Homework 2: Routing and File Sharing

CS438 - Decentralized Systems Engineering 2018

Homework out: Friday, 19.10.2018

Due date: Sunday, 23:59, 11.11.2018

Introduction

In this lab 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 via gossip.
2. A simple file-sharing protocol to enable nodes to copy files between each other.

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 ;)

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 your node receives a `Rumor` message with a new sequence number, 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. This will be the next-hop on your route to the given node, which will remain in effect until you receive the next rumor message (**with a higher sequence number**) from the same node. Of course, you need to forward the message to a random neighbor, whenever you receive a Rumor message.

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 Sequence # is `23`. 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 sequence number 25 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"`.

Unfortunately, if Peerster nodes only ever “announce” themselves to the network when the local user actually types in a message, nodes which 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 (in Exercise 2) 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.

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

Add a periodic timer, that generates a *route rumor* message. A route rumor message is just like the “chat rumor” messages you already generate, except it contains only the `Origin` and `ID` fields with an empty `Text` field. Modify your Peerster’s implementation receive path to accept and forward route rumors as well as chat rumors: the processing should be essentially the same, except you display a message to the user only when the rumor message contains a `Text` field.

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

Besides the periodic announcements (specified by **`rtimer`**), 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.

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).

```
Usage of ./gossiper:
  -UIPort string
     port for the UI client (default "8080")
  -gossipAddr string
     ip:port for the gossiper (default "127.0.0.1:5000")
  -name string
     name of the gossiper
  -peers string
     comma separated list of peers of the form ip:port
  -rtimer int
     route rumors sending period in seconds, 0 to disable
sending of route rumors (default 0)
  -simple
     run gossiper in simple broadcast mode
```

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

Exercise 3

Add a GUI mechanism to display a list of node `Origin` identifiers known to this node—i.e., for whom a “recent” 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
  -msg string
     message to be sent
```

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**:

```
pm := PrivateMessage {
      Origin : "Alice",
      ID : 0,
      Text : ""
      Destination: "Bob",
      HopLimit: 9,
}
```

- an `Origin (string)` containing the identifier of your node
- 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` key whose value is a `uint32`, which you initialize to some constant value, say 10 by default, when sending a private message, and every node on the forwarding path will decrement along the way, 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 PrivateMessage struct {
    Origin      string
    ID          uint32
    Text        string
    Destination string
    HopLimit    uint32
}
```

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

Finally, update your Peerster implementation to accept private messages, forward them (after decrementing the `HopLimit`) if the `Destination` field refers to another node, and display them to the local user if the `Destination` field indicates that the message is for the local node. Test your code by creating indirectly-connected chains of Peerster instances on your localhost.

Important! When your gossiper receives a `PrivateMessage`, it should not consider its `HopLimit` until sending it. Right before sending it you should decrement it and check to see if `HopLimit == 0`. If this is the case, you **DO NOT** send the message out!

When receiving a `PrivateMessage` destined to your gossiper, write at stdout:

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

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 identify files to each other to announce them 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 1

To share files, the first thing you will need to do is to allow the user to specify a file to share, and 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=carlton.txt
```

```
Usage of ./client:
  -UIPort string
    	port for the UI client (default "8080")
  -dest string
    	destination for the private message
  -file string
    	file to be indexed by the gossiper
  -msg string
    	message to be sent
```

To avoid working with absolute paths, create a directory called `_SharedFiles` **in the same directory as your gossiper 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 `carlton.txt` should be `$CWD/_SharedFiles/carlton.txt`, where `$CWD` is the absolute path of the executable **`gossiper`.**

Exercise 2

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` 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 gossiper.

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! SHA-256 hash of the metafile is the only unique identifier you will have for a file.

Exercise 3

Finally, we need a protocol for one node to retrieve a file from another node. For now we will assume the retrieving node already has the `MetaHash` of the desired file. We will use a simple one-chunk-at-a-time request/response download protocol.

Add support for two new messages to your protocol, for chunk requests and chunk replies, as well as for metafile requests and replies. These messages are similar to a `PrivateMessage` with additional fields for requesting and delivering 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 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 whether to route a message on to another node or to process it at the local node simply by looking for (and at the contents of) these fields before doing any local processing.

`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 DataRequest struct {
        Origin string
        Destination string
        HopLimit uint32
        HashValue []byte
  }
  ```

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

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

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

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

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 node ID and a hexadecimal `HashValue` of the metafile (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=carlton.txt
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe699
2be5318
```
```

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., carlton.txt 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. This means that you will need to keep a state for files being downloaded in your gossiper. You can keep the protocol simple by requesting only one chunk at a time during a given file download.

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

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, you can 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 gossiper does not have to reparse the whole file.

```
Usage of ./client:
  -UIPort string
     port for the UI client (default "8080")
  -dest string
     destination for the private message
  -file string
     file to be indexed by the gossiper, or filename of the
requested file
  -msg string
     message to be sent
  -request string
     request a chunk or metafile of this hash
```

Output format for `./client -UIPort=8080 -Dest=anotherPeer -file=carlton.txt -request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318` (assume the file has 3 chunks, which the gossiper can see from the metafile)

DOWNLOADING metafile of carlton.txt from anotherPeer
DOWNLOADING carlton.txt chunk 1 from anotherPeer
DOWNLOADING carlton.txt chunk 2 from anotherPeer
DOWNLOADING carlton.txt chunk 3 from anotherPeer
RECONSTRUCTED file carlton.txt

This completes the homework.