Homework 1: Gossip Messaging

CS438 - Decentralized Systems Engineering, Fall 2019

Homework out: Friday, September 27, 2019

Due date: Tuesday, October 15, 2019 @ 23:59

Introduction

In this homework, you will start building a small peer-to-peer application in the **Go programming language** called – for lack of a more imaginative name – *Peerster*. We will specify the functionality and protocol your application needs to implement, along with some implementation hints and pointers to relevant information, but in this and all labs in the course you will ultimately be responsible for gathering all the necessary information and figuring out how to implement what you need to implement – just as you will need to in industry or research programming jobs. Since everyone in the class will be developing an application that is supposed to "speak" the same protocol, your application should – and will be expected to – interoperate with the implementations built by the other course participants. **We provide the room INF 1 every Monday from 13:15 to 15:00 for you to hack together and test your implementations.**

Throughout this development and debugging process, you are welcome to discuss challenges and techniques with your fellow students, exchange pointers to relevant information or algorithms, debugging tips, etc., ***provided you each write your own code independently*****. Homeworks are individual per student.**

Teaching and student assistants will be available at INJ 218 every Friday 15:15-17:00, to discuss with you how to architect your implementation. They are not going to debug your code but they can help you ask the right questions just like your software engineer colleagues will do in the future.

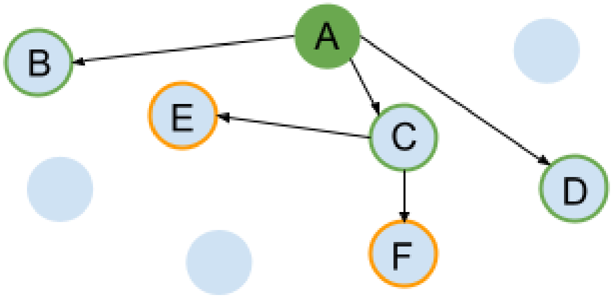
If you are not familiar with Go, parts of this tutorial will prove useful.

This first homework focuses on putting together the basic elements of a simple peer-to-peer application providing text-based multicast chat, similar to Internet Relay Chat (IRC). The homework contains three main components:

1. Creating a UDP (datagram) socket to communicate with other nodes.
2. Implementing a simple gossip algorithm to distribute messages among directly and indirectly connected nodes.
3. Constructing a simple user interface for viewing and entering messages.

Gossiping in Peerster

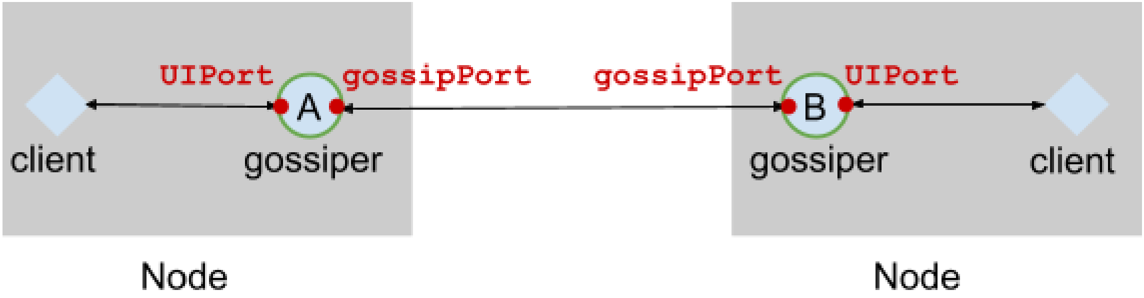
Gossip protocols are distributed protocols for **robust information exchange**, typically deployed on **dynamic network topologies**, e.g, because nodes can join and leave the network (also called churn), they are mobile, their connectivity varies, etc. Examples of applications are ad-hoc communication between self-driving cars, peer-to-peer networks that broadcast a TV program, sensor nodes that detect fire hazard in remote areas. The way gossip protocols spread information resembles gossipping in real life: a rumor may be heard by many people, although they don't hear it directly from the rumor initiator. The figure below depicts a gossip protocol, with A the rumor starter.



When a node joins a gossip protocol, it has the contact information (e.g., network address) of a few nodes it can send messages to. For instance, node C in the figure above knows the addresses of nodes E and F. Additionally, when a node receives a message, it learns the address of the sender. As an example, node C learns the address of node A when it receives the message from A.

Peerster Design

Each node in Peerster acts as a **gossiper**, as depicted above, but also **exposes an API to clients** that enables a client to send messages, list received messages etc. The client could, in principle, run anywhere and contact nodes remotely, e.g., via IP Anycast. However, for simplicity for this assignment, we assume clients run on the node's machine. The gossiper node communicates with other peers on the `gossipPort`, and with clients on the `UIPort`. Below you can find the high-level design:



You will build both the gossiper and the client:

- In a first version, the client interaction is through a simple command-line interface (CLI), as we explain in Part 1. Later, in Part 3, the client interaction is web-based.
- The gossiper needs to listen on two different ports for client and peer messages, respectively. Because listening on a port is an infinite loop, you could have two different threads in gossiper, each handling one port.

Submission structure for automatic testing, 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..
```

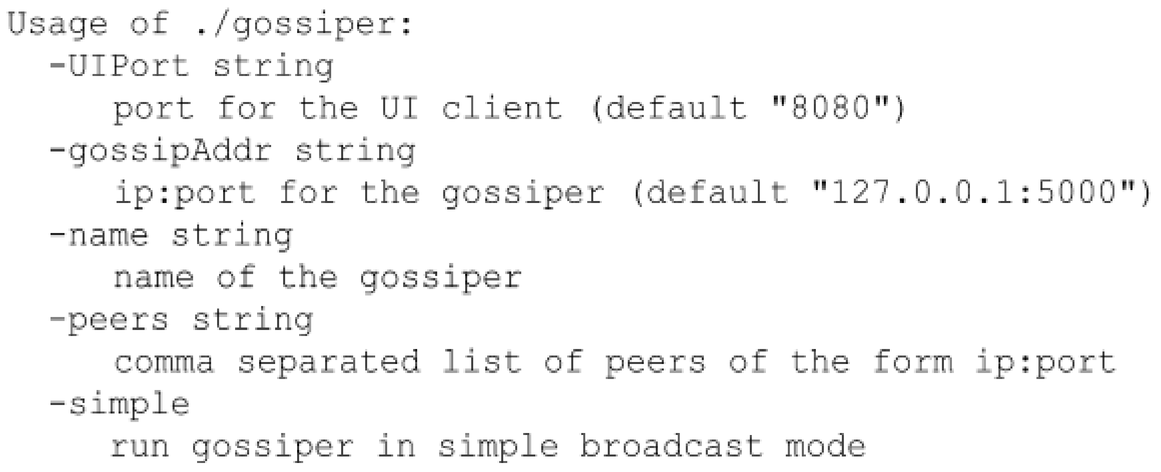
In order for your imports to work correctly during testing (for instance, if you have multiple modules in your project), please submit an archive that contains your Peerster code, with the directory structure `src/github.com/JohnDoe/Peerster/<your files and directories>`

Our testing will do the following: it copies the tests (e.g. test_1_ring.sh) in `GOPATH/src/github.com/JohnDoe/Peerster/` and runs in. The script runs `go build` in the directory `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. Similarly if you copy the script `test_2_ring.sh` to `GOPATH/src/github.com/JohnDoe/Peerster/` and run it.

(15 p) Part 1: Network Programming in Go

As a first step, you need to build a simple protocol that allows communication over UDP (User Datagram Protocol). There is no real gossiping functionality yet, just broadcast. We'll add gossiping in Part 2. Your `gossiper` program takes as arguments the two ports (`UIPort` and `gossipAddr`), as well as a list of one or more addresses of other gossipers it knows of at the beginning, in the form `ip1:port1,ip2:port2,etc` (note that list elements are separated by a comma). Pay attention: using ports below 1024 requires root privileges and quite a few of them are reserved, thus please use higher ports. It also needs a flag "simple" to enforce simple broadcast mode for compatibility. **You need to follow this input format in order to comply with automatic testing, otherwise tests will fail.** We recommend you use the flag package from go's standard library to implement your CLIs.

```
./gossiper -UIPort=10000 -gossipAddr=127.0.0.1:5000 -name=nodeA
       -peers=127.0.0.1:5001,10.1.1.7:5002 -simple -antiEntropy 20
```



```
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
  -simple
     run gossiper in simple broadcast mode
```

- antiEntropy <time_seconds>

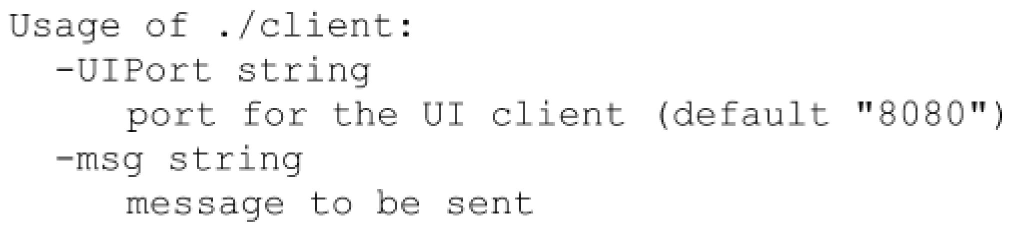
Use the given timeout in seconds for anti-entropy (relevant only for Part 2). If the flag is absent, the default anti-entropy duration is 10 seconds.

We assume the `client` runs locally, so the `gossiper` listens for `client` messages on `127.0.0.1:UIPort`. During interoperability tests, the IP addresses in `gossipAddr` and `peers` will of course be different than `127.0.0.1`.

Your local `client` program takes two arguments: `UIPort`, which is the port on which the `gossiper` listens for the messages that are sent by the client, and `msg`, which is the message to be sent to the `gossiper`.

```
./client -UIPort=10000 -GUIPort=-msg=Hello

Usage of ./client:
   -  UIPort string
             port for the command line interface
   -  GUIPort string:
             Port for the graphical interface
   -  msg
             Message to be sent
```



```
Usage of ./client:
  -UIPort string
     port for the UI client (default "8080")
  -msg string
     message to be sent
```

A **Peerster simple message** contains the following (in Part 2, Peerster messages will have different types and contain more fields, but in Part 1 we are concerned only with these simple messages):

- Original sender's name
- Relay Peer's address, in the form `ip:port`
- The text of the message

```
type SimpleMessage struct {
    OriginalName string
    RelayPeerAddr string
    Contents string
}
```

To provide compatibility with future versions, we define a GossipPacket, the **ONLY** type of packets sent to other peers. For now it may only contain a SimpleMessage:

```
type GossipPacket struct {
    Simple *SimpleMessage
}
```

Creating a GossipPacket:

```
packetToSend := GossipPacket{Simple: simplemessage}
```

When a `gossiper` receives a simple message in simple broadcast mode:

- If it comes from a client, the gossiper sends the message to all known peers, **sets the OriginalName of the message to its own name and sets relay peer to its own address**
- If it comes from another peer A, the gossiper (1) stores **A's address from the relay peer field** in the list of known peers, and (2) **changes the relay peer field to its own address**, and (3) sends the message to all known peers **besides peer A, leaving the original sender field unchanged**

Food for thought: is it necessary to put the Relay Peer's address in the message? Where else could the receiving peer obtain the relay peer's address from?

Your `gossiper` program should write at standard output, every time it receives a SimpleMessage, two lines:

- The first line should be:
    - **`SIMPLE MESSAGE origin <original_sender_name> from <relay_addr> contents <msg_text>`** when the message comes from another peer, `relay_addr` being the `ip:port` of the relay.
    - **`CLIENT MESSAGE <msg_text>`** when the message comes from a client.
- The second line contains the addresses of all known peers, including those given when initializing the node, in the form **`PEERS ip1:port1,ip2:port2,etc`**

Example of a running `gossiper` program:

```
CLIENT MESSAGE Hello
SIMPLE MESSAGE origin nodeB from 127.0.0.1:5002
contents Bonjour
PEERS 127.0.0.1:5002,127.0.0.1:5003
```

Those not yet familiar with UDP can learn about it at these or any number of other references:

- Wikipedia on UDP - like a box of chocolates, you never know what you'll see there today, but it'll probably be OK.
- RFC 768, the official (though fairly trivial) specification of UDP as an Internet transport protocol. You'll be reading a few RFCs in this course, so it's worth getting used to their ASCII-art style.
- Go library for network communication.
- Linux udp man page on using UDP sockets via the raw Berkeley sockets API.

Here is how an udp socket address is represented in Go's "net" package:

```
// UDPAddr represents the address of a UDP end point.
type UDPAddr struct {
	IP   IP
	Port int
	Zone string // IPv6 scoped addressing zone
}
```

Then, creating a `Gossiper` running at address, where address is a string of the form *ipaddr:port*, could look as follows:

```
func NewGossiper(address, name string) *Gossiper {
	udpAddr, err := net.ResolveUDPAddr("udp4", address)
	udpConn, err := net.ListenUDP("udp4", udpAddr)
	return &Gossiper{
		address:   udpAddr,
		conn:      udpConn,
		Name:      name,
	}
```

```
        }
```

Your program also would need to store a list of peer addresses that it knows (either from the beginning or because it receives messages from them) and implement a callback function that is invoked when a new gossip message arrives. Adding these to your `Gossiper` structure is up to you.

Serializing and Deserializing Messages

Because nodes that communicate may run on different architectures and operating systems, we need to serialize (or marshall) messages before sending them over the network, which means we first convert the message to an architecture-independent format.

Let's start by sending simple text messages over the network

```
        Message struct {
                Text string
        }
```

For serializing and deserializing, you will use the [Protobuf library](), a standard approach for encoding structures in Go. It defines message format using Go types. You need to include `https://github.com/dedis/protobuf` in your imports to be able to use the library. You can encode a message as follows (pay attention that `msg` is a pointer, of type `*Message`)

```
        msg := &Message{"hello"}
        packetBytes, err := protobuf.Encode(msg)
```

Then you can pipe the resulting bytes to the UDP connection of your gossiper as

```
        net.UDPConn.WriteToUDP(packetBytes, upd_addr)
```

Similarly, reading from UDP connection and decoding can be done as

```
        net.UDPConn.ReadFromUDP(packetBytes)
        protobuf.Decode(packetBytes, packet)
```

Exercise 1

For now, run multiple instance of the program on your local machine (use as IP `127.0.0.1`) and check that you can send messages, and that the other peers receive them at standard output. Also, check that you store the addresses of peers that you receive messages from.

Testing

Run `./test_1_ring.sh` to test your program. Please follow the directory structure we indicated in the section Peerster Design. If the test fails, then most probably your program isn't correct. If the test passes successfully, it means your program might be running correctly, but you should test your implementation in more realistic scenarios (e.g. test if your peerster can communicate with your classmates' peersters).

(30 p) Part 2: A Simple Gossip Protocol

There are two key problems with the trivial protocol we implemented so far:

1. The Internet in general, and UDP in particular, are unreliable and offer *best-effort* communication, which means messages may get lost or duplicated in the network for a wide variety of reasons (which you already know, of course, from your networking course). This may be unlikely to happen when you're sending a message from one UDP socket to another on the same host, but becomes much more likely once you start sending messages between hosts: UDP datagram loss or duplication in the network may cause one user's chat message to be lost, or to appear multiple times, in another user's chat log.
2. Although in the Internet's original architecture the Network Layer (IP) underlying UDP was intended to guarantee universal “any-to-any” connectivity, so that any Internet host could communicate directly with any other, this original design principle has become seriously eroded as middleboxes such as firewalls and Network Address Translators (NATs) have proliferated in the Internet for practical and security reasons. Thus, UDP-level connectivity is now often *asymmetric*: if A can talk to B and B can talk to C, that doesn't necessarily mean A can talk directly to C via UDP. Thus, we will need to explore mechanisms for *indirect* communication, so B can forward a message from A to C if necessary.

When the main objective is merely to ensure that a number of cooperating hosts or processes each obtain copies of whatever messages any of them send, as when implementing a chat room, one of the simplest yet also fastest and most reliable known algorithms for propagating those messages is known as a *gossip protocol*. USENET, the Internet's original widespread and decentralized public “chat room” used such an algorithm. We will not describe gossip protocols here in detail; you should familiarize yourself with them in some of the many resources available online or in any distributed systems textbook. A few pointers to start with:

- The Wikipedia page offers a high-level summary, though probably not all the details you will need (perhaps depending on the mood of the current editors and the phase of the moon).
- The original Epidemic Algorithms research paper by Demers et al at Xerox PARC in the 1980s. Perhaps not the easiest read, but there's no more definitive source.
- RFC 1036, the standard describing the way USENET news messages were formatted and propagated gossip-style in USENET's heyday. Pay particular attention to section 5 at the end on propagation, and section 3.2 on the Ihave/Sendme protocol.
- Textbook: Tanenbaum 4.5.2 “Gossip-Based Data Dissemination”

Gossip in Peerster

To implement a gossip protocol in Peerster, we will basically need to do two things:

- Since these messages will be propagated via unreliable UDP datagrams rather than on reliable TCP streams, hosts will need to acknowledge messages they have received, and the sender must be able to resend messages whose UDP datagrams may have been dropped by the network (i.e., for which the sender did not receive an acknowledgment).
- Since all peers may not directly know about all others, each host must be able to forward messages it has received to other hosts who might not yet have received them, while ensuring that this forwarding does not cause infinite loops (e.g., A sends a message to B, which sends it back to A, which sends it back to B, etc.).

To accomplish these goals, we will need to give messages unique IDs with which Peerster hosts can keep track of and refer to user chat messages. Read (or re-read) sections 2.1.5, 3.2, and 5 of RFC 1036 for one classic example of how to design and use message IDs in gossip protocols.

Peerster will identify messages via a pair of values:

- an *origin* uniquely identifying the Peerster application from which a particular user chat message originated, and
- a *sequence number* that distinguishes successive messages from a given origin.

A given Peerster node assigns sequence numbers to client messages consecutively starting with 1. Of course, each Peerster node generates its own sequence numbers. Sequence numbers allow peers to “compare notes” on which messages from which other peers they have or have not received. For example, if peer A has received messages originating from peer C up to sequence number 5, and compares notes with peer B that has received C's messages only up to sequence number 3, then A knows that it should propagate C's messages 4 and 5 to B. This convention essentially amounts to implementing a *vector clock*:

- Wikipedia
- Fidge, “Timestamps in Message-Passing Systems That Preserve Partial Ordering”
- Mattern, “Virtual Time and Global States of Distributed Systems”
- Textbook: Tanenbaum 6.2 “Logical Clocks”, especially 6.2.2 “Vector Clocks”
- Background: Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System”

Given this approach to identifying user messages, we can now define more specifically the two types of messages comprising Peerster's gossip protocol:

- **Rumor message:** Contains the actual text of a user message to be gossipped. The message must be a `struct` that has two fields: `Origin` (**string**), which identifies the message's original sender, and two fields: `Text` (**string**) is the content of the message; and a new `ID` (**uint32**) that contains the monotonically increasing sequence number assigned by the original sender. Let's make the format more clear. For example, if Alice is sending "Hi" to Bob (we assume the sequence number is 23), the rumor message should be:

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

```
type RumorMessage struct {
    Origin   string
    ID     uint32
    Text   string
}
```

As you see, the Relay Peer's address is not in the RumorMessage. To answer the question in Part1: that's because we obtain the relay peer's address when receiving as a return value when calling the method to receive an UDP packet.

- **Status message:** Summarizes the set of messages the sending peer has seen so far. The status message contains only one field `Want`, whose value is essentially a vector clock with origin IDs the peer knows about and its associated values (**uint32**) represents the lowest sequence number for which the peer has *not yet* seen a message from the corresponding origin. This vector clock should be implemented as a slice (array) of `PeerStatus` structures, which has two fields: `Identifier` for the origin's name (**string**) and `NextID` (**uint32**) for the next unseen message sequence number.

```
ps := PeerStatus {
      Identifier: “Alice”,
      NextID: 4,
}

type PeerStatus struct {
      Identifier string
      NextID     uint32
}
```

That is, if A sends a status message to B containing the pair <C,4> in its Want structure, this means A has seen all messages originating from C having sequence numbers 1 through 3, but has not yet seen a message originating from C having sequence number 4 (and A may or may not have seen messages from C with sequence numbers higher than 4). Anyway, for a regular status message, it should be:

```
sp := StatusPacket {
      Want:   [PeerStatus{Identifier:   “Alice",   NextID:
4,}]
}
```

```
type StatusPacket struct {
	Want []PeerStatus
}
```

Message types and interoperability

Just as in Part 1, in Part 2 the client always sends a type `Message` to the gossiper.

When a peer receives a message, it needs to be able to tell whether the message is a `RumorMessage` or `StatusPacket` (or `SimpleMessage` from Part1). A simple way to do this is as follows: peers encapsulate messages in a generic packet of type `GossipPacket`:

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

To send a rumor `RumorMessage`, for instance, you simply initialize

```
packetToSend := &GossipPacket{Rumor: rumor}
```

and send it. `Status` will be `nil` by default.

The receiving peer decodes the UDP bytes that it receives into `receivedPkt,` which is a `GossipPacket`.

```
receivedPkt := &GossipPacket{}
err := protobuf.decode(recvBytes, receivedPkt)
```

In this case, `receivedPkt.Rumor` is not `nil` and contains the rumor message, while `receivedPkt.Status` is `nil`. By checking which of `receivedPkt.Rumor` or `receivedPkt.Status` is `nil`, you know the type of message you received. More message types will be added in the next homeworks. In a `GossipPacket`, one and only one field should be non-nil.

Make sure that you use the same ordering of variables in your structs as it is stated in the handout!

Rumormongering

We will now implement a rumormongering protocol.

- Whenever a peer S (sending peer) obtains a new ~~**`RumorMessage`**~~ message **that *it did not have before*** – either a **`Message`** from the local client (in which case this peer becomes the message's origin), or a **`RumorMessage`** from another peer in a new rumor message – the peer picks a **random receiver peer R (from the list of all known peers, which includes peers given at bootstrap, as well as peers this node received messages from)** and sends a copy of the rumor to that target.

- **The receiver peer R acknowledges the message by sending a `StatusPacket` back to peer A.**
    - If the sending peer **receives a `StatusPacket` acknowledging the transmission**, it compares the vector in the status message with its own status. There are three cases: (1) The sender has *other* new messages that the receiver peer has not yet seen, and if so repeats the rumormongering process by sending **one** of those messages **to the same receiving peer**. Note that this message causes the receiver peer to start rumormongering; (2) The sending peer does not have anything new but sees from the exchanged status that the *receiver* peer has new messages. Then the sending peer itself sends a **`StatusPacket`** containing its status vector, which causes the receiver peer to send send the missing messages back **(one at a time);** (3) If neither peer has new messages, the sending peer (rumormongering) peer S flips a coin (e.g., `rand.Int() % 2`), and either (heads) picks **a new random peer to send the rumor message to**, or (tails) ceases the rumormongering process.

    - If, after some time, e.g., a timeout of 10 seconds, the sending peer has not received a **`StatusPacket`** from the receiver peer, then it simply picks another peer to rumormonger its message. To implement this, you will need to set a timer to have a handler reinvoked after some period if no message has been received by then. The `time.NewTicker(`*`some_value`*`)` function is your friend.

To keep things simple, you should always send new rumor messages from a given origin in sequence number order. That is, if A is rumormongering with B and has new messages (C,3) and (C,4), then A should propagate (C,3) to B before propagating (C,4).

An important question is how to make sure a peer's name (identifier) is unique. Ultimately how you come up with an origin identifier is up to you, as long as you have a good reason to believe it will be unique (e.g., your SCIPER). A reasonable approach would be to pick a random number when the program starts and include that in the peer's name. Even better might be to use a long, cryptographically strong random number or cryptographic hash, but we will get to do that in later labs. **For now, let's keep the peer's name the one given in command line, as it's easy to use different name when all peers run on localhost :) We advise you to use your SCIPER number when we'll test your program's interoperability with other students' peers.**

Exercise 2. Implement a simple rumormongering scheme as described above. Test your application by running two, three, or four Peerster instances on your local machine.

Anti-entropy

As you should know from what you've read on gossip algorithms so far, rumormongering by itself is not guaranteed to ensure that all participating nodes receive all messages: the process may stop too soon. To ensure that all nodes eventually receive all messages, you will need to add an anti-entropy component. In Peerster, we will take a simple approach: just create a timer that fires periodically (**set the timeout to 10 seconds for a default mildly aggressive approach when the antiEntropy flag is absent, and to the value of the antiEntropy flag when it's present**), and causes the peer to send a status message **to a randomly chosen peer from the list of known peers, which includes peers given at bootstrap, as well as peers this node received messages from**. If the peer who receives the status message sees a difference in the sets of messages the two nodes know about, that neighbor should either start rumormongering itself or send another status message in response, so that the original node will know which message(s) it needs to send.

Exercise 3. Implement anti-entropy as outlined above. Test it to make sure it reliably propagates messages across multiple hops.

Testing

Your `gossiper` program should write at standard output:

- **`CLIENT MESSAGE <text_content>`** when receiving a message from a client
- **`SIMPLE MESSAGE origin <original_sender_name> from <relay_addr> contents <msg_text>`** when receiving a `SimpleMessage` from another peer
- **`RUMOR origin <original_sender_name> from <relay_addr> ID <msg_id> contents <msg_text>`** when receiving a `RumorMessage` from another peer
- **`MONGERING with <ip:port>`** when sending a `RumorMessage` to a peer at `ip:port`
- **`STATUS from <relay_addr> peer <name1> nextID <next_ID1> peer <name2> nextID <next_ID2> etc`** when receiving a `StatusPacket` from another peer with address `ip:port`
- **`FLIPPED COIN sending rumor to <ip:port>`** when rumormongering continues after coin flip
- **`IN SYNC WITH <ip:port>`** when the peer receives a status message and is up-to-date with all messages it contains
- **`PEERS <ip1:port1>,<ip2:port2>,etc` list of addresses of all known peers, printed every time the node receives a message**

Please follow the format exactly as specified, otherwise automatic testing will fail.

Example of a running `gossiper` program:

```
RUMOR origin E from 127.0.0.1:5000 ID 1 contents Weather_is_clear
PEERS 127.0.0.1:5002,127.0.0.1:5000
MONGERING with 127.0.0.1:5002
STATUS from 127.0.0.1:5002 peer E nextID 1
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5002 peer E nextID 2
PEERS 127.0.0.1:5002,127.0.0.1:5000
CLIENT MESSAGE Winter_is_coming
STATUS from 127.0.0.1:5000 peer E nextID 2 peer B nextID 1
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5002 peer E nextID 2 peer B nextID 2
```

```
PEERS 127.0.0.1:5002,127.0.0.1:5000
STATUS from 127.0.0.1:5000 peer E nextID 2 peer B nextID 2
PEERS 127.0.0.1:5002,127.0.0.1:5000
IN SYNC WITH 127.0.0.1:5002
```

Run `./test_2_ring.sh` to test your program. Please follow the directory structure we indicated in the section Peerster Design. If the test fails, then most probably your program isn't correct. If the test passes successfully, it means your program might be running correctly, but you should test your implementation in more realistic scenarios (e.g. test if your peerster can communicate with your classmates' peersters).

`Test_2_ring.sh will be uploaded soon.`

Interoperability

The above functionality should give your application everything it needs, at least in theory, to operate "at large" over the real Internet. **We'll test interoperability outside your `localhost` by using our own instance of the program that implements all the aforementioned functionalities.** Our program will not know any peers by default but it will "learn" them upon the reception of correctly formatted rumor messages.

(15 p) Part 3: Basic GUI and Web Server Programming

You'll add to the CLI client a basic graphical user interface (GUI). Although GUI programming isn't a primary topic of this course, your P2P application will need a user interface of some kind to be able to chat with other students in the classroom. We encourage you to keep the interface as minimal as you like, simply using HTML to create a layout and Javascript along with jQuery to implement communication between the frontend and backend. But feel free to use any frontend framework to simplify this process. The GUI is not tested automatically, so no jQuery / HTTP API is imposed.

Your GUI needs to contain three boxes:

- a ***Chat box*** where peer messages appear
- a ***Node box*** where a list of known nodes (their ip addresses and port numbers used) is shown
- an ***ID box*** where your peer ID is displayed

Subsequently, the GUI needs to contain two buttons:

- *Send* a message to peers (which you also implemented in the CLI),
- *Add* a new node to the list (new functionality previously nonexistent in the CLI),

Your frontend should communicate with the backend using e.g. jQuery to keep the chat box up-to-date with the messages received from the peers via gossiping, and to refresh the list of nodes (an unknown node should be added to the list if a gossip message is received from it).

To serve your frontend and to handle POST and GET requests, you can use Go FileServer functionality, the HTTP request multiplexer library [1] and HTTP handlers [2] that simplify handling requests to a single subdomain to one line of Go code.

If you are free to usage any serialization for messages between the GUI and your backend. We recommend using the native to Javascript JSON format. The simple and acceptable approach for refreshing GUI is to periodically send GET request to the backend to retrieve the latest information and refresh the chat and the list of connected nodes.

Use port `8080` for communication with your frontend. It is better that you use a fixed port in this case to avoid unlikely but possible clashes when we test your program by running multiple instances on one machine.

Exercise 4. Use HTML to create basic GUI layout so that it contains a Chat box, a Node box, an ID box, and two buttons for sending messages and adding nodes.

Exercise 5. Create Javascript handler functions that will be invoked when the buttons are pressed and that will enable communication of the GUI with your backend using jQuery POST and GET requests.

Exercise 6. Create a web server in Go that will server you GUI files to a given port and will wait for POST and GET requests from the GUI to subdomains /message, /node, and /id. Firstly, implement the /message receive path, so that whenever you type a message into the GUI and press Send, it is sent to the backend. Also, the backend should return the latest chat messages to GUI upon reception of a corresponding GET request, same thing for the list of connected nodes with a GET request to /node.

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) by the due date, which you can find at the beginning of this document. **You can always update your submission on moodle, so please start submitting early. You wouldn't want to miss the deadline because of a few minutes of delay. Late submissions are not possible.**

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 must compile 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 keep your code clean and maintainable, because you will most likely be 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.

We will provide you with some test scripts, together with an example test data, for you to test that your code compiles and behaves as expected in simple cases (date TBD). Note that these scripts are not going to be extensive, meaning, you should not assume that your code is working correctly and you are done just because it passes these tests. Therefore, we strongly encourage you to test your implementation with the implementations of your classmates by having them communicate with each other. For the actual grading of your submission, we will use our own test framework.

Unless otherwise specified, the tests assume a full implementation of the homework. If you implement only parts of a homework, we cannot guarantee there will be tests for that particular functionality that would give you points for it.

Inter-homework Dependencies

Every homework builds on the previous one and requires a full and working implementation of the previous homework. Thus, we strongly encourage you to fully implement every homework. However, if it so happens that: (1) you were not able to complete a homework, or (2) you fully implemented your homework but the poor code design makes is hard for you to build on top, we offer you an alternative. After each homework, you will receive 3 random anonymized submissions of your colleagues that you need to review, **and can choose to build on top of any of these 3 assignments for your next homework. If you decide to do so, you need to specify that homework's identifier**.

This completes the homework.