D7024E Lab assignment – Grand Repository in the Sky

The main purpose of this assignment is to build a proof-of-concept prototype of a working distributed system. You are going to learn about the underlying technologies that are used under the hood in modern cloud services on the Internet today.

In short, you are going to develop your own distributed cloud storage system. The idea is quite simple; anyone can set up a server and provide storage resource to the "Sky" network. Stored data should be distributed and spread out among all servers so that data cannot be lost even if some servers disconnect from the Sky network. Finally, the system should scale with millions of servers.

To fulfill this grand mission, a large part of the lab will focus on implementing a Distributed Hash Table (DHT) called Chord[1]. In short, a DHT is basically a distributed lookup protocol designed to locate a node in a network of nodes where a particular data object is stored. Finally, you are going to play around with virtualization/cloud technologies such as Docker and Kubernetes to set up a cluster of nodes.

Working methods

There are a few guidelines for the lab assignment:

The lab should be done in groups of two 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.) that you have used. The assignment is generally defined, so we expect all solutions to be different to some extent.

- The lab should preferable be implement in Golang. However, if you decide to use another language, you are own your own and get not 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 5 objectives. Before starting a new objective, you should mail a sprint planning to the lab teacher (johan.kristiansson@ltu.se). To pass the lab assignment you should make a demonstration for Johan Kristiansson by the end of the course when you are done.

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

[1] http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

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.

Mail the report to johan.kristiansson@ltu.se when you are done. 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. It is not required that the code is thread safe.

Objective 1 – Chord DHT algorithm playground

The purpose of the first objective is to get you familiar with the Chord algorithm. The first step is to read the Chord paper to figure out it works. The second step is to learn Go. The final step is to develop a simple simulator and a first version of the Chord DHT algorithm to play around with and to learn more how it works.

To save some time, you will get some basic code. See Appendix A.

Task 1: Organize nodes in a ring

To this this you need to develop a lookup function. You also should develop a debug function to print the ring on the screen. Note that all RPC function can be implemented as simple function calls in the same program. The lookup function can thus be implemented using recursive/iteration, similar to traversing a linked list. Don't worry; you will add network support later.

In this first experiment, you should use 3 bits in the key/id identifier space, which means that we can maximum 8 nodes in the Chord ring. When you got it working, then switch to 160 bits as used by the SHA-1 hash algorithm.

node0 := makeDHTNode("00")
node1 := makeDHTNode("01")
node2 := makeDHTNode("02")
node3 := makeDHTNode("03")
node4 := makeDHTNode("04")
node5 := makeDHTNode("05")
node6 := makeDHTNode("06")
node7 := makeDHTNode("07")

node0.addToRing(node1)
node1.addToRing(node2)
node1.addToRing(node3)
node1.addToRing(node4)
node4.addToRing(node5)
node3.addToRing(node6)
node3.addToRing(node7)

Calling **node1.printRing()** should return the following node IDs.
01
02
03
04
05
06
07
00

nodeXX.lookup("03") should return "03". Since we don't have finger tables yet, we have to linearly traverse the ring until we found a node that is "responsible" for the key.

Tip: The **between(id1, id2, key []byte) bool** function found in Appendix A can be very useful to implement the lookup function, i.e. to find out if a node is responsible for a specific key. Also, use the **generateNodeId() string** to calculate node ids.

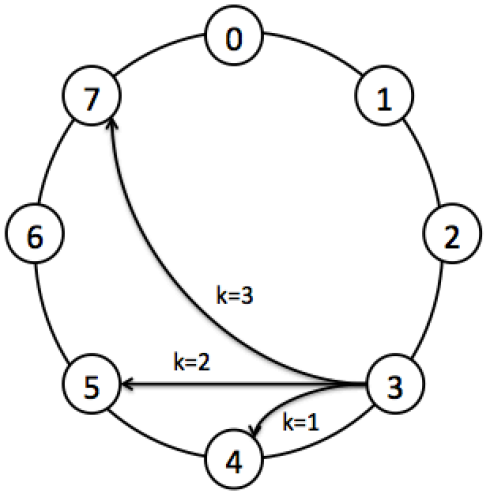
Task 2: Calculate finger tables
Each node should have as many fingers as in the key/id identifier space. 3 bits means 3 fingers, 160 bits means 160 fingers. Each finger should be calculated as (n + 2^(k-1)) mod (2^m), where n is the node id, k finger index (e.g. 1,2,3), and m number of bits. To save some time, use the **calcFinger(n []byte, k int, m int) (string, []byte**) found in Appendix A.

Note that the fingers must be recalculate (stabilization) at regular interval, in case nodes leaves or joins the ring.

Calling **node3.printFinger(1, 3)** // k=1 m=3 bits
should return (or something similar):

n	3
k	1
m	3
2^(k-1)	1
(n+2^(k-1))	4
2^m	8
result	4
successor	04

Note that the successor of a finger can be calculated by simply calling **node3.lookup(finger).**



All fingers can now be calculated by calling **node3.printFinger(1, 3)**, **node3.printFinger(2, 3)**, and **node3.printFinger(3, 3).** Then the ring and the figure tables for node 3 should look as in the figure depicted above.

Improve the previously develop lookup function to take advantage of the finger tables, i.e. instead of sending a message clockwise in the ring, we can use the fingers as shortcuts. The **distance(a, b []byte, bits int) *big.Int** function can be very useful to figure out which finger to forward a lookup request to.

Now, repeat the experiment using 160 bits instead of only 3 bits. We can now have maximum of 2^160 nodes instead of just 8 nodes. 2^160 is a really big number btw.

Objective 2 – Fully working Chord DHT algorithm

By now you should fully understand the Chord algorithm. It is time to actually implement the algorithm for real. Instead of running the Chord ring within the same program using function calls, each Chord node should now run as a separate server and communicate using a network protocol, for example UDP. All the Chord nodes together will form the Sky network.

This is pseudo code of a network code that could save you some time. Note that implementing a parser is simple in Go. See the code below.

```
type Msg struct {
        Key     string
        Src     string
        Dst     string
        …
}

type Transport struct {}

func (transport *Transport) listen() {
        udpAddr, err := net.ResolveUDPAddr("udp", transport.bindAddress)
        conn, err := net.ListenUDP("udp", udpAddr)
        defer conn.Close()
```

```
        dec := json.NewDecoder(conn)
        for {
                msg := Msg{}
                err = dec.Decode(&msg)
                // we got a message
                …
        }
}

func (transport *Transport) send(msg *Msg) {
        udpAddr, err := net.ResolveUDPAddr("udp", dhtMsg.Dst)

        conn, err := net.DialUDP("udp", nil, udpAddr)
        defer conn.Close()

        _, err = conn.Write(msg.Bytes())
}
```

It is up to you to divide this objective into suitable tasks. Don't forget to send the sprint planning to lab teachers.

NOTE: Use channels when communicating between Go routines.

Objective 3 – Replication

By now, you have a fully working Chord network (the Sky network). It is time to actually store some data in it. However, to solve this problem you need to figure out to mange data when new Chord node joins and leaves the network. Remember no data should be lost! And data should be spread among all the Chord nodes.

Use the previously developed lookup function to find out where data should be stored (i.e. which node) and develop a replication protocol. Note that the Chord algorithm (consistent hashing) itself will spread stored data evenly distributed among all available Chord nodes, but you need to develop an algorithm that allows any Chord nodes to leave the ring without loosing data. You cannot assume that a Chord node will gracefully leave the ring; rather you should assume that any Chord node could crash at any time.

One strategy to handle this problem is to replicate data to a node's closest neighbor. In this case, the closest neighbor could just take over responsibility if a specific Chord node is not responding and becomes unavailable. However, you need to come up with a model to handle data consistency. For example, if a Chord node temporary becomes unavailable and is replaced by another Chord node, the system should eventually converge to the same state even if the old Chord comes back online again. Note that different clients may have a different view of the system and may still use the old Chord node even if has been replaced and is not used by other clients. What is the cost of re-balancing the ring?

Objective 4 – Sky web service

Develop web service that uses the Sky network (i.e. the previously develop Chord network). To implement such a service, you need to add a HTTP server to each Chord node. For example)

HTTP POST http://CHORDNODE/storage
Upload a new key-value pair. The post request should contain both the key and the value.
HTTP GET http://CHORDNODE/storage/KEY
Returns the value for a specific key (KEY)
HTTP PUT http://CHORDNODE/storage/KEY
Update the value for a specific key (KEY)
HTTP DELETE http://CHORDNODE/storage/KEY
Delete a key-value pair with key (KEY).

You should also develop a test application to demonstrate the web service. The test application can be fairly simple, for example just setting key and values in a HTML form. It is also ok to develop a CLI-based test client.

Important: Don't spend time on developing a fancy user interface, as that will not improve the grade. This is a course in distributed and mobile systems and not in human computer interaction.

Also note: It should not matter which Chord node the client connects to as the data is spread out and stored in the entire Chord network.

Objective 5 – Cloudification

This objective is about learning virtualization and cloud technologies. More specifically, you are going to learn how to use Docker[2] and Kubernetes[3].

Docker is an open source project to pack, ship, and run application in a lightweight container. It does not use a hypervisor based virtualization technology as used in OpenStack, but a more lightweight virtualization technology called Linux Container (LXC).

Kubernetes is an open source orchestration tools for managing Docker containers. It provides support to schedule containers across multiple hosts, self-healing, and features to upgrade and scale containers.

In this objective you should package the previously developed Skynodes as a Docker containers and use Kubernetes to deploy clusters of Skynodes.

For example,
$ kubectl scale rc skynode --replicas=2

Should result in two Skynode pods being deployed.

---

[2] http://www.docker.com/
[3] http://www.kubernetes.io/

```
$ kubectl get pods -l="name=skynode"
NAME         READY   STATUS   RESTARTS   AGE
skynode-1    1/1     Running  0          2s
skynode-2    1/1     Running  0          2s
```

By increasing the replicas count, you should be able to increased number of running Skynodes running.

Next step is to develop a Chaos Monkey[4] that randomly kills sky nodes. Configure Kubernetes to restart crashed nodes. Note that data should never be lost as the data is replicated across the Chord ring. How many nodes (churn rate) can crash before data is lost?

Make a demonstration showing how the Chaos monkey kills nodes and that data is never lost.

[4] https://en.wikipedia.org/wiki/Chaos_Monkey

Appendix

```
package dht

import (
	"bytes"
	"crypto/sha1"
	"fmt"
	"github.com/nu7hatch/gouuid"
	"math/big"
)

func distance(a, b []byte, bits int) *big.Int {
	var ring big.Int
	ring.Exp(big.NewInt(2), big.NewInt(int64(bits)), nil)

	var a_int, b_int big.Int
	(&a_int).SetBytes(a)
	(&b_int).SetBytes(b)

	var dist big.Int
	(&dist).Sub(&b_int, &a_int)

	(&dist).Mod(&dist, &ring)
	return &dist
}

func between(id1, id2, key []byte) bool {
	// 0 if a==b, -1 if a < b, and +1 if a > b

	if bytes.Compare(key, id1) == 0 { // key == id1
		return true
	}

	if bytes.Compare(id2, id1) == 1 { // id2 > id1
		if bytes.Compare(key, id2) == -1 && bytes.Compare(key, id1) == 1 { // key < id2 && key > id1
			return true
		} else {
			return false
		}
	} else { // id1 > id2
		if bytes.Compare(key, id1) == 1 || bytes.Compare(key, id2) == -1 { // key > id1 || key < id2
			return true
		} else {
			return false
		}
	}
}

// (n + 2^(k-1)) mod (2^m)
func calcFinger(n []byte, k int, m int) (string, []byte) {
	fmt.Println("calulcating result = (n+2^(k-1)) mod (2^m)")

	// convert the n to a bigint
	nBigInt := big.Int{}
	nBigInt.SetBytes(n)

	fmt.Printf("n	%s\n", nBigInt.String())
```

```
	fmt.Printf("k          %d\n", k)

	fmt.Printf("m          %d\n", m)

	// get the right addend, i.e. 2^(k-1)
	two := big.NewInt(2)
	addend := big.Int{}
	addend.Exp(two, big.NewInt(int64(k-1)), nil)

	fmt.Printf("2^(k-1)     %s\n", addend.String())

	// calculate sum
	sum := big.Int{}
	sum.Add(&nBigInt, &addend)

	fmt.Printf("(n+2^(k-1))  %s\n", sum.String())

	// calculate 2^m
	ceil := big.Int{}
	ceil.Exp(two, big.NewInt(int64(m)), nil)

	fmt.Printf("2^m         %s\n", ceil.String())

	// apply the mod
	result := big.Int{}
	result.Mod(&sum, &ceil)

	fmt.Printf("result      %s\n", result.String())

	resultBytes := result.Bytes()
	resultHex := fmt.Sprintf("%x", resultBytes)

	fmt.Printf("result (hex) %s\n", resultHex)

	return resultHex, resultBytes
}

func generateNodeId() string {
	u, err := uuid.NewV4()
	if err != nil {
		panic(err)
	}

	// calculate sha-1 hash
	hasher := sha1.New()
	hasher.Write([]byte(u.String()))

	return fmt.Sprintf("%x", hasher.Sum(nil))
}
```