D7024E Lab assignment

Peer-to-peer distributed file system

Introduction

The primary task of this assignment is the implementation of a working Distributed File System (DFS)[1]. The goal is quite simple; the system must facilitate storing and retrieving of files associated with arbitrary names, while allowing anyone to join the distributed system with their own file servers. All data stored by the system must be immutable, meaning that it cannot be modified or disappear once uploaded to the network.

At the center of your system there will be a so-called Distributed Hash Table (DHT). A DHT is similar to an ordinary hash table in the sense that it allows you to lookup and store key/value[6] pairs by the hashes of their keys. The difference, however, is that in a DHT, data is not stored in local memory, but is spread out over a network of multiple nodes. In a naive storage system spanning over multiple nodes, a single node maintains an index, which is a list of all stored key/value pairs and on what nodes they are stored. Whenever data is stored or retrieved, the index must be updated or consulted. The advantage of a DHT is that no central index is needed. Rather, each node uses a specific algorithm to determine where a value is stored only from its key, and that algorithm typically involves hashing the key. The key, hashed or not, is matched against a local list of known network participants. If the correct node is in that list, it is contacted directly. If the correct node is not in the local list, the node believed to most likely to know about it is consulted. As this removes the need for having any central node, the system becomes incredibly scalable and can be made highly tolerant against node failures, if each key/value pair is replicated across multiple nodes.

Your particular DHT implementation will adhere to the Kademlia algorithm[7], which you will have to study as part of this course. It will allow clients wishing to upload or download files to do so by contacting any of the nodes part of the system. When a file is uploaded, the contacted node takes the uploaded copy of the file, and sends a copy of it to each node explicitly designated by your Kademlia algorithm. When a file is downloaded, the contacted node takes responsibility for sending it to the downloading client, even if it must be retrieved from another node first. To ensure that the DFS is not flooded with file copies no one will ever use, each node maintains an

---

[1] The system you will build is inspired by the InterPlanetary File (IPFS), which you can read more about here: https://ipfs.io. Another highly related project is Information-Centric Networking (ICN), which is an attempt to build an entirely new Internet around principles similar to those employed by IPFS and the system you will build. You can read more about ICN here: https://irtf.org/icnrg.

[6] When talking about hash tables, the terms *key* and *value* are typically used, while the terms *name* (or *path*) and *file* are used when talking about filesystems. We use *name/file* and *key/value* interchangeably since your DFS is based on a DHT.

[7] Kademlia is used by projects such as BitTorrent and IPFS. It is also an important component for building distributed cloud platforms. Compared to other DHTs, like Pastry or Chord, it is fairly easy to implement.

expiration time for each stored file. When a file expires, it is deleted from that particular node. However, clients must be able to request that particular files are *pinned*, which means that some of its copies never expire. In particular, the node receiving the original file copy and the designated nodes must store pinned files indefinitely.

Pinning files. The pinning and unpinning of files can be implemented in a variety of ways. When a client uploads a file to the network, the file is first stored by the contacted node, which we will refer to as the *original publisher*. This node, as per Kademlia, will distribute and store the given file at certain nodes in the network, which we will refer to as *recipient nodes*. Recipient nodes MUST republish their data, to the correct subset of recipient nodes, with a regular interval (e.g., every 24h). If a node that holds a given file is not republished, then after some time (e.g., 24h+) that data should be purged from that particular node. However, if a file is pinned at any given node, then that data should not be purged, independently of if republishing happens or not. Unpinning could simply release a file so it is eventually purged from that particular node as a by-product of purging. Unpinning could also be defined so that if a file is unpinned, it creates a chain reaction within the system that will forcefully delete the file from the whole network of Kademlia nodes. In this scenario, perhaps pinning could ensure that nodes are able to maintain their copy of the file even if system wide purging of a file is set in motion. In the latter definition of unpinning, a decision about which nodes are able to unpin data from the network needs to be considered. It could be that only the original publisher should be able to unpin, and thus, irrevocably delete data from the network or it could be done by any node in the network. Furthermore, Kademlia requires the original publisher to republish the file it uploaded to the recipient nodes. However, whether this is part of your solution or not, has been left up to you. These are only some examples illustrating different variations of pinning and unpinning, along with their effect on republishing and purging. These will garner different system properties, some of which are simpler solutions whilst others are more intricate. These design choices, and thus the definitions of what pinning and unpinning means within your system, are left up to each group. These choices will have to be properly explained at their corresponding meeting and in the lab report.

Containerization. Your implementation of Kademlia must run containerized. That is, you will set up a group of Docker containers, where each container shall represent a node in the Kademlia network. A Docker image is provided in this lab. This image already has golang and protobuf preinstalled. You may either use this image as a starting point, create your own or use some other image you find online. The repository for the provided image is larjim/kademlialab, and its description can be found at https://hub.docker.com/r/larjim/kademlialab/.
You will need to dynamically deploy a 100 or more containers (i.e. Kademlia nodes). There are many ways of deploying a group of containers and attaching them all to a Docker network: 1) You could create your own script that uses the Docker CLI to create a Docker network, then deploy a given number of containers and attach each to the created network. 2) You could use a Docker Compose file (.yml / .yaml) to define the network and the services to deploy. The Compose file can then be deployed through Swarm via “deploy stack”, for example. 3) You may instead want to use Docker Compose directly, to deploy the specified network and containers within your host. 4) Perhaps a solution of scripting together with a Compose file might be best.
Use whichever solution is best for you. For example, if your solution is going to run on a single Docker host, then setting up a user-defined bridge network would allow you to have all the containers find each other, as containers in this network will expose all ports to each other and no ports to the outside world. When creating the Docker network remember to think about which driver makes sense for your deployment. You should also think about Kademlia’s bootstrapping

process. Nodes will need to either know or obtain the IP of a node to bootstrap to, so you should keep this in mind.
As an example, a Docker Compose file to be used with "stack deploy" is provided. The file can be found in Canvas. This is simply an example; it is not a directive that sets how you should do this. This file will set up 5 containers from the image that is provided in the lab. It also creates a network: kademlia_network. Because this is deploying through Swarm, it will create an overlay network. All the containers get an IP on this network. You can attach to a running bash shell on any of these replicas by "docker attach <container>". You can take a look at the network configuration through "docker network inspect <theNetwork>"
To run the example:

1) docker swarm init #Initiate Swarm manager
2) docker stack deploy -c docker-compose-lab.yml putHereNameOfApp
3) docker stack rm putHereNameOfApp #Take down the app
4) docker swarm leave –force #Take down the swarm

Working methods

There are a few guidelines for the lab assignment:

The lab should be done in groups of 2-3 students.

- A key ability is to search, find and re-use existing knowledge, software and other information that can bring you forwards effectively. Obviously, you cannot use software that directly solves the lab assignment. Don't forget to make references.

- In your reports you must name the people you have cooperated closely with, and which persons you have discussed with (if it has had impact on your solution). Obviously, you must reference all external information (web-pages, papers etc.) you have used. The assignment is defined in general terms rather than specific terms, which means we expect all solutions to be different, to some degree.

- The lab should be implemented in Golang[8]. However, if you decide to use another language, you are on your own and cannot expect help to the same extent.

- All students, within their group, are expected to participate in equal measure in performing the assignment.

Workflow

It is encouraged to follow an agile and modern demo-driven development method[9], i.e., development and adaptation in small steps accompanied with a demonstration at the end of each

[8] See https://golang.org.
[9] See https://en.wikipedia.org/wiki/Agile_software_development for details. Consider the sections on working iteratively/evolutionary especially important, as the other aspects assume a situation that cannot be easily emulated as part of a university course.

step. The lab consists of several objectives. You must divide and prioritize the work and put it into sprints. You may use as many sprints as you like, but to pass the lab assignment you should meet at least three times with your instructor, first time after a week to present your priorities and initial backlog, then two times to make demonstrations, first demo after about 3 weeks and second demo by the end of the course when you are done.

VERY IMPORTANT: Make sure you can demonstrate each objective separately.

Additionally, you must write a lab report containing the following information:

- Use-cases
- List of requirements (backlog), and assumptions.
- Sprint plan
- Frameworks and tools you are using.
- System architecture description and an overview of the implementation. Motivate all design choices you have made.
- Threading model, including how concurrent requests for modifications of the routing tables and other critical data structures are serialized.
- Links to code (i.e., your teacher should be able to check the code). You are encouraged to publish your work as open source on online project hosting services e.g. GitHub.
- A description of the system's limitations and the possibilities for improvements.

For each meeting with your instructor, upload your report to Canvas and book a time slot to present your solution. There will be three hand-ins in Canvas where you can upload. The work is incremental so already at the first review you should have some draft of your report with your initial principal approach or solution for the objectives, including initial backlog and sprint plan. Then it will improve over the following sprints. Remember that you are going to develop a proof-of-concept prototype to learn how to develop distributed systems, and not a fully working bug free product.

Objective 1 – Understanding Kademlia

Your first task is to learn the Kademlia algorithm. You need to understand the algorithm in detail in order to implement the lab, so read the Kademlia paper[10] thoroughly and all material you can find on the Internet. If you don't know how to program in Golang[11], this might also be a good opportunity to spend some time learning it.

To get you up to speed, we provide you with some source code that you can use and build upon. To be more specific, you get the source code for managing the routing table required by Kademlia. The code is not complete, and you will have to modify it later on—but it is a good starting point. The routing table is a simplified version of the one described in the Kademlia paper. The Kademlia paper describes a solution where the routing table is implemented as a binary tree, but the one in the simplified version provides only a fixed list of buckets.

---

[10] See https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf.
[11] A good place to start learning it may be https://tour.golang.org.

The file *routingtable_test.go* contains an incomplete unit test of the routing table. You can increase your grade if you provide additional unit tests, demonstrating the code in question complies with the paper description. The test code ought to give you a rough idea on how to use the routing table. The function *NewRoutingTable* creates a new routing table and takes k (the replication factor) and the contact information of the local peer as arguments. Each peer in the network is represented by a *Contact* object, which contains a K*ademliaID, an* IP address and port number. The K*ademliaID* is an opaque 160 bit integer, which could be generated using a random number generator or hash of an UUID, for example. For testing purposes (as in the example below), it could also be hard-coded to some specific values.

```
rt := NewRoutingTable(NewContact(NewKademliaID("FFFFFFFF00000000000000000000000000000000"), "localhost:8000"))


rt.AddContact(NewContact(NewKademliaID("FFFFFFFF00000000000000000000000000000000"), "localhost:8001"))
rt.AddContact(NewContact(NewKademliaID("1111111100000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111200000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111300000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("1111111400000000000000000000000000000000"), "localhost:8002"))
 rt.AddContact(NewContact(NewKademliaID("2111111400000000000000000000000000000000"), "localhost:8002"))


contacts := rt.FindClosestContacts(NewKademliaID("2111111400000000000000000000000000000000"), 20)
for i := range contacts {
    fmt.Println(contacts[i].String())
}
```

The example above creates a routing table and populates it with some fake contacts. The *kademlia.FindClosestFunction* returns other peers in the network that are close the specified target. If you read the Kademlia paper, you will understand that distance is the XOR value of two identifiers. Play around with the code and make sure you fully understand every line of it.

Meeting 1, requirements: DATE

- Understanding the provided code.
- Being able to use the Compose file to deploy the would-be nodes.
- Complete understanding of Kademlia.
- Submitted lab report by DATE. The required content of the lab report is described in the *Workflow* section.

The meeting with your instructor will revolve around clarifying requirements and design choices for the system. It is absolutely necessary for all students to meet these requirements prior to the meeting.

Objective 2 – Fully Functional Kademlia System

This is the toughest part of the lab. You should now implement a fully working version of Kademlia. In order to do this, you first need to implement a UDP based protocol to allow peers to exchange messages between each other. To save some time, it is recommended that you use Google Protocol Buffers[12], but you can also implement your own parser and message format if you prefer that, for example using JSON.

The next step is to implement the *LookupContact*, *LookupData* and *Store* functions, as described in the Kademlia paper.

---

[12] See https://github.com/google/protobuf.

To simplify a bit, the *LookupContact* function can be implemented by: 1) first looking up the α closest contacts to a specific target using the local routing table, and then 2) sending a message to each of these peers, asynchronously, to learn about which are the *k* closest contacts to the target that they have in their routing table. This is done recursively until there are no more contacts to query or no new contacts are discovered. As a side effect, the local routing table is populated with new contacts for each query. α and *k* are Kademlia system-wide variables and their values are, typically, 3 and 20 respectively.

Note that all queries ought to be executed concurrently, which means that you need to figure out a strategy how to handle concurrency. Although Golang has Mutex support, it is highly recommended to use Golang's channel mechanism for communication between Goroutines. You should also note that concurrent code has a tendency of quickly becoming messy and error-prone, resulting in hard-to-catch bugs such as race conditions or deadlocks. It is, therefore, recommended for you to use as few Goroutines as possible, and to be sure to carefully plan every Goroutine you spawn. One possible strategy is to have one Goroutine being responsible for executing tasks generated by other Go routines. If it becomes apparent that your code is not thread safe, your grade will be affected negatively. Describe the threading model you created in your report.

Also note that you should also implement a mechanism to remove data from the network (i.e. a file is only protected if *pinned*, otherwise it will be deleted from a node after some time, if the file is not republished to that node within a given time), as well as a re-publish mechanism. The purpose of the re-publishing mechanism is to make sure that data is always stored by the peers closest to each given target. What do you think would happen if the network rapidly increased in size? This could be quite a severe issue. How do you handle it? Write a section in the lab report mentioning theoretical problems with Kademlia and possible solutions.

The last step in this objective is to develop a unit test that simulates a Kademlia network of at least 100 peers. The unit test should prove that your Kademlia implementation is working correctly. By calling **go test,** a large number of tests should run and result should be **PASS**.

Meeting 2, requirements: DATE

- Working implementation of:
  - FIND_NODE RPC
  - Lookup algorithm
  - STORE RPC
  - FIND_VALUE RPC
  - PING RPC
  - Republishing
  - Bootstrapping
  - Unit test that proves Kademlia and all its most important functionality to be working
- Deployment of 100 containerized nodes
- Submitted lab report by DATE

The meeting with your instructor will involve a thorough demonstration and explanation of a fully functional Kademlia solution and the design choices made. Unit test/s will be assessed.

Objective 3 – File system

It is now time to use the Kademlia implementation to implement the distributed file system mentioned in the introduction. The first you need to do is to implement a *Server* that runs in the background as a daemon. The *Server* uses the Kademlia implementation developed in Objective 2 and connects to the Kademlia network. The *Server* also offers a REST API, and provides a Command-Line-Interface (CLI) for fetching and store files. For example:

$ dfs store myFile.txt
eb9d5f1f1874cfa48c5442e6172d45d307267eb9

The *dfs store* command should return the hash of the stored file, i.e. the address of the file in the Kademlia network.

The *dfs cat* command should print the content of a file given the address as argument, e.g.
$ dfs cat eb9d5f1f1874cfa48c5442e6172d45d307267eb9
… content of my file …

The *dfs pin* command should make sure important data is not deleted.
$ dfs pin eb9d5f1f1874cfa48c5442e6172d45d307267eb9

Finally, the *dfs unpin* command should remove pinned data.
$ dfs unpin eb9d5f1f1874cfa48c5442e6172d45d307267eb9

Optional Objective X – Republishing optimization and refreshing :

The Kademlia paper presents some variations of republishing in the form of republishing optimizations. Consider implementing one of these described optimizations or better yet, come up with your own optimization. Moreover, one of Kademlia's properties, which has not been made mandatory to this lab is the refreshing of the routing table. Refreshing should be in operation during the whole lifetime of the system as well as during the bootstrapping process.

Optional Objective Y – Experimental analysis of system variables and their effect on performance

You could perform an analysis of which α, bucket size (k), and replication factor (k) values perform best, depending on the number of nodes in the network, churn rate (i.e. how many nodes leave the network), how many new nodes join or old nodes rejoin, number of files in the network, delay between nodes, etc. This is an example of a possible experiment to analyze the performance of Kademlia and some suggested variables that could be of interest. However, you may choose your own experiment where you may focus on whichever variable/s are of interest to you.

Optional Objective Z – Your own idea?

Contact the lab teacher if you have an idea and what to implement that. Why not develop a decentralized supercomputer platform similar Golem[13]? That could also be a good opportunity to play around with cloud platforms like Docker and Kubernetes if you are interested in that.

---

[13] See https://golem.network.

Note that implementing optional objectives grant you a better grade.

Grading

+1 Complete unit tests
+1 Your implementation works with that of another lab group
+1 One optional objective implemented
+1 A working threading model described in the lab report
-1 Non thread-safe code (i.e, vulnerable to race conditions)
-1 Incomplete or too simplistic unit tests
-1 Incomplete Kademlia implementation