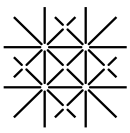

University of Basel

Prof. Dr. Florina Ciorba
Prof. Dr. Heiko Schuldt
Prof. Dr. Christian Tschudin
Prof. Dr. Isabel Wagner

Foundations of Distributed Systems Fall 2024

Exercise 7 Hand-in: 2024 December 8th (13:00pm)

Advisors: Yiming Wu (yiming.wu@unibas.ch)

Modalities of work: The exercise is solved in teams of 2 people.

Modalities of the exercise: The solutions to the exercise must be uploaded to ADAM before the deadline and presented to the advisor. For the presentation, every student is required to schedule an appointment by reserving a slot on `https://xoyondo.com/dp/pzedfa6cfg99hd5`. Every slot has a capacity of two people and for groups, every group member must sign-up. Please note, that reservations are made on a "first come first served" basis. During the meeting, the work and the understanding of the technical background is evaluated. Therefore, each group member has to attend this meeting in order to be awarded points for the exercise. Please be prepared to demonstrate your solutions on your machine! If you need to have the interview via zoom, please send an email to Yiming Wu (yiming.wu@unibas.ch)

Introduction

In this exercise, you will implement a Peer-to-Peer (P2P) overlay network based on the CHORD algorithm. The network of nodes builds up a data store. The data should be distributed among all available nodes in the network. Each *data item* is represented as a *key-value pair*: the *key* uniquely identifies the data item and is used to reference and search for the item in the P2P network. The *value* contains the payload data.

Nodes in the P2P network offer the following operations to client applications:

- **join(n)** joins the P2P network using another node n.
- **leave()** leaves the P2P network (if it is part of the network).
- **store(key,value)** stores data in the P2P network. May be invoked from any active node. It should be able to locate the node "responsible" for the provided data item.
- **lookup(key)** queries for data with the given key. Same properties as store().

- **delete(key)** deletes the data with the given key. Same properties as store().

We assume, that every node has a unique identifier (such as an IP address).

Prerequisites

The goal of this exercise is to implement a P2P data storage over a network of *virtual nodes* by using the CHORD algorithm on top of *Java*. *Virtual nodes* means that these are node objects running in one virtual machine rather than computers on a network. This allows you to focus on the P2P algorithm.

We provide you with a Java project that simulates the CHORD network. Basically, this exercise involves completing the implementation of the required classes and methods in order for the code to do something useful. Some advice on how to use the simulation framework is given in the Markdown file accompanying the project!

Requirements: Java 17. Other Java versions can be used but one must adjust the OpenJFX dependency, to do so.

The framework provides you with the functionality required to solve this exercise.

- `Identifier` class represents an identifier on the circle (given a value of m).
- `IdentifierCircle` class lets you "lookup" identifiers from hash values.
- `HashFunction` class provides the hashing functionality.
- `IdentifierCircularInterval` class deals with the modulo arithmetic on circular intervals.

Use these classes as you see fit. The `ChordNetwork` and/or the `ChordPeer` classes provide accessors for usable instances of most of them. For example, you can access to the underlying hash function via `ChordNetwork#getHashFunction()`.

Chord

As you already know from the lectures, the CHORD algorithm is based on identifiers that are ordered on an identifier circle modulo 2^m called the *CHORD Ring*, where m is the number of bits used to construct the identifier. In order to keep the bits for peer identifiers and key identifiers constant and also to have an even distribution of load amongst the peers, a *hash function* is used to derive the *identifier* from the peer's address and/or the keys of the data items.

One important term in CHORD is the *successor of* k ($successor(k),\ with\ 0 \leq k < m$), which is the first node that follows after or is equal to the given identifier k. Location

of the node that is responsible for a particular data item is handled via the *successor(k)* relationship. For a particular data item, the identifier of its key is obtained using the underlying hash function. The node that succeeds the resulting identifier, i.e. *successor(id)*, is reponsible for that particular data item both during storage and lookup. This is illustrated in Figure 1. Each node holds a table of keys in its responsibility.

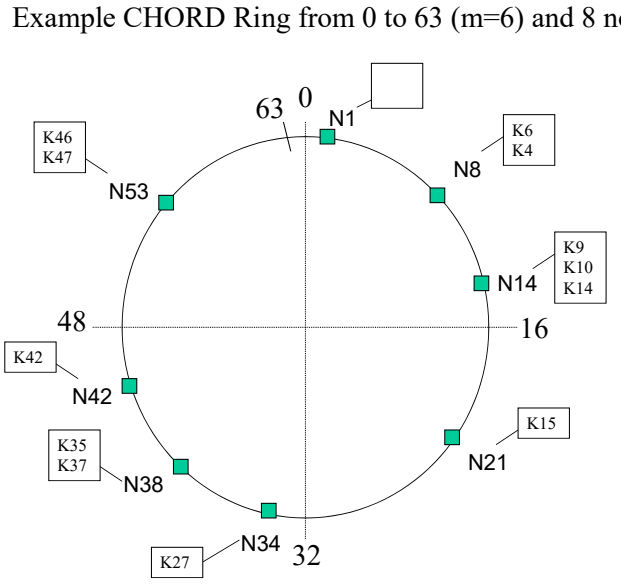

Figure 1: Example CHORD Ring for for $m = 6$.

Now we need a strategy to find the *successor* node for a given identifier. In a real world P2P network, where potentially millions of peers are part of it, it is not feasible for a peer to keep a list of all the other peers in the network, which would be needed to lookup the successor for any request that arrives. A naive solution to this problem is illustrated in Figure 2, in which case each peer knows only its immediate successor[1]. Any request concerning a particular data item has to be passed from successor to successor until it finally reaches the node that hosts the data item. In the worst case, the request will be passed along the entire circle.

Therefore, CHORD proposes an advanced mechanism for query routing, which involves a so called *finger table*. The finger table of a node n has m entries, where the i-th finger (*finger[i]*, one based indexing) points to the peer succeeding n by at least 2^{i-1}. Consequently, *finger[1]* must be the immediate successor of n (calculate and see for yourself). Figure 3 illustrates routing based on the finger table.

Now of course, with nodes joining and leaving a CHORD network, the question is how nodes can keep their *finger tables* up-to-date and how data is re-distributed, as nodes become available or disappear. This is, what we are going to look at in this exercise in

[1]Note, that when talking about the successor of a *node* n, one starts looking at the position $n + 1$.

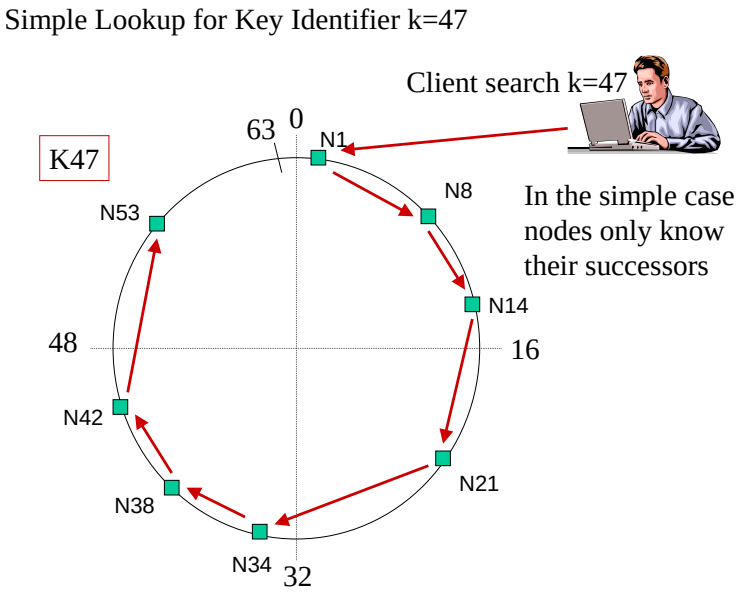

Figure 2: Naive CHORD Routing. Each node knows only its successor.

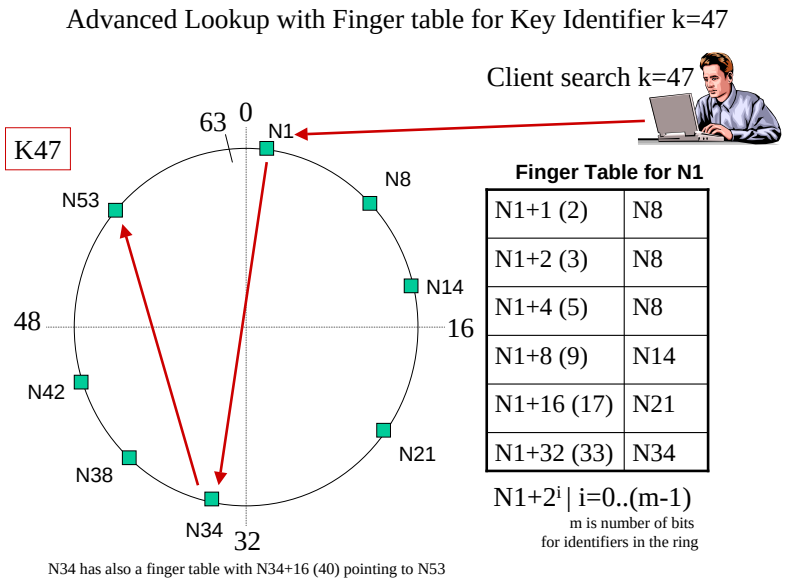

Figure 3: Advanced CHORD Routing

more detail. In fact, the SIGCOMM CHORD paper [1] presents two different strategies for maintaining the network in a correct state – namely the ones presented in Figures 6 and 7 of the paper. There is a moderate coding effort required for this exercise, but the goal is for you to *understand* the inner workings of the protocol! Therefore, you should carefully read through – and understand – the paper!

Question 1: Basic Chord Network (12 points)

Complete the implementation of the `ChordPeer` class to create a functional CHORD overlay network in the static case without stabilization (see [1], Figure 6). Use the settings described in the paper ($m = 3$). This allows you to compare your results with the provided examples. You should **NOT** be required to change the remaining classes, including but not limited to `ChordNetwork`, `AbstractChordPeer`, `SimulationNetwork` and `SimulationPeer`. However, if you want to you can, but be prepared to justify your decision.

a) Complete the implementation of the `ChordPeer` class, so as to allow for bootstraping the network and for new nodes joining it. In the case without stabilization, which we are looking at now, joining of new nodes is supposed to be handled by the `ChordPeer#joinAndUpdate()` method.

Hint: Please mind, that the pseudo-code in [1], Figure 6 for *update_others* and *update_finger_table* is not quite correct. Read carefully through the description and try to spot the difference. Think about the behaviour in a two-node (bootstraping) scenario, where Node 1 uses Node 0 to join the network.

(6 points)

b) Now that the a basic CHORD network can be setup, implement basic data storage and lookup. To do so, implement `ChordPeer#lookupNodeForItem()`.

(2 points)

c) Once a new node joins the CHORD network, responsibility of the existing data can change. Handle re-distribution of data in `ChordPeer#joinAndUpdate()`.

(4 points)

Question 2: Dynamic Joining/Departing (8 points)

The solution implemented in the previous question imposes a lot of overhead upon joining, since finger entries all need to be updated at once. Furthermore, departure of nodes is not handled at all. The authors of [1] propose a solution to this problem, which is called stabilization and involves regular invocation of some logic (see [1], Figure 7).

a) Implement the mechanism proposed in [1], Figure 7, by implementing the required methods. **Hint:** To test these changes, you must run the simulation in dynamic mode (set *dynamic* flag to *true*, when starting the simulator).

(5 points)

b) Now that dynamic joining is supported; what needs to be done in order to maintain correctness after departure of a node? This is actually not described in detail in [1]. Try to find a simple solution using the existing stabilization mechanism. You can assume, that departure takes place in an orderly manner, hence, a departing node can notify other nodes.

(3 points)

Question 3: Interview (0 points)

During the presentation of your solutions, you will be asked questions related to the topic of the exercise (P2P, CHORD, etc.) as well as your specific solutions. Also, please be prepared to run a demo on your local machine!

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proceedings of the ACM Conference on SIGCOMM*, (San Diego, USA), pp. 149–160, 2001.