

Homework 2: Routing and File Sharing

CS438 - Decentralized Systems Engineering 2019

Homework out: Friday, 18.10.2019

Due date: Friday, 23:59, 1.11.2019

Introduction

In this homework you will enhance Peerster with the following features:

1. A simple routing protocol to enable Peerster instances to send each other unicast, point-to-point messages, rather than having to broadcast everything through gossip.
2. A simple file-sharing protocol to enable nodes to copy files between each other.

This homework builds on homework 1. If you decide not to use your own homework 1, but instead use one of the three homeworks you received for review, please specify in the “Online text field”, upon homework 2 submission, which of the three reviews you’re using as follows: “Built on review assignment, ID x”, where x is 0, 1 or 2. If you use your own assignment 1, please specify “Built on own assignment”.

Using any other sources besides your own homework 1 or one of the three reviews you received is considered plagiarism and will receive 0 points.

Part 1: Routing

To give Peerster nodes point-to-point communication ability, you will first implement a simple *destination-sequenced distance vector* (DSDV) routing scheme. This scheme has proven popular for routing in ad hoc mobile networks, due to its combination of simplicity and robustness against routing loops. You should first familiarize yourself with the general scheme via appropriate background readings, such as:

- [Wikipedia](#) (very brief summary)
- [Original DSDV paper](#) by Perkins and Bhagwat
- Background: [Tanenbaum “Computer Networks”](#) 5.2 Routing Algorithms; [Coulouris](#) 3.3.5 Routing.

In this scheme, each node maintains a table of destinations and, for each destination, a “next hop” to reach that destination. Peerster will piggyback its routing scheme on its gossip protocol: each rumor message will double as a route announcement message, and the sequence numbering scheme you already implemented for gossip purposes will act as the “destination sequence numbers” that the DSDV routing scheme requires.

Exercise 1

Change your Peerster implementation to build and maintain a **next-hop routing table**: a key/value dictionary (e.g., a map) where keys are `Origin` identifiers and values are `(IP`

address, port number) pairs (e.g., map[string]string). Whenever a Peerster node receives **the first Rumor message from Origin**, record it in your next-hop table, at the key corresponding to the message's Origin, the IP address and port number from which that rumor message arrived. The tuple (IP address, port number) will be the next-hop on your route to Origin, which will remain in effect until you receive the next rumor message (**with a higher sequence number**) from the same node Origin. Of course, you still need to forward the message to a random neighbor whenever you receive a Rumor message, just as in Homework 1.

For example, at the beginning, Dave sends a Rumor message to Bob. We assume that Dave's IP is "1.2.3.4", his port is "43433" and ID is 1. Thus, Bob updates his DSDV routing table with item ["Dave": "1.2.3.4:43433"]. Then, Bob forwards that Rumor message to Alice. After receiving the message, Alice also updates her DSDV routing table with item ["Dave": "5.6.7.8:33333"], where "5.6.7.8" is Bob's IP address and "33333" is Bob's port. If Alice wants to send message to Dave, she just needs to send message to "5.6.7.8" (with port 33333), although she does not know the concrete IP address of Dave. After several hours, if Eve sends another Rumor message to Alice through Bob, both Alice and Bob will add new items in their routing tables. Specifically, a new item ["Eve": "5.6.7.8:33333"] will be added to Alice's routing table. (Of course, Bob will also update his routing table.) Please note that if Dave sends a new Rumor message with ID 3 to Alice through Jack, Alice will have a new item ["Dave": "3.3.3.3:55555"] instead of the old one (i.e., ["Dave": "5.6.7.8:33333"]). We assume "3.3.3.3" is Jack's IP and his port is "55555".

When updating a DSDV routing table entry (e.g. for peer "Dave" updated to ["Dave": "3.3.3.3:55555"]), your program should print out the following to stdout:

```
DSDV <peer_name> <ip:port>
```

where <ip:port> is the new next-hop for <peer_name>. For instance :

```
DSDV Dave 3.3.3.3:55555
```

Exercise 2

Route Rumors

Unfortunately, if Peerster nodes only ever “announce” themselves to the network when the local user actually types in a message, nodes whose users are inactive for extended periods will never be announced in the network, and thus other nodes won't be able to find a route to them. We'll fix this by ensuring that nodes always send Rumor messages occasionally, merely for the purpose of updating other nodes' routing tables, even when the local user is idle.

Add a periodic timer, that generates a *route rumor* message. A route rumor message is just like the “chat rumor” messages you already generate, i.e., also a RumorMessage type, except it contains only the Origin and ID fields with an empty Text field. Modify your

Peerster's implementation message handler to accept and forward route rumors as well as chat rumors. *The processing should be essentially the same, except that you DO NOT display a DSDV message to the user when the rumor message contains an empty Text field.*

```
rm := RumorMessage{
    Origin: "Alice",
    ID: 23,
    Text: "",
}
```

For testing purposes, add a `rtimer` flag that indicates how many seconds the peer waits between two route rumor messages. Put 0 seconds as the default value for `rtimer`. A value of 0 should disable the sending of all route rumors (including the one sent at startup).

Start-up Route Rumor

Besides the periodic route rumors, specified through the `rtimer` flag, a Peerster node should also **generate a single route rumor message when it first starts up** (either sent to a random neighbor or broadcast to all neighbors; your choice) to “prime the pump” and get itself known to other nodes quickly. Test your code to ensure that a routing table entry appears “quickly” in other Peerster instances after startup without having to manually type any chat messages.

Usage of gossipper:

- UIPort string
port for the UI client (default "8080")
- gossipAddr string
ip:port for the gossipper (default "127.0.0.1:5000")
- name string
name of the gossipper
- peers string
comma separated list of peers of the form ip:port
- simple
run gossipper in simple broadcast mode
- antiEntropy int
Use the given timeout in seconds for anti-entropy. If the flag is absent, the default anti-entropy duration is 10 seconds.
- rtimer int
Timeout in seconds to send route rumors. 0 (default) means disable sending route rumors.

This routing table will enable you to send point-to-point messages in Exercise 3.

Important: Route rumors and start-up route rumors **DO NOT** output DSDV messages.

Exercise 3

Private Message

Add a GUI mechanism to display a list of node `Origin` identifiers known to this node—i.e., for whom a next-hop route is available. This node list might be located right beside the main chat text-display box, for example. The user should be able to select a node from this list (e.g., double-click, or select an item then press a separate button) to open a “private message” dialog box, into which the user enters a private message to send to that node.

Add a CLI option `-dest` that defaults to the empty string. If `-dest` is filled along with a `-msg` option, your program should send a private message to the destination (**and no rumor**).

Usage of client:

- UIPort string
port for the UI client (default “8080”)
- dest string
destination for the private message; **can be omitted**
- msg string
message to be sent; if the `-dest` flag is present, this is a private message, otherwise it’s a rumor message

The client uses the following structure to communicate with the gossipier. The Last two fields are explained in the following sections of this assignment.

```
Message struct {
    Text string
    Destination *string
    File *string
    Request *[]byte
}
```

When the gossipier receives a private message from a local client, it writes at stdout:

CLIENT MESSAGE <msg_text> dest <dst_name>

You’re **not required** to implement correct sequencing of private messages between pairs of hosts - *which means there’s no need to use vector clocks*. You can of course implement it if you want to, but keep in mind that our DEDIS peer will not implement it. For compatibility, we set the `ID` field to 0 to signal “*no order imposed*” for private messages.

A private message should be formatted as a structure containing five fields:

```
type PrivateMessage struct {
    Origin      string
    ID          uint32
    Text        string
    Destination string
}
```

```

        HopLimit    uint32
    }

```

- an `Origin` (`string`) containing the identifier of your node
- an `ID` (`uint32`), set to 0 to denote no sequencing
- a `Text` (`string`) containing the text of the private message (a string)
- a `Dest` (`string`) key whose value is the destination node identifier (corresponding to the `Origin` of a prior rumor message)
- a `HopLimit` (`uint32`), which you initialize with a constant default value, e.g., 10. Every node on the forwarding path (including the **source peer**) will decrement the `HopLimit`, discarding the message if the value reaches 0 before the message reaches the destination. Although routing loops should *normally* not arise in a DSDV protocol such as this, bugs or malicious behavior could still create loops, and the `HopLimit` protects against this risk.

```

type GossipPacket struct {
    Simple  *SimpleMessage
    Rumor   *RumorMessage
    Status  *StatusPacket
    Private *PrivateMessage
}

```

For example:

```

pm := PrivateMessage {
    Origin : "Alice",
    ID : 0,
    Text : "hello"
    Destination: "Bob",
    HopLimit: 10,
}

```

Finally, update your Peerster implementation to accept private messages. Every node on the routing path, **including the Origin**, processes the packet and:

- **If the `Destination` field refers to another node, first check the `HopLimit`. If the `HopLimit == 0` then the node DOES NOT send the message out and the processing stops here. Otherwise, if the `HopLimit > 0`, decrement the `HopLimit` just before forwarding the packet to the next hop.**
- **If the `Destination` field indicates that the message is for the local node, you need to write at stdout:**

```
PRIVATE origin <origin> hop-limit <hop-limit> contents <contents>
```

Test your code by creating indirectly-connected chains of Peerster instances on your localhost.

Important! Do **NOT** display route rumor messages in the GUI.

Part 2: File Sharing

We will now give Peerster nodes the ability to send and receive potentially large files, not just short text messages. Since Peerster uses UDP for everything and UDP works reliably only with short (e.g., up to 8KB) datagrams, we will have to break files up into chunks for transfer. Nodes will also need to be able to announce files to each other and ask to download them. As in many P2P systems, we will use hash trees to identify both complete files and parts of files ([Wikipedia page](#)). **To compute hashes, please use the sha256 hash function in the Go `crypto` package.**

Exercise 4

File Selection

To share files, the first thing you will need to do is to allow the user to specify a file to share, to index and divide that file into chunks.

Add a "Share File..." option (e.g., button) to the UI, which opens a file selection dialog box that allows the user to select a file to share. *If a user wants to share multiple files, they simply share them one at a time.*

Also, add a flag `-file` to your command-line client so you can index and share a file with a command like the following:

```
./client -UIPort=8080 -file=flyingDrone.gif
```

Usage of client:

- UIPort string
port for the UI client (default "8080")
- file string
file to be indexed by the gossipier

File Paths

To avoid working with absolute paths, create a directory called `_SharedFiles` **in the same directory as your gossipier executable** and put the files that your Peerster is going to share in this directory. This means, given the command above, the absolute path of the file `flyingDrone.gif` should be `$CWD/_SharedFiles/flyingDrone.gif`, where `$CWD` is the absolute path of the executable **gossipier**.

Exercise 5

File Indexing

Now, for each file the user chooses to share via the GUI dialog above, you will need to scan the file, divide it into chunks, and compute a SHA-256 hash on the contents of each chunk. For simplicity we use a **fixed chunk size of 8KB**.

Important! When you are breaking files into 8KB chunks, in most cases the file will of course not be a natural multiple of 8KB in size; in this case your last chunk will be less than 8KB in size. **Do NOT pad** the last chunk to 8KB, since if you do so, you will "lose" the correct original size of the file.

As a result of scanning each file you'll need to build up a **metafile**, which is simply a file containing the SHA-256 hashes of each chunk. Simply concatenate the 32-byte SHA-256 hash of each chunk, in the order `chunk_1`, `chunk_2`, ..., `chunk_N` into one large `[32×N]byte` slice and write this slice to the binary metafile. Once you have this metafile, compute its SHA-256 hash (named `MetaHash`) and index it in your gossip.

After scanning a file to be shared, your Peerster should have the following metadata in its internal data structures, for each file:

- File name on the local machine.
- File size in bytes.
- The metafile computed as described above.
- The SHA-256 hash of the metafile.

Important! The SHA-256 `MetaHash` (hash of the metafile) is the only unique identifier you will have for a file.

Exercise 6

File Share and Download

Finally, we need a protocol for one node to retrieve a file from another node. In this homework we assume the retrieving node already has the `MetaHash` of the desired file (obtaining the `MetaHash` is part of Homework 3). We will use a simple one-chunk-at-a-time request / response download protocol.

Add support for two new messages to your protocol: 1) for chunk and metafile requests, as well as 2) for chunk and metafile replies.

- ```
type DataRequest struct {
 Origin string
 Destination string
```

```

 HopLimit uint32
 HashValue []byte
 }

 • type DataReply struct {
 Origin string
 Destination string
 HopLimit uint32
 HashValue []byte
 Data []byte
 }

```

**HashValue** represents:

- either the hash of the requested chunk
- or the MetaHash, if the request is for a metafile

**Data** is the actual data (metafile or a chunk) that a node replies to a request with.

```

 • type GossipPacket struct {
 Simple *SimpleMessage
 Rumor *RumorMessage
 Status *StatusPacket
 Private *PrivateMessage
 DataRequest *DataRequest
 DataReply *DataReply
 }

```

These messages are similar to a `PrivateMessage` with additional fields for requesting and delivering chunks and metafiles. Peerster nodes must route these messages “point-to-point” exactly as described in Part 1, so that a node can download a file from another node that is connected either directly or only indirectly via intermediate hops. Note that these new messages both have `Destination` and `HopLimit` fields, just like point-to-point text messages, which are the **only fields** the routing logic needs to care about. Thus, your code can decide before doing any local processing, simply by looking at these fields (`Destination` and `HopLimit`), whether to route a message on to another node or to process it at the local node.

**Important!** Any file download starts by first requesting the metafile. Thus, any peer who is downloading a file has the corresponding metafile, even when the file download did not complete yet (not all chunks have been downloaded).

**Important!** For a downloaded chunk / metafile to be valid, the SHA-256 hash of the chunk / metafile must always match the explicit `HashValue` carried in the message's `DataReply` field. *Your code must check this invariant when it receives a chunk reply and drop the message if the message contains incorrect data.*



**Important!** Shared files are not bigger than 2 MiB, which means there's no need to chunk the metafile, as it's smaller than 8 KiB.

**When a gossipier receives a DataRequest for a metafile or a chunk that the gossipier does not have, the gossipier replies with a DataReply containing an empty Data field.**

## GUI

Also add a simple GUI button and command-line flags to download a file from another node. To start with, you can simply have the user enter a target `Dest` node and a *hexadecimal* `MetaHash` (passed with the `request` flag), which serves as a unique file ID. The example of the command can be:

```
./client -UIPort=8080 -dest=anotherPeer -file=flyingDrone.gif
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318
```

where you specify you would like to download the file with the metafile hash `ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318` from the peer `anotherPeer`. The file name that you specify with the `-file` flag (i.e., `flyingDrone.gif` in this example) is the name you will use to save the file on your local computer once you collect all the chunks.

The node requesting the file will first need to send a data request to download the metafile, wait for a valid reply to that request (with a hash matching the requested hash and matching content), retransmitting the request periodically if it doesn't receive a reply **after a timeout (5 sec)**, and then download each of the file's data chunks in turn, **retransmitting the chunk request periodically if it doesn't receive a reply after a timeout (5 sec)**. This means that you will need to keep a state for files being downloaded in your gossipier. You can keep the protocol simple by requesting only one chunk at a time during a **given** file download. You need, however, to support parallel file downloads on your gossipier, where **each download** deals with one chunk at a time.

When you collect all the chunks for a given file, you should reconstruct it and save it on your gossipier in a directory called `_Downloads`. Just like the `_SharedFiles` directory, `_Downloads` should be co-located with your executable `gossipier`. You will use the value specified with the `-file` flag to name the file you downloaded.

**Important!** Your protocol should allow that a peer, who has only a few chunks of a file because it has only started downloading for example, can already offer those chunks to other peers.

Finally, for performance, you should consider storing the chunks of a file separately after the file has been indexed and completely downloaded, so if some specific chunk is requested, your gossipier does not have to reparse the whole file.

Usage of `client`:

- `UIPort` string  
port for the UI client (default "8080")
- `dest` string  
peer to download the file from
- `file` string  
file to be indexed by the gossipier
- `request` string  
request a chunk or metafile of this hash

**Output format for** `./client -UIPort=8080 -dest=anotherPeer`

`-file=flyingDrone.txt`

`-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318` (assume the file has 3 chunks, which the gossipier can see from the metafile)

DOWNLOADING metafile of flyingDrone.gif from anotherPeer

DOWNLOADING flyingDrone.gif chunk 1 from anotherPeer

DOWNLOADING flyingDrone.gif chunk 2 from anotherPeer

DOWNLOADING flyingDrone.gif chunk 3 from anotherPeer

RECONSTRUCTED file flyingDrone.gif

**Important:** The only flag combinations allowed for the `client` are the ones specified in Exercise 3, Exercise 4 and Exercise 6. **Any other** flag combination (e.g. `-request` and `-msg`, etc) must output an error on the client (`ERROR (Bad argument combination)`) and return error code 1 (`os.Exit(1)`). Also, if the request flag has a wrong format, output an error `ERROR (Unable to decode hex hash)` and return code 1.

## Hand-in Procedure and Grading

For each homework, you will receive a grade out of 6, as according to the EPFL grading system. To be considered for receiving full points (max. 6), you must upload and submit your fully-working code on Moodle (simply as a collection of source files) before the due date.

**You can always update your submission on moodle, so please start submitting early.**

**You wouldn't want a few minutes delay to cause you to miss the deadline.**

We will grade your solutions via a combination of testing and code inspection (and code plagiarism detection). We will run automated tests on your application and make sure it works as required, both when communicating with other instances of itself and when communicating with our own instances. Manual code inspection will mainly come into play when evaluating that you have implemented GUI and its communication with the backend correctly and also to verify that you have implemented all the required techniques.

Our very first test is that your code compiles when `go build` is executed on your files!

**No points will be given if your code does not compile.**

We will not dock points merely for stylistic deficiencies or ugly hacks—although we strongly encourage you to try to keep your code clean and maintainable, because you will have to keep building on it throughout the semester, and design flaws that you manage to work around in one lab may well come back to bite you in the next.

You're responsible for writing your own test script, you could take inspiration from the first assignment ;)

**Submission structure for automatic testing is the same as in Homework 1, please make sure to follow it:**

If your project code is in `$GOPATH/src/github.com/JohnDoe/Peerster/` (following go directory structure conventions), we assume you'll have the following directory structure (note that `client` is a directory):

```
GOPATH/src/github.com/JohnDoe/Peerster/
 main.go
 client/
 main.go
 other files and directories..
 other files and directories..
```

In order for your imports to work correctly during testing (for instance, if you have multiple modules in your project), please submit an archive that contains your Peerster code, with the directory structure `src/github.com/JohnDoe/Peerster/<your files and directories>`

Our testing script will do the following: it runs `go build` in the directory `GOPATH/src/github.com/JohnDoe/Peerster/`, which produces an executable `Peerster` that the script renames to `gossiper`. The script also calls `go build` in `Peerster/client`, which produces an executable `client` in the same directory. Then it copies `gossiper` and `client` in our testing framework and runs the tests, both for your own implementation, as well as the integration tests with the reference `gossiper` and `client`.

-----

**This completes the homework.**