D7024E Lab Assignment

Creating a Peer-to-Peer Distributed Data Store

Introduction

The objective of this assignment is to produce a working *Distributed Data Store* (DDS)[1]. In contrast to a traditional database, a DDS stores its *data objects*[2] on many different computers rather than only one. These computers together make up a single *storage network*, which internally keeps track of what objects are stored and which *nodes*[3] keep copies of them. The system you are to create will be able to store and locate data objects by their *hashes* [4] in such a storage network, which is formed by many running instances of your system.

The primary intention of the lab is to help you understand, both practically and theoretically, how modern distributed applications are built and some design challenges faced while constructing them. We want you to see for yourself how these are more complicated and less efficient than their traditional client-server counter-parts, but also how they facilitate extraordinary degrees of scalability, fault-tolerance and concurrency.

We also want to introduce you to some of the tools and technologies that are commonly employed to build and manage distributed systems. In particular, you are adviced to write the program in Google's Go language,[5] which was specifically designed for writing distributed applications, and to manage nodes by using Docker containers.[6] You are not forced to use either, but you will not be able to receive programming or container help from the teaching assistants if you choose to use any other programming language or container solution. In other words, you are required to use a programming language and a container solution, but you are not strictly required to use Go or Docker. If you decide to use Go, we provide code you can use as starting point in the Canvas `D7024E Labs/labCode/` folder.

Lastly, we want to give you a gentle introduction to *Agile Software Development* (ASD)[7]. You will not be assessed about any aspect of ASD, but we use some of its terminology in this lab description. Many of you, if not most, that will end up as software engineers after graduation will work according to this methodology, or at least a variant of it. Using it to complete the lab can be an excellent way for you to coordinate your programming efforts.

Best of luck,
Teachers and Teaching Assistants

[1] See https://en.wikipedia.org/wiki/Distributed_data_store/ for more material on the topic.
[2] Here, we consider a *data object* to be an array of bytes with a well-known structure, such as a UTF-8 text or JSON.
[3] A *node*, or *peer*, is simply a participant in a peer-to-peer network.
[4] See https://en.wikipedia.org/wiki/Hash_function/ if you are unfamiliar with *hashes* and *hash functions*.
[5] A good place to start learning it may be https://tour.golang.org/.
[6] An extensive Docker tutorial is available at https://docs.docker.com/get-started/.
[7] See https://en.wikipedia.org/wiki/Agile_software_development/ for an introduction. 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.

Objectives

To pass the lab assignment, you and your group members must create an application that fulfills the *mandatory objectives*, listed below. Completing only those objectives will give you the lowest passing grade for the lab, if no serious bugs can be observed. To increase your grade, you must complete the *qualifying objectives* listed after the mandatory ones. Your lab grade will depend on the number of points you acquire after having completing all mandatory objectives. The number of points you get for each objective are stated within square brackets after its name.

Note that during assessment, each group member is expected to be able to demonstrate every objective, as well as describing how and why you approached it the way you did.

Mandatory

M1. *Network formation.* **[5p].** Your nodes must be able to form networks as described in the Kademlia paper.[8] Kademlia is a protocol for facilitating *Distributed Hash Tables* (DHTs).[9] Concretely, the following aspects of the algorithm must be implemented:

   (a) **Pinging**. This means that you must implement and use the `PING` message.

   (b) **Network joining**. Given the IP address, and any other data you decide, of any single node, a node must be able to join or form a network with that node.

   (c) **Node lookup**. When part of a network, each node must be able to retrieve the contact information of any other node in the same network.

M2. *Object distribution.* **[5p].** The networks your nodes form must be able to manage the distribution, storage and retrieval of data objects, as described in the Kademlia paper.[10] Concretely, you must implement the following aspects of Kademlia:

   (a) **Storing objects**. When part of a network, it must be possible for any node to upload an object that will end up at the designated storing nodes[11].

   (b) **Finding objects**. When part of a network with uploaded objects, it must be possible to find and download any object, as long as it is stored by at least one designated node.

M3. *Command line interface.* **[5p].** Each node must provide a command line interface through which the following commands can be executed:

   (a) **put**: Takes a single argument, the contents of the file you are uploading, and outputs the hash of the object, if it could be uploaded successfully.

   (b) **get**: Takes a hash as its only argument, and outputs the contents of the object and the node it was retrieved from, if it could be downloaded successfully.

   (c) **exit**: Terminates the node.

M4. *Unit testing.* **[5p].** You must demonstrate that the core parts of your implementation work as expected by writing unit tests. Note that unit tests *never* cross process boundaries, which means that they do not send network messages, read or write files, require user input, etc. Unit tests prove that the internal constructs of an application behave as expected, such as that the calculation of XOR distances is correct, or that contacts are inserted at the correct places in buckets, and so on. See Appendix A for a brief tutorial. **We expect a test coverage of at least 50%.**[12]

[8] While the official paper should be the document you first examine to understand the algorithm, there are plenty of complementary resources that can be helpful for clarifying some parts of that paper, such as this interactive description or this specification.

[9] See `https://en.wikipedia.org/wiki/Distributed_hash_table/` if you want more details.

[10] Note that in the Kademlia paper, objects are referred to as *values* and their hashes as *keys*.

[11] In Kademlia terminology, the *designated nodes* are the K nodes nearest to the hash of the data object in question.

[12] You can read more about test coverage in Go at `https://blog.golang.org/cover/`. If you use an IDE like GoLand which you can get a student license for via your LTU e-mail address, tools for test coverage are built in.

M5. *Containerization.* **[5p].** You must be able to spin up a network of nodes on a single machine. **The network must consist of at least 50 nodes, each in its own container**.[13] You may spin up and take down the network any way you like, but you will likely save a lot of time if you either use a script or an orchestration solution[14] to start and stop the network.

M6. *Lab report.* **[5p].** You must continously work on and update a lab report. The required contents of the report are outlined later in the Method section.

Qualifying

U1. *Object expiration.* **[2p].** Each node associates each data object it stores with a certain Time-To-Live (TTL). When the TTL expires, the data object in question is silently deleted. However, every time the data object is requsted and transmitted, the TTL is to be reset. You may decide the TTL yourself. It should, however, be changeable so that you can demonstrate that the object expiration mechanism you build does work as intended.

U2. *Object expiration delay.* **[2p].** To prevent files from expiring, the node that originally uploaded each object sends a *refresh command* to the nodes having copies of it to prevent them from being deleted. In particular, the command resets the TTL for the refreshed data object without actually requesting it. As long as the uploading node can contact the storing nodes, the object in question should never expire. Completing this objective requires that U1 is also completed.

U3. *Forget CLI command.* **[2p].** To allow the original uploader of an object to stop refreshing it, you add the **forget** CLI command, which accepts the hash of the object that is no longer to be refreshed. Completing this objective requires that U2 is also completed.

U4. *RESTful application interface.* **[6p].** A CLI interface may be useful for humans with terminals, but makes it difficult to integrate your storage network into web applications or other applications. To remedy this, you will make every node also provide a RESTful[15] HTTP interface with the following endpoints:

   (a) **POST /objects**: Each message sent to the endpoint must contain a data object in its HTTP body. If the operation is successful, the node must reply with `201 CREATED` containing both the contents of the uploaded object and a `Location: /objects/{hash}` header, where `{hash}` must be substituted for the hash of the uploaded object.

   (b) **GET /objects/{hash}**: The `{hash}` portion of the HTTP path is to be substituted for the hash of the object. A successful call should result in the contents of the object being responded with.

U5. *Higher unit test coverage.* **[2p].** This objective is considered completed if you can get **a unit test coverage of 80% or higher**. To reach such a level of test coverage, you may have to use mocking, which means that you create fake versions of components that would normally cross the process boundary. For example, you could create a fake message sending component that can record network messages and then list them in the order they were submitted, which would allow you to test if sent messages are correct and provide mock responses.

U6. *Concurrency and thread safety.* **[6p].** To complete this objective, you must use some form of concurrency construct, such as threads or Go channels, to make the handling of messages concurrent. You must also be able to account for how you guarantee thread safety, via locks or otherwise.

[13] If you choose to use Docker as container solution, which we recommend, you may want to use the Docker image described at https://hub.docker.com/r/larjim/kademlialab/ as starting point.

[14] Docker compose, which you can read about at https://docs.docker.com/compose/, may be a good alternative if you choose to use Docker. There is already a Docker compose file named `docker-compose-lab.yml` on Canvas in the `D7024E Labs/` folder you can use as starting point.

[15] As described at https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

Delimitations

To make this task manageable within the time frame of this course, we make these delimitations:

1. Data objects can **not** be explicitly deleted or modified. In other words, they must be *immutable*. They can still, however, expire or be lost.
2. Data objects can only be requested by their hashes. They have no other names or identifiers.
3. Data objects are always UTF-8 strings, which makes it possible to write them into terminals. You are allowed to set an arbitrary string length limit, such as 255 bytes.
4. Data objects are not saved to disk, which means that they disappear if you terminate the nodes that hold copies of them.
5. No communication is encrypted.
6. All network nodes have access to all stored data objects without needing any permissions.

You are allowed to ignore any of these delimitations if you like, but be aware that it may complicate your implementation significantly. You are not guaranteed a higher grade on the lab for making your implementation more complete, but may consider the learning experience worth the effort.

Method

You are expected to work loosely according to the principles of ASD, as we already meantioned in the Introduction. This means that you need to break down and prioritize the objectives you want to complete and organize them into so-called *sprints*. As each sprint will end with you presenting your progress to the teaching assistants, you may not decide the length of each sprint yourselves. Make sure to look in Canvas for when each sprint ends.

The Lab Report

While working, you are required to record your plans, progress and design decisions in a lab report, which you will present to the teaching assistants at the end of each sprint. It is not expected to be completed until the last sprint review. The report must contain the following:

1. The group number and the names of the two or three members of your group.
2. A link to your code repository, which could reside on e.g. GitHub.[16]
3. A list of the frameworks and other tools you are using.
4. A system architecture description that also contains an implementation overview.
5. A description of your system's limitations and what possibilities exist for improvements.
6. A separate section for each sprint that contains (a) a sprint plan, (b) a backlog and (c) other reflections.

[16] See https://github.com/.

Assessments

During each sprint, you will need to sign up for a sprint review at the end of the sprint. Information about signing up will be announced via Canvas. You are expected to be able to demonstrate the following at the sprint reviews:

Sprint 0

(a) A working understanding of the Kademlia algorithm. The members of your group will be selected at random to answer some questions we prepared.

(b) You must be able to spin up a network of at least 50 containerized nodes via a script or some orchestration software, as well as showing that any network member can send a message, of any kind, to any other. The nodes do not have to carry any Kademlia-related software at this point.

(c) A plan for how you will organize your work the coming sprint.

(d) A lab report, which you submitted before the deadline announced via Canvas. The report must at least account for your plans for sprint 1.

Sprint 1

(a) Objectives M1 to M5, if you plan on a higher grade. Otherwise, you need to be able to demonstrate how far you come on completing these objectives.

(b) What you did during the sprint and a plan for how you will organize your work the coming sprint.

(c) A lab report, which you submitted before the deadline announced via Canvas. The report must at least account for your plans and progress.

Sprint 2

(a) Objectives M1 to M6 and as many of the qualifying objectives as you like.

(b) What you did during the sprint.

(c) A lab report, which you submitted before the deadline announced via Canvas. The report must be complete at this point.

Note that your performance on sprint reviews 0 and 1 have no bearing on your final grade on the lab. The expectations are set to help you divide your work evenly across the duration of the course.

Appendices

A Unit Testing

The file routingtable_test.go contains an incomplete unit test of the routing table. 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 KademliaID, an IP address and port number. The KademliaID 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("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 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.