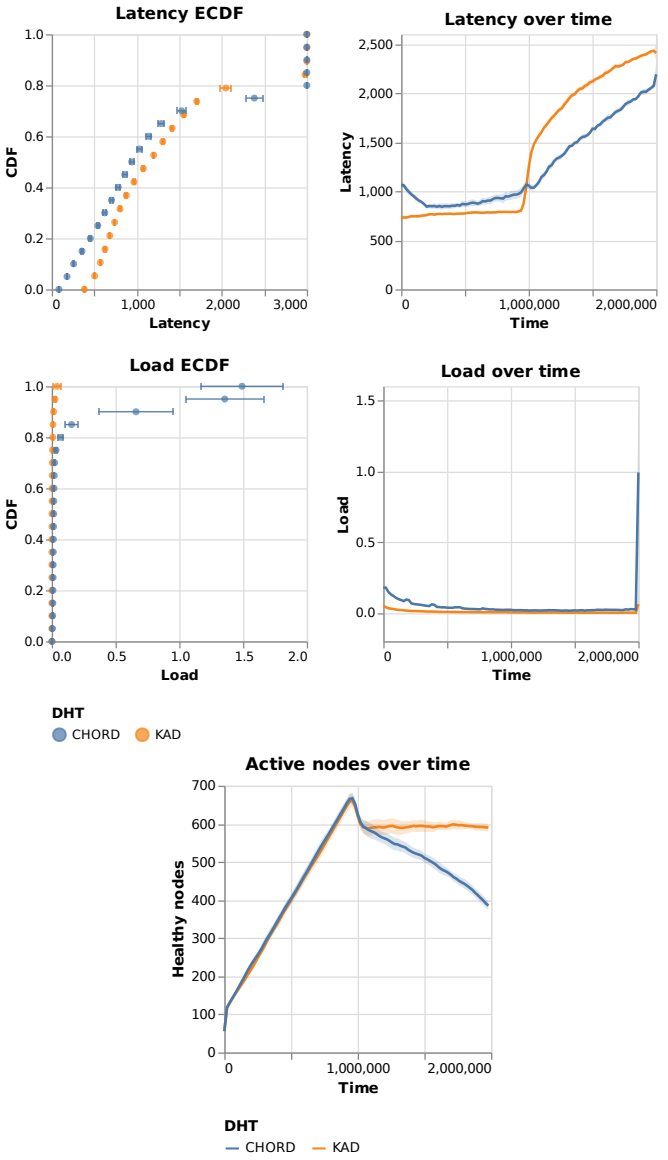

Figure 2: Moderate-churn scenario (client rate 0.1, join rate 1, crash rate 0.1), looking at time interval 0 – 2000000.

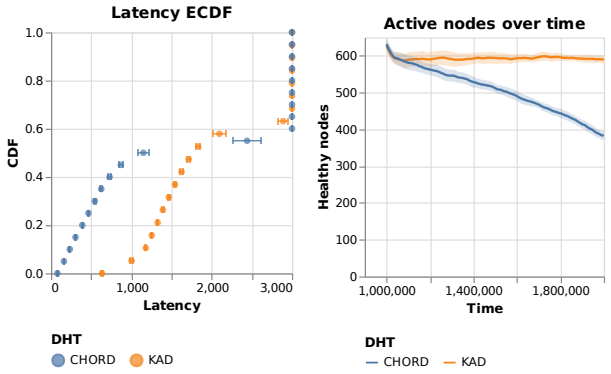

Figure 3: Moderate-churn scenario (client rate 0.1, join rate 1, crash rate 0.1), looking at time interval 1000000 – 2000000.

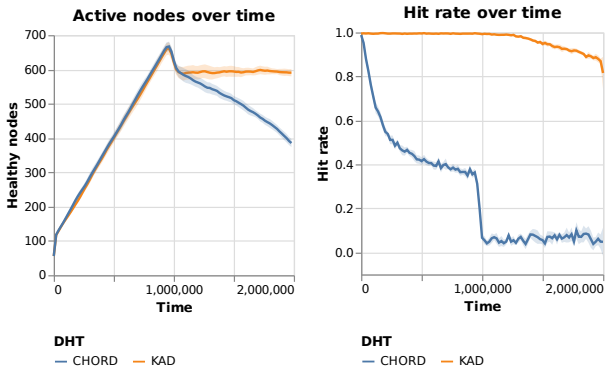

Figure 4: Moderate-churn scenario (client rate 0.1, join rate 1, crash rate 0.1)

collapse of the number of nodes can only happen if the nodes are not able to join, which means their `find_node` requests are timing out. We were seeing timeouts both for Chord and Kademlia (all requests with 3000 latency are timeouts, and in Figure 3 we can see that 40% of the requests are timing out for both), but Kademlia nodes were able to join while Chord nodes were not.

We tried to reason about what was going on and, after extensive testing we speculated that it was a problem with the protocols and not with the implementations. Specifically, we thought that Kademlia was behaving as expected but that Chord was faulty. Our intuition was that the Chord ring topology was getting destroyed due to the high churn and that there was no ring anymore, just a forest of small chains. In such a scenario, **a client asking for a key on such a DHT would not find the correct keys, but the reply would come fast** because the several communities were composed of a small number of nodes.

To test our belief, we decided to collect a new metric that we will call "hit rate". **A request is considered a *hit* when it returns the last value that was set for the corresponding key**. We define `hit_rate` as the number of hits over the total number of requests. As we can see from Figure 4, Chord's hit rate has been severely impacted by the churn and this explains the weird behavior that we were observing. Everything seemed to be going great for Chord in terms of latency and load, but as soon as one digs deeper the shortcomings get unveiled. From this figure we can see that Chord becomes basically useless since its hit rate is close to zero, while Kademlia is able to sustain the churn quite well and does not flinch as hard even if a hundred nodes crash almost at the same time.

C. High client rate

In order to see the effect of a larger node population in the system, we plot a low-churn scenario with a high client load in Figure 5, which is a scenario that starts with 100 nodes which grow over time. It is interesting to see that, for Kademlia, as the number of active nodes over time increase, all its metrics improve. On the other hand, Chord is not able to keep up with the load as its hit rate is close to zero.

We also wanted to see how the two DHTs behaved in a no-churn scenario under different client arrival rates. Figure 6 shows that, as long as the client arrival rate is sustainable (0.1 and 0.2), Chord and Kademlia are able to achieve a perfect hit rate with Kademlia being slightly better in terms of load. However, as we increase the number of clients arriving, we can see that Chord's hit rate drops dramatically, while Kademlia's load gets higher. This is because Chord, even if there is no churn, needs to periodically update its finger tables and stabilize its successors, and considers itself out of the ring if it does not manage to do so within a timeout.

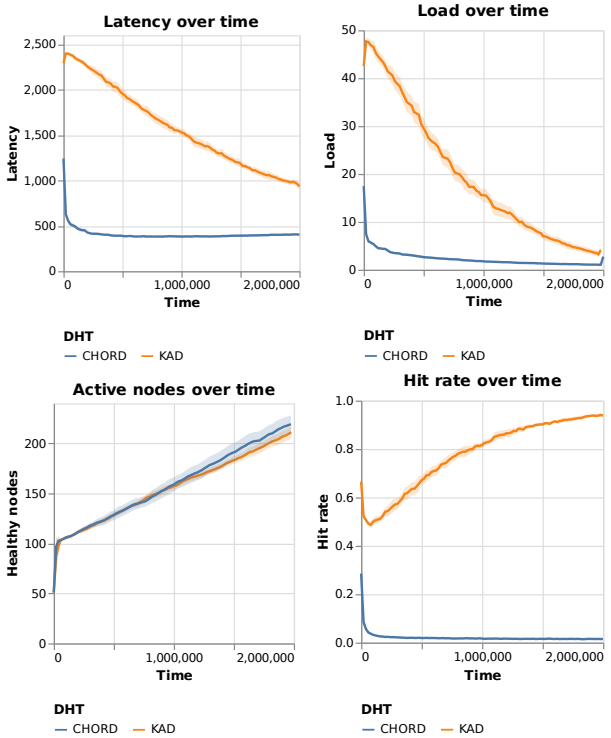

Figure 5: Low-churn scenario, high client rate (client rate 1, join rate 0.1, crash rate 0.01)

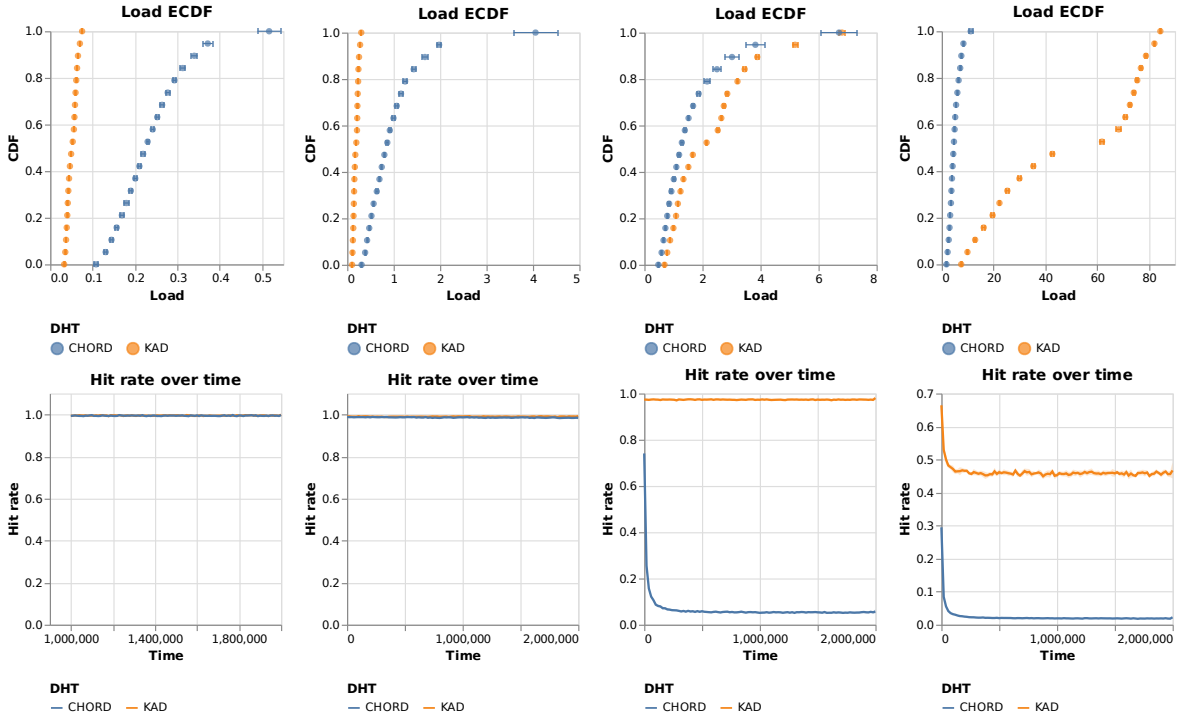

Figure 6: No-churn scenario, the four columns show client rates 0.1, 0.2, 0.5, 1.0 respectively

D. Stress test

As a stress-test, we tried to introduce an unrealistically high churn to see how the two DHTs would behave. Figure 7 shows that the churn was unsustainable for both, as the number of client timeouts reached 90%.

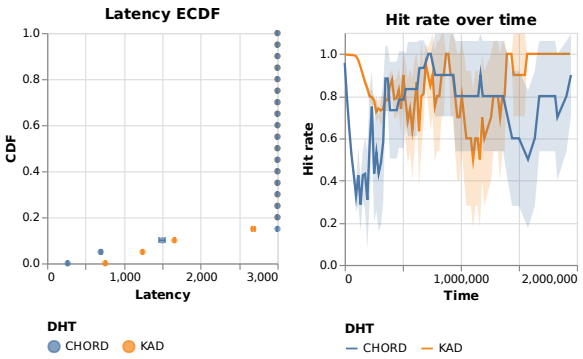

Figure 7: Extreme-churn scenario (client rate 0.1, join rate 10, crash rate 1)

VI. Conclusions

At first, we were surprised to see Chord performing better than Kademlia in the scenario of Figure 2, wondering what was going on. The main takeaway that we had from this project is that often it is not enough to look at a few metrics to draw conclusions, but that it is a good idea to dig deeper in trying to understand the whole context. Indeed, thanks to the hit rate metric, we were able to assess that Chord, despite seemingly being much better than Kademlia, was totally unusable due to the topology destruction that was induced by churn. This drawback could be partially addressed by significant changes in the protocol that would make it more resilient to failures, but we would like to stress that Kademlia main's advantage is that it keeps learning the topology from the messages that come in, while Chord cannot do that. Indeed, our Chord implementation needed particular attention when dealing with node failures to make sure it was still functioning, while we barely touched Kademlia.

At a first glance Kademlia seems to be more complex than Chord, but after implementing both protocols it is apparent that Chord in reality needs much more attention and complexity to manage its topology, while a basic Kademlia implementation is able to obtain respectable results.

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] D. Erman, D. Ilie, and A. Popescu, "Bittorrent traffic characteristics," in *2006 International Multi-Conference on Computing in the Global Information Technology - (ICCGI'06)*, pp. 42–42, 2006.

[5] D. Magrin, D. Zhou, and M. Zorzi, "A simulation execution manager for ns-3: Encouraging reproducibility and simplifying statistical analysis of ns-3 simulations," 2019.

Appendix

Parameter	Default value	Description
NODES	100	Initial number of nodes
DEFAULT_PEER_TIMEOUT	500	Maximum time a node waits before freeing the resources for a sent request
DEFAULT_CLIENT_TIMEOUT	3000	Maximum time a clients waits before freeing the resources for a sent request
DEFAULT_NUMBER_OF_SERVERS	1	Number of packets a server can process simultaneously.
DEFAULT_QUEUE_CAPACITY	100	Size of the network buffer
DEFAULT_MEAN_SERVICE_TIME	5	The service time follows an exponential distribution with the value of this parameter as mean.
Join distribution parameters		The inter-arrival time of new nodes joining the network follows an hyper-exponential distribution with the following parameters
• DEFAULT_JOINLAMBDA1	10	
• DEFAULT_JOINLAMBDA2	5	
• DEFAULT_JOIN_P	0.3	
Crash distribution parameters		The lifetime of a node in the network follows a log-normal distribution with the following parameters
• DEFAULT_CRASHMEAN	10	
• DEFAULT_CRASHVARIANCE	5	
DEFAULT_N_KEYS	1000	Number of keys stored and requested in the hash table.

Table I: Simulation constants

Parameter	Default value	Description
DEFAULT_K	5	Number of hash functions used.
Update protocol parameters		Chord nodes wait a time between messages to update the finger table that follows a normal distribution, with the following parameters
• DEFAULT_UPDATE_PERIOD_MEAN	200000	
• DEFAULT_UPDATE_PERIOD_STD	5000	
Stabilization protocol parameters		Chord nodes wait a time between messages to ping their successor and predecessor that follows a normal distribution, with the following parameters
• DEFAULT_STABILIZE_PERIOD_MEAN	5000	
• DEFAULT_STABILIZE_PERIOD_STD	100	

Table II: Chord parameters

Parameter	Default value	Description
DEFAULT_ALPHA	3	Number of parallel requests
DEFAULT_K	5	Bucket size.

Table III: Kademlia parameters

Parameter	Values	Description
CLIENT_RATE	$0.1, 0.2, 0.5, 1$	Rate parameter of the exponentIal distribution followed by client requests
JOIN_RATE	$0, 0.1, 1, 5, 10$	Division factor for the mean of the join distribution
CRASH_RATE	$0, 0.01, 0.1, 0.5, 1$	Division factor for the mean of the crash distribution

Table IV: Simulation parameters