Project Report

# Analysis of Overheads in Chord as a Distributed Hash Table

December 2023

## 1 Introduction

Distributed Hash Tables (DHTs) are a type of Key-Value store (KV store) where the key space is divided between the nodes in the network, and each node is only responsible for its own region. The distributed nature of DHTs allows for large scaling and high availability.

Chord [3, 4] is a protocol for implementing a peer-to-peer DHT. Chord uses a m-bit ID space where m is the number of bits in the hash digest of a given key. A typical Chord implementation uses the SHA-1 hashing algorithm with a 160-bit ID space. Each Chord node is only responsible for the portion of the ID space between itself and the node immediately before it, i.e., its *predecessor*. The next node immediately after a node is called its *successor*. Each Chord node stores an m-bit routing table, called a *fingertable*, where each entry $i$ (*finger*[i]) has the node which is the successor of the ID $= (n + 2^{i-1}) \bmod 2^m$, meaning it is at least $2^{i-1}$ further away from the current node $n$ on the Chord *ring*. In a stable ring with $N$ nodes, insertion and lookups require $O(log(N))$ messages while joining and leaving the network can be done in $O(log^2(N))$.

However, the evaluation sections of Chord papers did not discuss two cases:

1. Hashing overheads were not considered, and it was not mentioned how different hashing algorithms could potentially change the performance of Chord.
2. The effects of network a single network topology was mentioned in [4]. However, different topologies could affect the performance of Chord outside of the theoretical bounds discussed in the papers.

This project builds a Chord implementation in Erlang to investigate the effects of hashing and network topology on Chord's performance. This investigation provides insight into DHTs and peer-to-peer applications and factors other than message-passing overheads that affect their performance.

### 1.1 Changes from Original Proposal

The object of the original proposal was to build either a Distributed Key-Value Store or a Distributed Hash Table. Chord functions as both and thus is a good fit.

The original proposal underestimated the scope of the project, specifically building a network simulator with routers in between nodes. Due to time limitations, this is left as future work.

The discussion during office hours concluded with interfacing between Erlang and C++ to pinpoint the performance bottleneck between hashing and message-passing. However, Erlang has a crypto module [1] that uses NIFs to use hash functions provided by OpenSSL, and thus this module was used in lieu of a C++ implementation.

## 2 Design & Implementation

The code for the project can be found on GitHub at `https://github.com/sepehr541/chord`

The project was implemented in Erlang with three main pieces:

1. Custom structures to model the state of a node and messages using Erlang's Records[2] and Types[3].
2. The Chord API to interact with a given node
3. Implementing the `gen_server`[4] behaviour for message handling and RPC calls to other nodes

[1]`https://www.erlang.org/doc/man/crypto_app`
[2]`https://www.erlang.org/doc/reference_manual/records`
[3]`https://www.erlang.org/doc/reference_manual/typespec.html`
[4]`https://www.erlang.org/doc/man/gen_server`

## 2.1 Data Structures

The state of each node, node references, and all other structures were defined as Erlang Records, which allow for creating a tuple with a certain "type" and naming the tuple's fields. Moreover, it allows for accessing the fields directly via their name or pattern-matching and also easy copying of state where a new record can be instantiated from an existing record where only the specified fields were changed.

Erlang also supports types that can be used to specify the type of fields in records and specify function interfaces. Furthermore, primitive datatypes can be aliased to be used as abstract types in the application without revealing the underlying representation; for example, `-type id() :: binary().` assigns the type of `id()` to be the type `binary()`.

Dialyzer[5] is a type checker for Erlang that, using the specified types, will report any definite inconsistencies (conservative type checking). Throughout the implementation process, Dialyzer proved to be a critical tool in detecting bugs early. However, since it is a conservative type checker, it does not consider potential issues, and subtle bugs did bypass its checks.

Using records allowed for convenient pattern-matching on function parameters and simplified the code significantly. However, `case` expressions were used instead when appropriate to simplify code and prevent repetition on functions with a larger variety of parameters, e.g., the `handle_call` function, which pattern-matches on the request received from the caller.

## 2.2 Chord API

While Chord does not enforce an API, it defines a number of functions in pseudo-code to implement the protocol. In this project, the same API as the pseudo-code was used as the basis for the implementation, and other APIs were added to provide the full functionality. Notably, the `putEntry` and `getEntry` functions were defined to put values into and get values from the nodes. Originally, Chord delegates this functionality to a higher-level application built on top of it, and Chord is only used to locate nodes according to IDs. The complete list of these functions can be found in [3, 4], and thus is not included here.

## 2.3 Erlang's `gen_server`

Erlang provides a set of behaviors that are frequent patterns, similar to design patterns, implemented as a module. The `gen_server` is a behavior that enables a process to receive RPC calls from other processes while hiding the underlying message passing and wait-for-response mechanisms. To implement a behavior, a module is annotated with a `-behavior(gen_server).` attribute and the required callbacks for the behavior are then implemented and exported from that module.

The following `gen_server` behavior callbacks are essential to the project:

- `init` which is called when a new `gen_server` process is spawned
- `handle_call` which is called when an RPC call is received from another process. After processing the call, a reply is sent to the caller, and a state is returned locally to reflect changes to the callee's state.
- `handle_cast` which is called when a cast is received from a caller. Cast requests are one-way only and do not allow for a reply. Furthermore, they are non-blocking on the caller side.

The `gen_server` behavior requires other callbacks to be implemented as well, but in this project, their implementation is simply a stub with no functionality.

A `gen_server` processed is spawned using `start_link`[6] where a symbolic name is registered with the `gen_server` used in place of a Pid to interact with the process. A call to a `gen_server` is made via `gen_server:call` [7] where the server is called to perform a given request. When such a call is made, the

[5]`https://www.erlang.org/doc/man/dialyzer.html`
[6]`https://www.erlang.org/doc/man/gen_server#start_link-3`
[7]`https://www.erlang.org/doc/man/gen_server#call-2`

`handle_call` callback is called on the callee's side with a request, a reference to the caller, and the current state of the `gen_server`.

### 2.4 Limitations

The current implementation has a number of limitations.

- The current prototype does not support nodes leaving the ring, while Chord allows this operation.
- There is a subtle bug where the initial caller is called by some other node when a request is forwarded to other nodes. This results in a deadlock where both nodes block indefinitely, waiting on the other node. This limits the number of nodes that can currently be instantiated and used for evaluation.
- The current prototype uses `cast` to notify other nodes to update their fingertable. This assumes that all the notify messages get delivered, and all the servers will correctly set their fingertables.
- The correct prototype does not support concurrent join operations. Chord allows for multiple nodes to join the ring concurrently and other nodes will fix their fingertables over time using timeouts.

## 3 Evaluation

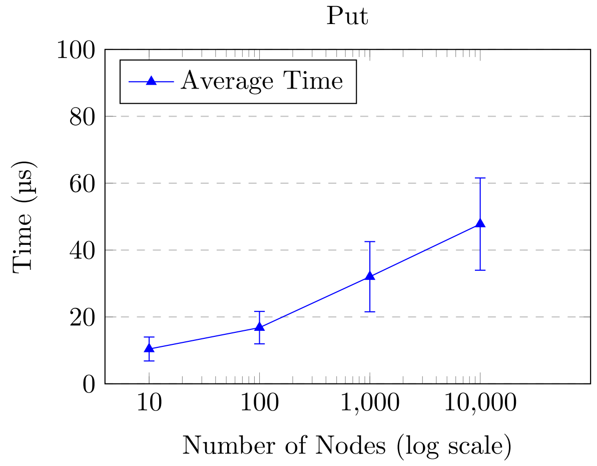
The current prototype is evaluated in its performance with various number of nodes and put/get operations. Furthermore, the overheads of hashing and message passing are compared with various key sizes, which greatly affect the overhead of the hashing.

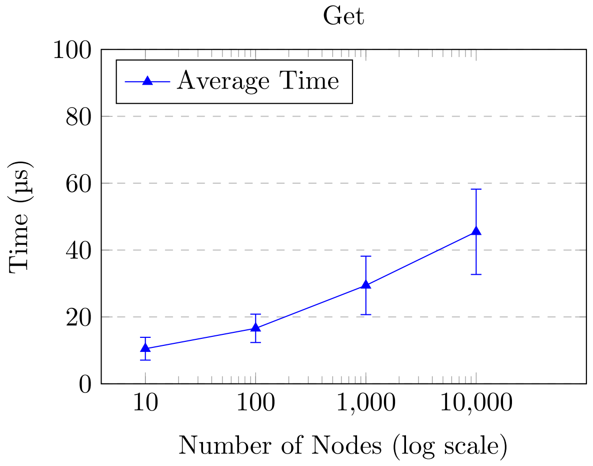
### 3.1 Setup

The evaluation was done on a machine with an AMD 7840U CPU with 8 cores/16 threads and 32 GB of memory. The machine is running Fedora 39 with Linux kernel version 6.6.6. The system is running Erlang/OTP 26 [erts-14.1.1]. The measurement and analysis were done using Python and Erlang scripts.

### 3.2 Key-Value Operations

The latency for Put and Get operations with varying numbers of nodes over 100 requests. The benchmark initially performs 100 Put operations one at a time, followed by 100 Get operations. For each operation, a random node is chosen to perform the operation.

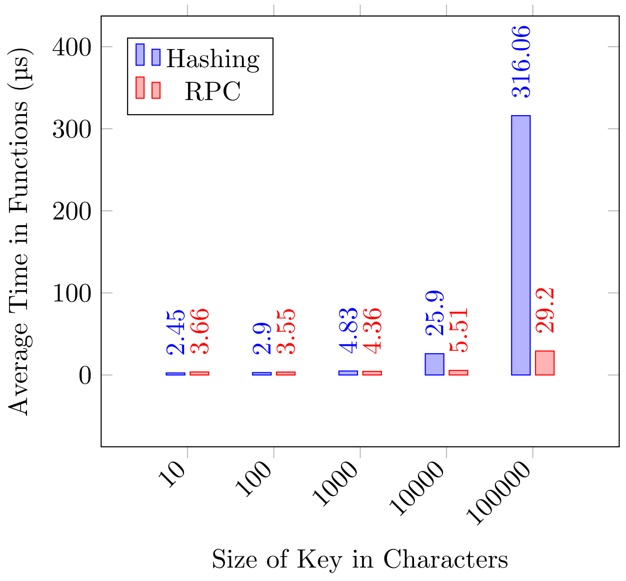


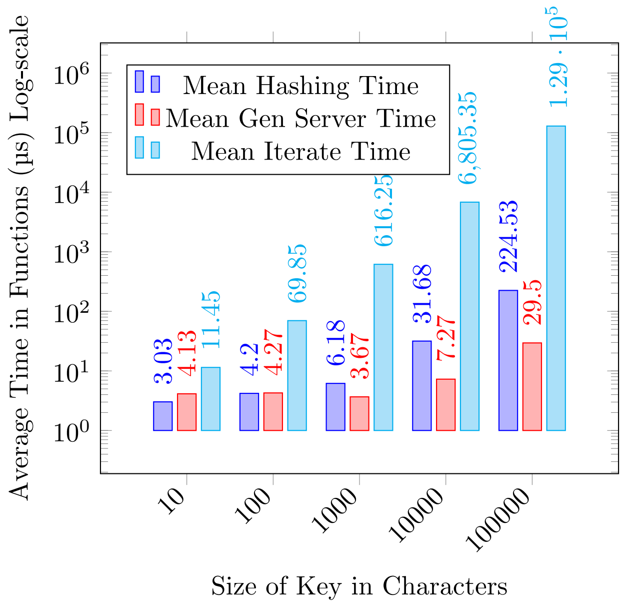


The plots show a nearly identical pattern for Put and Get, where the latency is increased with the number of nodes, as claimed in the Chord papers. More nodes result in more messages passed between the nodes, and each node is responsible for a smaller region of the ID space.

## 3.3 Comparing Overheads of Hashing and RPC

The time spent in each "category" of functions, i.e., hashing and RPC, was measured using Erlang's profiling module `eprof`[8]. Each abstract operation has many functions involved, and thus, the times in each of the related functions were summed up for a given process. Then, the measured times were averaged across the processes. The following plot compares the overheads of each category, measured for 10 nodes, with 100 sequential Put operations over varying key sizes.





The overhead of hashing is less than RPC for small keys while it surpasses RPC significantly for larger keys. This insight is critical in measuring application performance and addressing the correct bottleneck considering the application properties such as key size.

[8]`https://www.erlang.org/doc/man/eprof`

# 4 Reflection & Lessons Learned

This project has been most insightful in the semantics of Erlang, RPC, profiling performance measurement, and software engineering. The following are a few of the prominent learning points from the project.

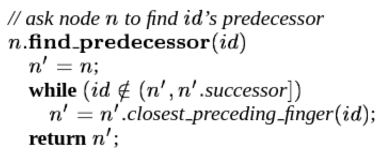
## 4.1 Theoretical Bounds vs. Real Measurements

One of the major lessons from this project was to distinguish between theoretical bounds and real measurements, especially the implicit factors that can significantly affect the performance of the system. It is critical to consider a broader range of factors alongside theoretical bounds to best decide whether a system has the performance characteristics required for a given application. The concrete example in this project is the effect of the key size on the hashing overhead, which for larger keys significantly surpassed the RPC overhead, revealing that the performance bottleneck of larger keys is in fact, hashing and not message latency. On the other hand, smaller keys did incur more messaging overhead than hashing.

Simply put, **implementation details matter**.

## 4.2 Pseudo-code is Not Code

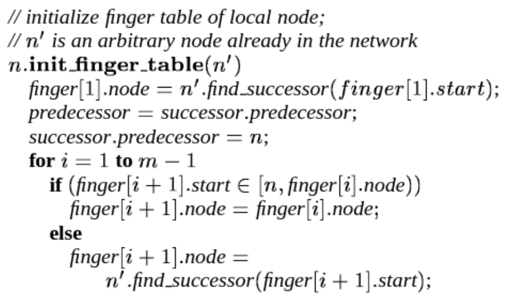
The Chord papers provide pseudo-codes throughout the papers. This project was implementation according to the pseudo-code provided; however, this approach proved to be insufficient to get a correct implementation. The following snippets from the papers demonstrate the subtle semantics that were not captured in the pseudo-code but proved to be critical to the implementation.



```
// ask node $n$ to find $id$'s predecessor
$n$.**find_predecessor**($id$)
  $n' = n$;
  while ($id \notin (n', n'.successor]$)
    $n' = n'.closest\_preceding\_finger(id)$;
  return $n'$;
```

Figure 1: Pseudo-code for `find_predecessor`

The pseudo-code for `find_predecessor` seems fine at first glance; however, it loops forever if there is only one node in the ring. With a single node $n$, $n.successor = n$. This results in $id$ never fitting in the interval $(n, n]$ which is empty[9]. Since there is a single node in the ring, `n.closest_preceeding_finger(id) = n`, resulting in an infinite loop. Furthermore, if $n' = n$, then by the convention established in the Chord paper, RPC calls are prefixed with the node name, this call would be an RPC to itself. This could cause a deadlock; however, Erlang's `gen_server` does not allow for self-calls and conveniently throws a `calling_self`[10] error and terminates the process.

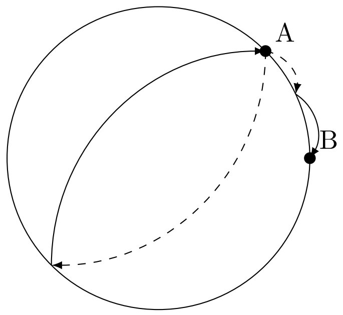


```
// initialize finger table of local node;
// $n'$ is an arbitrary node already in the network
$n$.**init_finger_table**($n'$)
  $finger[1].node = n'.find\_successor(finger[1].start)$;
  $predecessor = successor.predecessor$;
  $successor.predecessor = n$;
  for $i = 1$ to $m - 1$
    if ($finger[i+1].start \in [n, finger[i].node)$)
      $finger[i+1].node = finger[i].node$;
    else
      $finger[i+1].node =$
          $n'.find\_successor(finger[i+1].start)$;
```

Figure 2: Pseudo-code for `find_predecessor`

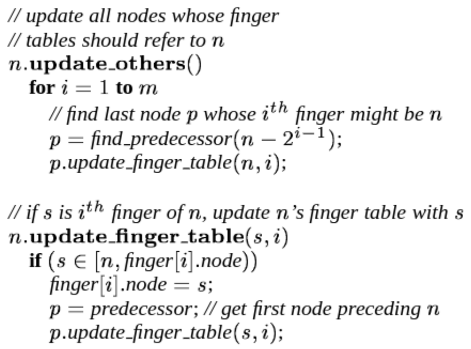
[9]`https://en.wikipedia.org/wiki/Interval_(mathematics)#Including_or_excluding_endpoints`
[10]`https://www.erlang.org/doc/man/gen_server#call-3`

The pseudo-code for `init_finger_table` is further missing a case. This function is called when a node first joins the ring, and thus no other node is aware that this node exists. In the last statement in `else`, the *finger*$[i+1]$ is assigned to the successor found by node $n'$; however, the node must consider itself as a valid successor for that position as well, which is missing from the pseudo-code. The following figure illustrates this case.



The dashed arrows indicate the starting id for *finger*[i], while the solid arrows point to the succeeding node of that id. As it can be seen, if the two nodes are close on the ring, after a certain range, *finger*[i] should point to the node itself, which is the case missing from the pseudo-code.

It is important to set up the finger table correctly, as the majority of the functions rely on the finger tables. This could be mitigated if the requests are simply forwarded to the successor of the node. The Chord papers prove the correctness of their protocol using this approach. However, in a real implementation, finger tables are used, and thus, the pseudo-code is incomplete.



```
// update all nodes whose finger
// tables should refer to $n$
$n$.update_others()
  for $i = 1$ to $m$
    // find last node $p$ whose $i^{th}$ finger might be $n$
    $p$ = find_predecessor($n - 2^{i-1}$);
    $p$.update_finger_table($n, i$);

// if $s$ is $i^{th}$ finger of $n$, update $n$'s finger table with $s$
$n$.update_finger_table($s, i$)
  if ($s \in [n$, finger[$i$].node))
    finger[$i$].node = $s$;
    $p$ = predecessor; // get first node preceding $n$
    $p$.update_finger_table($s, i$);
```

Figure 3: Pseudo-code for `find_predecessor`

The pseudo-code for `update_others` and `update_finger_table` has a couple of subtle points to look out for. First, in `update_others`, the loop needs to consider the case where $p = n$ and avoid updating itself. If not considered, it causes an RPC call to self. Second, the condition to check if the node passed in, $s$, is between the current node $n$ and its current *finger*[i] is missing the case when there is a single node in the ring. In this case, on a second node joining, the case for $s \in [n, n)$ will always be false as the interval is empty. Furthermore, the call to `update_finger_table` can cause an RPC self-call where the initial node is called again when the request is propagated around the ring.

Throughout the paper, intervals are used extensively; however, no examples for wrapping around the ring are provided, and it is not explicitly mentioned that the checks for intervals need to consider this case. Furthermore, interval checks could be replaced with comparing distances between points.

## 4.3 Erlang's `gen_server`

Initially, the implementation of the prototype used Erlang's explicit message passing. However, the Chord papers implicitly assume RPC semantics. This was not obvious from the start, and thus there was a refactor from message passing to using `gen_server` for RPC. This change significantly simplified implementation and testing. Chapter 3 in [1] provided invaluable insight into working with `gen_server`.

### 4.4 Testing, Type-checking, and Profiling

Testing in the project was done using Erlang's EUnit[11] module. Though the test suite is not extensive, the tests revealed many bugs early and ensured robustness in the basic cases. A particularly useful resource around tests was the EUnit chapter in *Learn You Some Erlang*[12].

Using Dialyzer for type-checking proved invaluable in catching bugs early and reasoning about the types of inputs and outputs of functions. This feature is essential in developing larger-scale projects where there are lots of working pieces.

Profiling has been another insightful tool for both evaluation and testing of the prototype. The initial bug regarding updating finger tables was revealed in the first profiling of the performance, where a single node did the majority of the work, even though the requests were sent to random nodes. This revealed that all the finger tables pointed to that node and forwarded their requests only to that specific node.

Profiling also played a critical role in determining the implicit factor that affected the performance of the system: **the key size**. The key size can greatly affect the overhead of hashing, and thus, it must be considered when a DHT will be used to store large data with large keys.

# Master's Thesis Proposal

## 5 Problem Statement

Distributed Hash Tables (DHTs) are scalable and highly available systems that allow for fast lookup across a network of nodes. DHTs are usually analyzed in their communications overhead, either in terms of latency or number of messages to perform an operation. However, other factors such as hashing overhead or network topology, are not considered, which are major factors in the real-world implementation of DHTs.

This thesis analyzes the effect of hashing and network topologies on the Chord protocol, which is a peer-to-peer protocol for implementing a DHT. The effects of hashing and network topology on the performance of a peer-to-peer network such as Chord provides greater insight into the performance analysis of other DHTs and provides a real analysis into the gap between theoretical bounds on protocols and their real-world performance. The real-world performance of DHTs is important as factors such as hashing overhead or network latency can overshadow the theoretical bounds of the number of messages.

## 6 Methodology

The proof-of-concept done for CPSC 521 provided the opportunity to first learn and understand Chord at the implementation level and then experiment with varying parameters, such as node count and key size. The lessons learned from evaluating the prototype reveal that, in fact, factors other than messaging can significantly affect the performance of a DHT. Thus it would be feasible to further investigate and reveal more hidden factors that can affect the real performance of a system.

For example, in profiling the prototype, it was revealed that a significant portion of computation time is allocated to transforming terms, in particular strings, to binary via `term_to_binary`, as well as calculating the powers of 2 using a recursive power function. The latter can be replaced with a bit-shift operation, which is a primitive operation in Erlang.

The research goal of this thesis can be further expanded to reveal more hidden factors. For example, given a persistent model of the DHT where values are written to disk, how would disk latency affect the

[11] `https://www.erlang.org/doc/apps/eunit/chapter.html`
[12] `https://learnyousomeerlang.com/eunit`

performance of a node? Another notable example is how consensus protocols and fault tolerance could affect the performance of a cluster of nodes that replace a single node in owning a region of the ID space.

Lastly, other DHT protocols such as Kademlia[2] can be implemented and compared with Chord with respect to factors other than message passing overheads. Such comparisons could provide great insight into generalizing the effects of secondary factors to DHTs.

The current prototype is not a full-featured Chord implementation as it is missing the functionality for nodes leaving, nodes concurrently joining, and Key-Value persistence. There are also a number of bugs in the current implementation. The next step for the project would be to complete the Chord implementation and fix the bugs.

## 7 Timeline

| Months | Milestone |
|---|---|
| Dec 2023 - Jan 2024 | Complete the implementation |
| Feb - Mar 2024 | Further Profiling |
| Mar - Apr 2024 | Compare with other implementations |
| May 2024 | Workshop Paper / First Draft |
| June 2024 | Revision of First Draft |
| Jul - Aug 2024 | Implement a Network Simulator |
| Sept 2024 | Analyze Varying the Network Topology |
| Oct - Dec 2024 | Second Draft |
| January 2025 | Conference Paper |
| Feb - Apr 2025 | Writing the Thesis |
| April 2025 | Thesis Submission |

Table 1: Timeline

## 8 Resources

This project does not require any specific resources. Access to servers and potentially a real network test bed would be more than sufficient.

## 9 Uncertainty

Due to the subtleties and complexities of the DHT protocols, the prototypes developed might have bugs that would go undetected. These bugs could potentially affect the measurements. However, with thorough testing and potentially using proof assistants, implementation bugs can be mitigated.

Another potential drawback could be negative results in protocols other than Chord which might not use hashing or already consider the effects of the network topology. Such protocols could prove to be strictly bound by their theoretical bounds as all the possible factors were considered when the bounds were calculated.

Furthermore, algorithms that use non-deterministic timeouts and such could provide varying results compared to deterministic protocols. As such, they could be an interesting point for comparison; However, it would be best to account for the non-determinism when analyzing the results.

## References

[1] Logan, M., Merritt, E., and Carlsson, R. *Erlang and OTP in Action*, 1st ed. Manning Publications Co., USA, 2010.

[2] Maymounkov, P., and Mazières, D. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (Berlin, Heidelberg, 2002), IPTPS '01, Springer-Verlag, p. 53–65.

[3] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (New York, NY, USA, 2001), SIGCOMM '01, Association for Computing Machinery, p. 149–160.

[4] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1 (feb 2003), 17–32.