SPRINT 2 – LAB GROUP 1

- Ludovic Peter – ludpet-7
- Thomas Rochette – thoroc-7
- Dolores Raigada Romero – dolrai-7

Mobile and Distributed Computing Systems

INDEX

1. Our kademlia Implementation

Kademlia is a peer-to-peer distributed hash table. Its algorithm is compare to a binary tree where the leaves are nodes and each of them is identified by a specific ID. Kademlia protocol ensures that every node knows of at least one node in each of its subtrees, if that subtrees contains a node. This way, any node can locate another one by knowing its ID.
We had used this algorithm to implement a peer to peer file storing system. We have made some choices in our implementation, first we store directly in the kademlia hash table the file stored in the network. It provides a good replication of the data in the network. We also made the choice to delete a file if the administrators (the node who publishes the file) stop to send messages to keep data up to date (it could be avoid by pinning the file, see the pin section for more detail).

2. Use cases

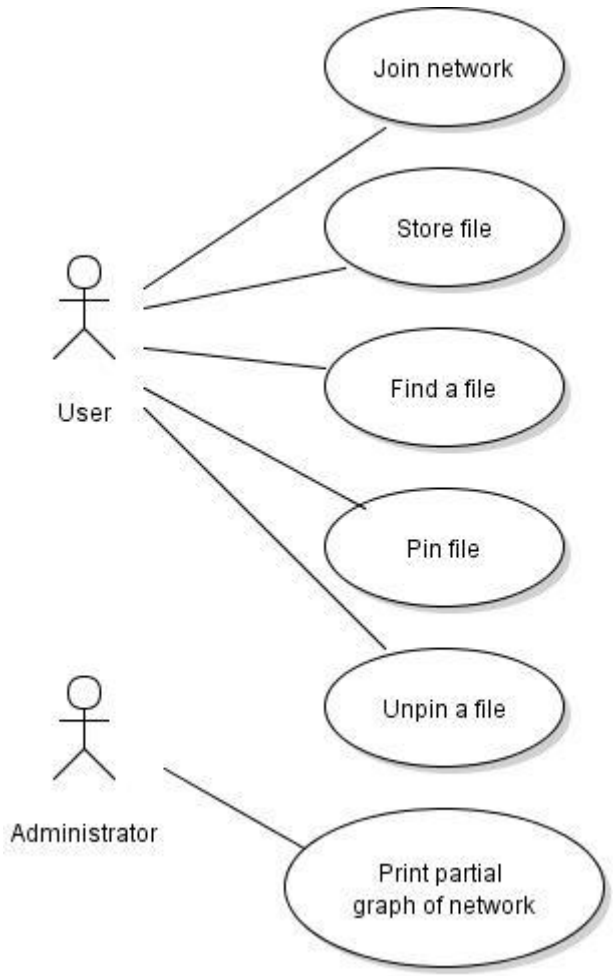

Figure 1: Use cases of Kademlia Assignment

3. Requirements

It is needed to have a Golang compiler on your computer.

4. Assumptions

It's a not byzantine resilient system, so a node cannot cheat.
A node or the network could have failure, the system is resilient.

5. Frameworks and tools used

- Json instead of Protobuf
- Visual Studio Code
- Atom
- Golang
- Github
- Graphviz
- Violet UML

6. System architecture description and an overview of the implementation

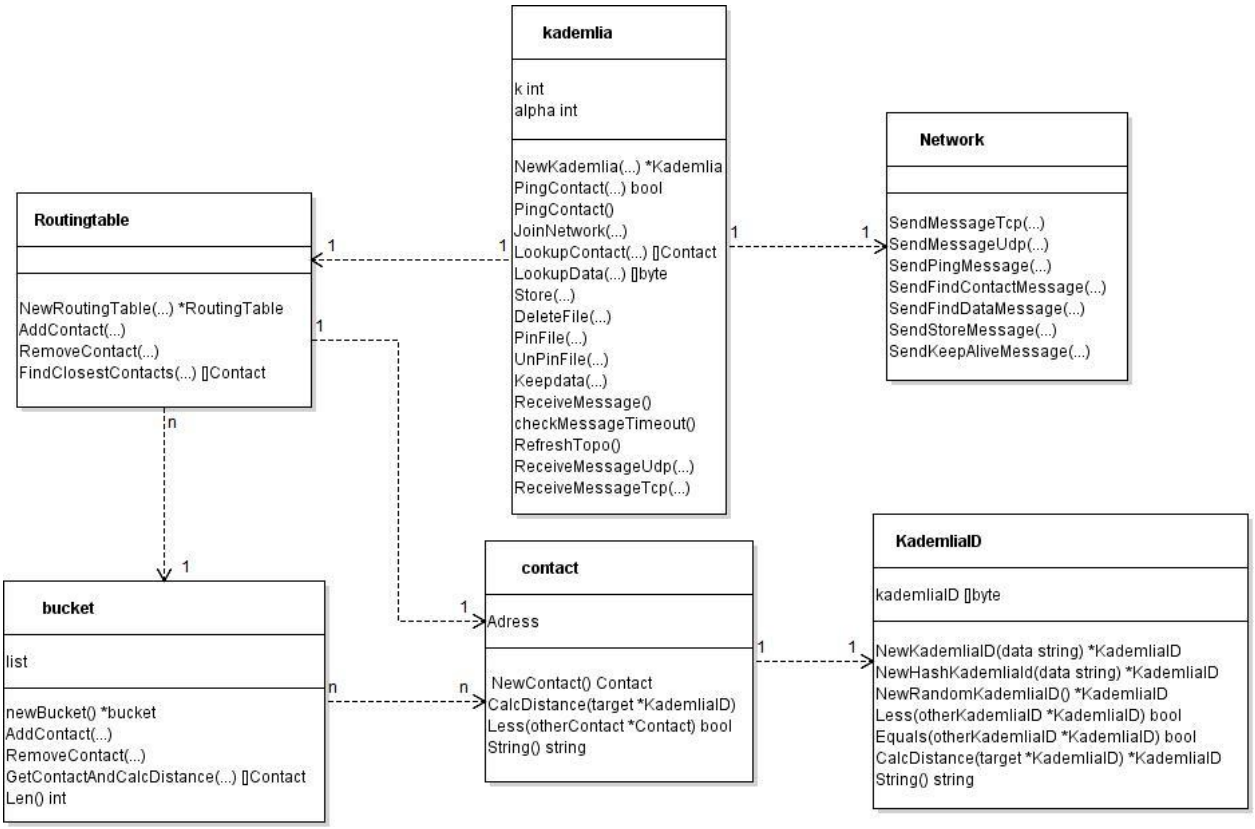

Figure 2: System architecture UML diagram

The Kademlia file contents all the functions needed for use cases. It communicates with other Kademlia class in another node via the network function. We use udp/tcp communication and Json marshalling to communication between nodes. The routing table file provides simple task to explore known nodes (for instance to find known closest nodes). Bucket file contents a simple list of known contacts and simple functions to explore it and add new nodes. Contact file contents information about a node, and methods to manipulate this.
The main function provides simple CLI interface, you can obtain a list of available commands by writing help. There are two modes on this CLI; one hypervisor node (it allows to manage many nodes on the same interface), and a standard client mode (it manages a simple node).

7. Your threading model:

The threading model used in this assignment is free threaded. By this way, methods can run in parallel without having to wait one for another. Our threating model is based on creating a go routine for each call to functions that need to run parallelly, simply by doing "go funct()" where funct() is the name of the function required.

Each node in the network use different threads to run. There are for each client node 6 threads with different goals for each:

- First there are two server threads (one for udp communication, another for tcp). They provide responses to each request received by the node and also treat the response sent to you by other node.

- In addition of this 2 server threads, there is also 1 client thread in charge of your CLI and of sending your request to the network.

- The 3 other threads are kind of scheduler, they wake up regularly to doing some specific tasks. The first one checks if stored data is still up to date, the second one checks if some node doesn't respond to request. If not, it sends a ping to check the availability of the node. The third one, is used to update the topology of the network. It just sends a ping to each node of its routing table (if a node doesn't respond, it will be handled by the previous thread.

In addition of this standard node configuration, you could have some thread running to keep your file available on the network (One thread for each file under your responsibility).

8. Pin and Unpin system:

In this part we will detail the store, pin, unpin and delete functions and the timeout and keepalive mechanisms which are all related. In Kademlia the node which wants to store a file needs to keep this file alive. In this purpose it sends messages to the k nodes that stored the file a few time before the file timeout so if there are lost messages the file will not timeout immediately. If a file is really important and we want to keep it, we can pin it and unpin it if needed. We can also delete a file from the network if it is not pinned.

Technically all the files stored by a node are kept in a map. We store a pointer to the structure with the title as the key, the fields Data, PinStatus, On, LastStoreMessage, ChangedDetected. The data field contains obviously the data of the file, the PinStatus field is a Boolean changed by the functions pin and unpin, On field is a Boolean changed by the delete function to know if there has been a request to delete the file, and the ChangedDetected is a Boolean to check if the Pinstatus or the On field have been modified.

We tested two version of the function store. The first one sends all the fields every time. We keep the second one that does not send the data which is better for big files (the file is sending on demand by the node who received the store message if it's needed). Every time a node stores a new file, there is a new thread that keeps the file alive. This thread will send the keep alive message every ¾ of the file timeout so if there is delay before the message is received the file won't be deleted, but if there are changes in the PinStatus or in the On fields the "keepalive" message is send quicker in order to have up-to-date data.

Pin and unpin functions are really simple ones which just change the PinStatus Boolean. For the delete function it is the same but we first check that the file is not pinned before changing the On Boolean. Those are the function used by the node that wants to keep a file, now every node run a thread keepdata to check the timeout of the files they store.

The nodes have another map with all the files they store, as the other map there are all the fields of a file structure.

In the thread keepdata there is a loop that run while the node is on, it will wait during a timeout time plus a random time between 0 and 30 seconds in order to desynchronized the nodes (it is not the best method because we can't be sure that the nodes are desynchronized but it was the easiest solution) and check if all the files it stores are not time out or deleted. First if they are deleted and not pinned we just delete the file. If the file is not delete we look if the file has timeout, if yes we need to look if it is pinned. If it is, the node will check if it is still in the k closest nodes, if it is not anymore it will delete the file. If it is in the closest nodes, it starts a new store thread so become the "owner" of the file. Of course, if the file is not pinned and timeout, the node deletes it.

9. Theoretical problems with Kademlia:

They are two major problems with kademlia algorithm: the congestion of the network and the rapid change of the topology.

- If the network increases in size, it could cause network congestion and UDP packets could fail to be delivering on time. This will induce more and less severe issues because nodes cannot communicate. A possible solution to avoid this issue is to have a well-balanced network to share work and communication between all the nodes. It could also be limited by a better protocol of communication (for instance we had choose to send file only on the demand to reduce data flow).
- The other problem occurs if a lot of nodes join or leave the network on the same time. It could have some issues like a file stored on any of the k closest nodes (because all are new) or a lot of dead nodes on the routing table, polluting the lookup. It could be resolved by a system to resend store message regularly and some mechanism to maintain the topology up to date (in our app we choose to regularly ping all node to check their availability.

10. Link to code

https://github.com/lpeter68/mobile-and-distributed.git

11. Backlog

Task is ordered in each domain by priority and order of realization.

TASKS	Not started	In progress	Done
Network:			
Have udp asynchronous communication			
Have tcp transmition for sending file			
(optional) Have a mechanism to limit data transmission (sending file on demand)			
(optional) Communicate through NAT			
Kademlia:			
Have Lookup for an ID			
Be able to send ping and update routing table			
Be able to store data on the network			
Have lookup for data			
Be able for a node to join the network			
Have a mechanism to delete crashed nodes from the routing table			
File Storing:			
send/store a file			
Find a file on the network			
Have mechanism for delete deprecated file			
Have republish system to keep alive data			

Have pin and unpin system			
(optional) Preserve data from a node crash (write data on file or database)			
Interface objective:			
(optional) Have an CLI to managed a simulation with many nodes			
Have a CLI client to use the kademlia network			
(optional) Made a simple graphical interface			
Quality objective:			
Implemented basic unit test			
Do more elaborated test (with many a lot of nodes, nodes crash, …)			
Check thread safety			

12. References

- Kademlia paper
- Course slides
- Golang by Gyga Code
- Go by example