D7024E Lab assignment 2017

Peer-to-peer distributed file system

Introduction

The main task of this assignment is to implement a proof-of-concept prototype of a working distributed system. In short, you are going to develop your own distributed file system inspired by the InterPlanetary Filesystem System (IPFS)[1] project. The idea is quite simple; anyone can set up a server and provide storage resources to other users without any permission. An important property of the system is that all stored data is *immutable*, meaning that it cannot be modified once uploaded to the network. The file system you are going to develop resembles an Information-centric Network (ICN), but is developed completely in the application layer and deployed on top of the Internet without specific network support.

A core component of the system is a so-called Distributed Hash table (DHT), which is an algorithm used to find out which servers to use to either store or download data from. Since data is immutable it becomes possible to calculate a cryptographic hash of the data, e.g. using SHA-2, which could be viewed as a unique fingerprint of the data. The DHT is used as a key-value store to map the hash of the data to contact information of servers storing the actual data. In other word, the DHT is used as a lookup mechanism to find out where to store or download data.

When a new file is uploaded to the network, the actual data is always stored in the user´s local computer. The DHT is then updated so that other users on the network can find the file and download it directly from the user´s computer. When another user downloads the file or data, they keep a local copy of it and then updates the DHT. In this case, the file is now replicated at two locations, which means that the more popular some data becomes; the more it will be replicated in the network. However, if a file has not been downloaded for some time, it should automatically be purged. To prevent purging important data, users should also have the possibility to *pin* important data to prevent it from being deleted.

A large part of the lab is going to be spent on developing your on DHT. The algorithm you are going to implement is called Kademlia and is very popular. It is for example used in both BitTorrent and IPFS. It is also important component building distributed cloud platforms. Compared to other DHT like Pastry or Chord, it is also fairly easy to implement

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

[1] https://ipfs.io

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.) that you have used. The assignment is generally defined, so we expect all solutions to be different to some extent.

- The lab should be implement in Golang. However, if you decide to use another language, you are own your own and cannot expect as much help.

Workflow

It is encouraged to follow an agile and modern demo driven development method, i.e., development and adaptation in small steps accompanied with a demo at each step. The lab consists of several objectives. To pass the lab assignment you should make a demonstration for the lab teachers two times: sprint 1 after 2-3 weeks, and sprint 2 by the end of the course when you are done.

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

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

- Use-cases, requirements, assumptions.
- Frameworks and tools you are using.
- System architecture description and an overview of the implementation. Motivate all design choices you have made.
- 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 sprint, upload the report to Canvas and sign up for review with your supervisor(s). There shall be a draft report already at the first review, where you have entered information that is critical for your solutions. At the first review you should have a principal solution for each objective (not yet implemented). You should also have a prioritized backlog for each of the objectives

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[2] thoroughly and all material you can find on the Internet. If you don't know how to program in Golang, this might also be a good opportunity to spend some time learning Golang.

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 to routing table used in Kademlia. The code is not completed and you will have to modify later on, but it is a good starting point. The routing table is simplified version of the routing table described in the Kademlia paper. The Kademlia paper

[2] https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

describes a solution where the routing table is implemented as a binary tree, but the in the simplified version it is just a fixed list of buckets.

The file *routingtable_test.go* contains an incomplete unit test of the routing table. In general, you can increase your grade if you implement complete unit tests. Anyway, the test code gives some rough idea how to use the routing table. The function *NewRoutingTable* creates a new routing table and takes the contact of the local peer as argument. Each peer in the network is represented by a *Contract* object, which contains its *KademliaID* and information (IP and port) how to connect to the server. The *KademliaID* is an opaque 160 bit number, and can for example be generated using a random number generator or hash of UUID etc. For testing purposes (as in the example below), it can 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("11111111000000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("11111112000000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("11111113000000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("11111114000000000000000000000000000000000"), "localhost:8002"))
rt.AddContact(NewContact(NewKademliaID("21111114000000000000000000000000000000000"), "localhost:8002"))

contacts := rt.FindClosestContacts(NewKademliaID("21111114000000000000000000000000000000000"), 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 *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.

Objective 2 – Fully working Kademlia

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 recommend that you use protobuf[3] for serialization and remote invocation, but you can select another package if you prefer, or alternatively 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* function as described in the Kademlia paper.

To simply a bit, the LookupContact can be implemented by first looking up the closest contacts to a specific target using the peer's local routing table, and then send a message to each of these peers to learn about their closest contacts in regards to the target. This is done recursively until there are

[3] https://github.com/google/protobuf

no more contacts to query or the no new contacts are discovered. As a side effect, the local routing table is populated with new contacts for each query.

Note that all queries should be done in parallel, which means that you need to figure out a strategy how to handle concurrency. Although Golang support Mutex, it is highly recommended that you use Golang's channel mechanism to communicate between Go routines. However, is very easy that the code becomes messy, resulting in bugs or deadlocks. For this reasons, it is recommended that you use as few Go routines as possible, and you need to have plan for every new Go routine you spawn. One good strategy could be to have only one single main Go routine that executes different tasks generated by other Go routines. You will get minus points if it apparent that your code is not thread safe. Describe your threading model in the report.

Also note that you should also implement a mechanism to purge data from the network, as well as a re-publish mechanism. The purpose of the re-publishing mechanism is to make sure that data is always stored at closest peers in regards to the target. What do you think will happen if the network rapidly increases in size? This can be a quite 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**.

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* should also offer a REST API allowing Command-Line-Interface (CLI) to fetch and store files, for example:

$ dfs store myFile.txt
eb9d5f1f1874cfa48c5442e6172d45d307267eb9

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

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

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

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

Optional Objective – Blockchain integration

The filesystem you have developed so far is completely flat as it does not support directories. It also does not have support for access control.

Since the filesystem you have developed is complete decentralized, it could make sense to complement it with a decentralized directory and user management mechanism. In this objective you are going to develop a directory file data structure using smart contracts. The rough idea is to store data using the filesystem developed in Objective 4, but then store the hashes in a blockchain using a smart contract. The smart contract would then store the hashes as states on the blockchain. It would also function as a gatekeeper so that only authorized users are allowed to add files to the blockchain.

Preferable this objective should be implemented using a permission-less public blockchain, for example Ethereum.

Optional Objective – NAT traversal

Add support for NAT traversal, for example using UPNP or STUN.

Optional Objective – 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[4]? That could also be a good opportunity to play around with cloud platforms like Docker and Kubernetes if you are interested in that.

Note that implementing optional objectives grant you a better grade.

Grading

+1 Complete unit tests
+1 Interop with another lab group
+1 Implemented for each optional objective implemented
+1 A working threading model described in the lab report
-1 Non thread-safe code
-1 Incomplete or too simple unit tests
-1 Incomplete Kademlia implementation

[4] https://golem.network