CS-438 - Decentralized Systems Engineering, Fall 2021

Introduction + Homework 0

Network Programming in Go

Author	DEDIS lab, EPFL
Revision	September 2021 - 1.0.1
Publish date	Friday, September 24, 2021
Due date	**Tuesday, October 12, 2021 @ 23:55**

Table of contents

Table of contents **1**

Introduction to all the CS438 homeworks **1**
- Gossip protocols 1
- What you're going to build during the homeworks 1
- Inter-homework Dependencies 3
- Workload 3
- Prerequisites 3
- General structure of homeworks 3
- Gitlab 5
- Programming environment 5

Homework 0 **6**
- End result 6
- Objectives 6
- Peer communication 6
- REST API and CLI 7

Your tasks **8**
- Task 0: Explore the codebase 8
- Task 1: Implement the basic peer functionalities 9
- Task 2: Implement a simple routing table 9
- Task 3: Implement a unicast function 10
- Task 4: Implement a chat messaging mechanism 10
- Task 5: Implement a socket based on UDP 10

How to test your program **11**
- Automatic testing 11
- Manual testing 11
- Automatic testing through GitLab 12

Hand-in Procedure and Grading **13**

Appendix A - Tests definitions **14**
- Socket implementation 14
  - Socket creation/close 14
  - Simple send/recv 14
  - Multiple send/recv 15
- Peer implementation 15
  - AddPeer function 15
  - SetRoutingEntry 16

RegisterNotify 16
Unicast function 17
Relays of packet 17
Integration tests 18
Unicast 18

Introduction to all the CS438 homeworks

Gossip protocols

Gossip protocols are distributed protocols for robust information exchange. Those protocols are 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, etc... 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. Figure 1 illustrates a gossip protocol in action.

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 figure 1 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.

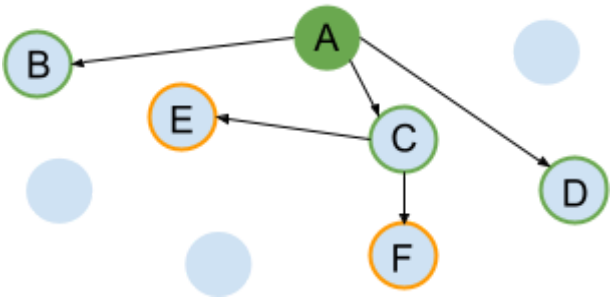

Figure 1: A gossip protocol in action. A is the rumor starter and sends a rumor to its known addresses B, C, D. Node C will then spread the rumor it received from A to its known addresses E, F. It will also learn the address of node A.

What you're going to build during the homeworks

In this course, you are going to build a gossip-based peer-to-peer application called ***Peerster***. As illustrated in figure 2, Peerster is made of multiple peers interacting with each other. A peer refers to an autonomous entity in the Peerster system, and each peer acts as a gossiper[1]. Peerster is built using the **Go programming language**.

With every homework, Peerster's functionality will become richer, starting with a simple message unicast, to gossip messaging, routing for one-to-one communication, data sharing

[1] Note that the homework handouts, technical docs, and code base use the terms *peer*, *node, participant,* and *gossiper* interchangeably

in the manner of BitTorrent, and finishing with an implementation to reach consensus on file names.

Your job will be to implement a peer that follows our specification. If your peer correctly follows the specification, then it should be able to talk to other peers, like the ones from your colleagues, and therefore form a Peerster system. Building the Peerster system and connecting the peers together is already done for you. In the tests that you can use to check your implementation, we build a Peerster system with multiple instances of your own peer implementation. We also have "integration tests", where we build a Peerster system mixed with your own peers and peers from our reference implementation.

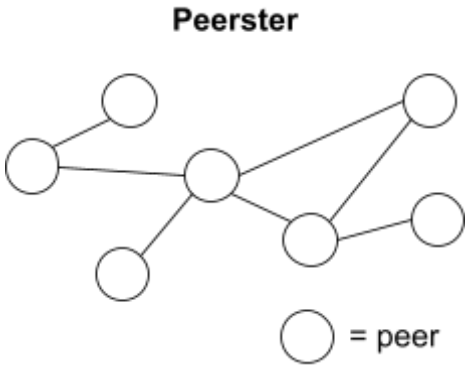

Figure 2: Global view of the Peerster system, which is made of connected peers that can exchange messages. Each peer in Peerster acts as a gossiper. Your job is to implement a peer that follows the provided specification.

We are going to specify the functionality and protocol that your peer needs to implement, along with some implementation hints and pointers to relevant information. But in this and all other homeworks in the course, you will ultimately be responsible for gathering the necessary information and figuring out how to implement what you need to implement – just as you will need to do in the 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.

For each homework assignment, we are going to provide you with the files containing the interfaces to be implemented, code skeleton, test framework, and some useful functionality and implementation hints. Note that the handout won't provide enough information to correctly implement the homeworks. You will have to gather the following sources: the handout, code comments, code documentation, and the tests.

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 assistants will be available in room INJ 218 every Friday 15:15-17:00, to discuss with you how to architect your implementation. ***TAs 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. Room INF 1 is available every Monday from 13:15 to 15:00 for you to hack together and test your implementations, without TA supervision.

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 assignment. However, if it so happens that: (1) you were unable to complete a homework, or (2) you fully implemented your homework but the poor code design makes it hard for you to build on top of, we offer you an alternative. After each homework, you will receive three random anonymized submissions of your colleagues that you will have the opportunity to review. You may build from any of these other three solutions to complete your next homework assignment, if you think one of them offers a more solid base than your own solution of the current homework. If you decide to adopt and build on another solution in this way, you need to specify that homework's identifier (more instructions will be given in HW 1).

However, be warned that there is no guarantee regarding the quality of these 3 randomly assigned submissions - they might be less good than your own implementation. Your best strategy, thus, is still to complete the homeworks yourself.

Workload

This is a systems building course and you are expected to deliver a working system. On average, we aim for you to spend about 5-10 hours per week on homeworks / project. That amounts to an estimate of 15-30 hours per homework. It is, however, difficult to assess more precisely how time consuming bugs or concurrency issues will prove to be for each individual.

Prerequisites

The major prerequisites for this project is a good understanding of the Go language. If you are familiar with a system language like C and an object oriented programming language like Java, then you should be able to quickly get familiar with the Go language. Here is a list of some topics with Go that we advise you to get strongly familiar with: channels, Goroutines, go embedding, and the sync package.

If you are not familiar with Go, we suggest you first complete the Tour of Go tutorial.

General structure of homeworks

You are going to start with a skeleton of code that you build on. This skeleton can be found in `/peer/impl/mod.go`. This is in this `impl` package that you are going to implement your peer.

We provide an abstraction of a peer, ie. an interface, and your job is to implement a "concrete" peer that follows this abstraction. The interface that defines a peer is located in `/peer/mod.go` as `type Peer interface{}`. The Peer interface will be augmented throughout the homeworks to add new functionalities to the peer. The skeleton already contains the functions that need to be implemented for HW0.

To get your peer implementation, we placed a special function in the skeleton that must return your implementation of a Peer:

```
NewPeer(conf types.Configuration) peer.Peer
```

This function must not be renamed or moved to another package. But you are free to change its content. Therefore, the only thing we are asking from your peer implementation is this `NewPeer` function, which tells the following: "*here is a types.Configuration argument, do something with it and return me back a peer.Peer*".

For example, to build a two-nodes Peerster system, we are going to call twice `NewPeer`:

```
p1 := NewPeer(conf1)
p2 := NewPeer(conf2)
```

and then we are going to invoke functions on peers `p1 p2`, which are the ones described by the peer.Peer interface, and implemented by you. Listing 1 shows an example of invoking peer.Peer functions on p1 and p2. It gives you an idea about how your implementation is going to be used.

```
p1.Start()
p2.Start()
p1.AddPeer(p1Addr)
p1.Unicast(p2Addr, myMsg)
p1.Stop()
p2.Stop()
```

Listing 1: Example using functions on peers p1 and p2. Here we start the peers and make p1 send a unicast message to p2.

To get familiar with the project architecture and design, **we invite you to read the design documentation** on the source repository in `docs/README.md`, and **come back later to read the rest of this document**.

Did you read the **design documentation** ? If not, please do. Then come back to read the rest of this document.

Gitlab

You will use Gitlab to host your code and work on it. Each student is provided with a GitLab repository that can be accessed with the credentials for gitlab.epfl.ch.

Programming environment

We highly recommend you to use a UNIX based system to develop and test your work. We observed that the Windows environment is extremely slow at running tests, which can make them fail. If you are on Windows we advise you to set up a virtual machine, or use Windows subsystem for Linux (Debian is a good choice).

In term of software packages, here are the dependencies:

- Golang (1.16)
- Make (if you want to use the makefile)

Homework 0

End result

In this homework you are going to add a simple messaging functionality to Peerster. By the end, your peers should be able to connect to each other and exchange direct messages, much like in a standard messaging application. Direct messages use what we call “unicast” calls, as opposed to “broadcast” - which will be implemented in the next homework. To play around with your peers, we provide a web-based GUI that offers a minimalistic chat interface. See figure 3.

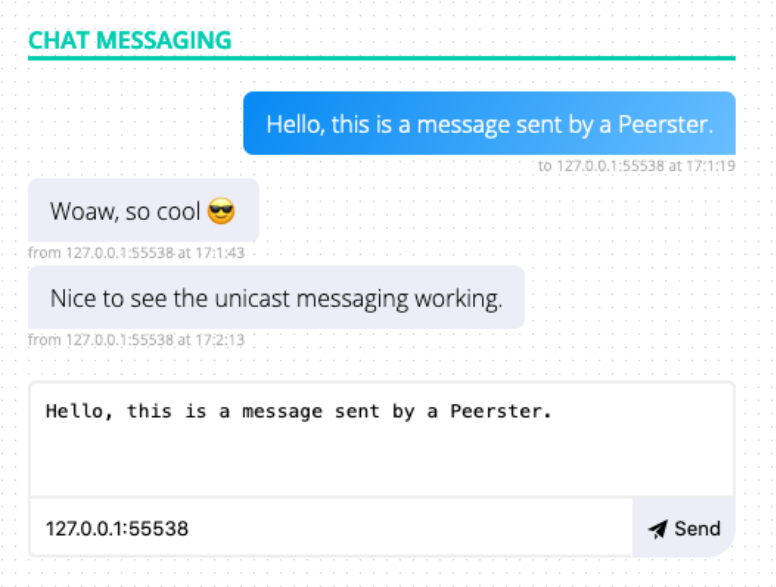

Figure 3: Minimalistic web-based GUI to play around with the chat functionality that HW0 implements. This is the result of HW0.

Objectives

The objective of HW0 is to introduce you to the various concepts and elements you will need to use during the homeworks: The Go language (and Go networking in particular), Peerster architecture, project structure, how you’re going to implement your peer, etc… We only ask you to implement basic functionalities on the peer, as it prepares the ground for the next homework. However, we advise you to already take care in maintaining a clean code base from the very beginning, as it will grow significantly in the following homeworks.

Peer communication

Each peer gets in its configuration argument (from the `NewPeer` function) a socket object responsible for the communication between peers. As illustrated in figure 4, sockets enable peers to communicate, and is actually for them the only way to interact with the outside world.

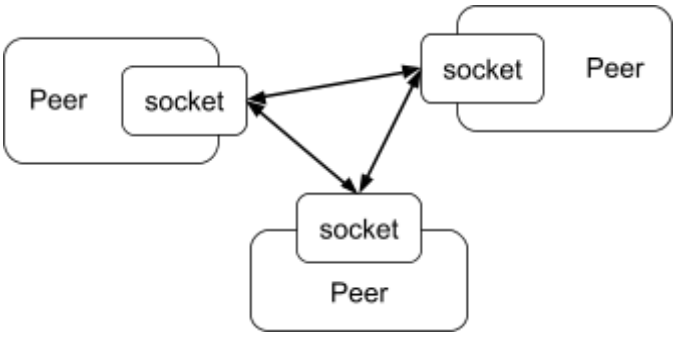

Figure 4: Three peers are exchanging messages using the socket they got in their configuration argument. Sockets are the only way for peers to communicate with the outside world.

A socket is yet another abstraction that is defined in transport/mod.go. Having an abstraction means that a peer doesn't need to worry about how data is sent/received. It can just use the functions provided by the socket, and let the underlying implementation do the rest. Indeed, a socket could use different implementations, like TCP, UDP, local in-memory, etc… A socket uses the notion of an address. The format of the address depends on the implementation. For example, a UDP based socket would use an ip + port combination.

In your implementation, you will have to use this socket object to send and receive messages from/to the outside world.

REST API and CLI

We provide a REST API[2] wrapper for your peer that allows you to use a peer with HTTP requests. It makes it possible to have a nice web-based GUI. This REST API is a Go http server that can be launched with a CLI. Figure 5 shows how the CLI and HTTP server are bundled together around the peer implementation. Figure 5.2 shows the same but with a different angle.

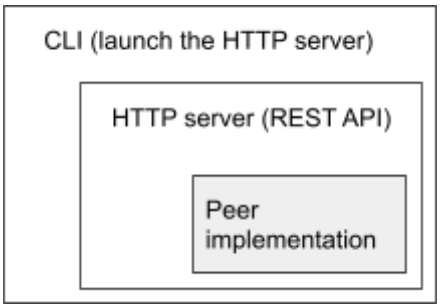

Figure 5: We provide an HTTP wrapper that offers a REST API for a peer. This wrapper is launched by a CLI.

[2] If you are not familiar with REST we encourage you to document yourself about this concept, but this is not required in the context of this course. Right now you can see it as a way to perform remote actions using HTTP calls.

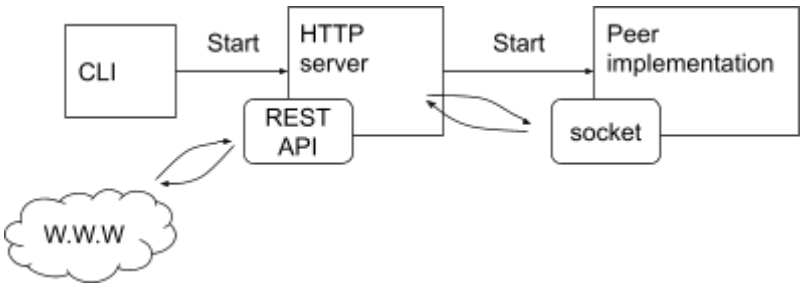

Figure 5.2: Another view of the CLI and HTTP server that offer a REST API for a peer.The HTTP server uses the peer's socket to talk to the peer, based on the call it gets from its HTTP endpoint, which come from anyone that can access the REST API.

The only element that you need to use is the CLI. Which is in **`/gui/mod.go`**. From that folder, you can launch the CLI with

```
go run mod.go start -h
```

the **`-h`** will output the help of the CLI and tell you what arguments can be provided. For HW0 you can just go without providing any argument.

Your tasks

Your first job is to implement the basic start/stop functions of your peer. Then you are going to implement the messaging functionality, and finally your own socket based on UDP.

In summary, here is what needs to be done:

- ☐ Implement the basic peer functionalities (start/stop)
- ☐ Implement a simple routing table
- ☐ Implement a unicast function
- ☐ Implement a chat messaging mechanism
- ☐ Implement a socket that uses UDP

Task 0: Explore the codebase

It is important that you first get familiar with the code base and the different packages it contains. Read the **`README.md`** and **`advices.md`** in **`/docs`** and explore the different folders in the repository. Your work will happen exclusively in the **`/peer/impl/`** and **`/transport/socket/udp/`** folders, but you will need to use elements from other parts. Get familiar with how interfaces work with Go and find out how they are used and implemented in the existing code.

Once you feel you have a good overview of the code base go to the next task.

Task 1: Implement the basic peer functionalities

What we call the “basic” peer functionalities are described by the peer.Service interface in `/peer/service.go`, which is in turn embedded in the peer.Peer interface. Each function is commented with the expected behavior.

You should start by implementing the start and stop functions you can find in the skeleton (`/peer/impl/mod.go`):

```
func (n *node) Start() error {}
func (n *node) Stop() error {}
```

The Start function starts listening on incoming messages with the socket. With the routing table implemented in the next section, you must check if the message is for the node (ie. the packet’s destination equals the node’s socket address), or relay it if necessary. If the packet is for the peer, then the registry must be used to execute the callback associated with the message contained in the packet. If the packet is to be relayed, be sure to update the `RelayedBy` field of the packet’s header to the peer’s socket address.

Have a look at the design documentation if you are not sure how to use the message registry.

Note that the Start function must not block indefinitely, and return once the peer is ready to be used. This means that you will have to use a non-blocking Goroutine to listen to messages on the socket, with a loop. If not, the Start function would block forever. Listing 2 shows an example of using such Goroutine. This routine is where your peer will receive packets and process them.

```
go func() {
  for {
    pkt, err := s.conf.Socket.Recv(time.Second * 1)
    if errors.Is(err, transport.TimeoutErr(0)) {
      continue
    }
    // do something with the packet and the err
  }
}
```

Listing 2: Example using a Goroutine to listen to new packets on the socket. Note that this loop should exit once the Stop function is called, which is not implemented in this listing.

Task 2: Implement a simple routing table

Implement the `AddPeer`, `GetRoutingTable`, and `SetRoutingEntry` functions defined in the `peer.Messaging` interface from `/peer/messaging.go`. Be careful with concurrent read/write, as maps in Go are not thread-safe. Have a look at the tips document if you are not familiar with synchronisation mechanisms in Go.

The routing table is used to get the next-hop of a packet based on its destination. It should be used each time a peer receives a packet on its socket, and updated with the `AddPeer` and `SetRoutingEntry` functions. The sequence field is not relevant for this homework and will be introduced in the next homework.

Task 3: Implement a unicast function

Implement the Unicast function defined in the Messaging interface. This function is responsible for building a packet and using the socket to send that packet to the specified address.

To set the correct values for the Header, you must use the `NewHeader` function from `transport/mod.go`. The provided TTL value must be set to 0.

Task 4: Implement a chat messaging mechanism

Your last job is to implement your first callback for the `ChatMessage` message. This message is defined in `/types/messaging.go`. When the peer receives a ChatMessage, the peer should log that message. Nothing else needs to be done. The `ChatMessage` is parsed by the web-frontend using the message registry.

Do not forget to use the message registry to register your callback function.

Task 5: Implement a socket based on UDP

A socket is a high-level interface that allows the exchange of messages between peers. You will find the socket abstraction in `/transport/mod.go`. In `transport/channel` you will find a socket implementation that is used in the test. It uses local in-memory channels. Your implementation must be in `/transport/udp`, where you will find a skeleton to work on.

The only assumption we make is that there is a

```
NewUDP() transport.Transport
```

function available in that package. Otherwise, you are free to organise your implementation as you want, as long as it implements the socket interface. We recommend you to use Go's networking library.

How to test your program

You have multiple ways of testing your program. We provide automatic testing, as well as a web-based GUI to play around with your implementation.

Automatic testing

Automatic tests are written in /peer/tests. There you can find two folders containing each unit and integration tests. To execute a single test, go to the folder holding the tests and run the following command:

```
go test -v -race -run Test_HW0_Network_Listen_Close
```

Note that go will execute every function that starts with the specified `-run` argument. Hence, to run all HW0 tests, you can execute:

```
go test -v -race -run Test_HW0_
```

You can also run all tests from a folder and its subfolders:

```
go test -v -race ./...
```

Integration tests mix your peer implementation with a reference one, ensuring the interoperability of your implementation. The reference implementation is contained in a binary placed in /peer/tests/integration/node. This binary will be given to you and already placed.

Finally, there is a Makefile in the root directory that ultimately allows you to launch the tests.

Note that for your convenience in implementation you can inspect the tests and change them to debug your program. When we test your program we use the original version of the tests, so be sure at the end that the original tests pass. Also, while you are free to modify everything in the code skeleton for your convenience, our tests will use the original skeleton files outside the `peer/impl` and `transport/udp` folders.

Manual testing

We provide a frontend to play around with your implementation. This frontend is implemented in /gui. You can run it from that folder with the following command:

```
go run mod.go start
```

This command will start your peer with a REST API proxy on top of it. The API address (called proxy address) and peer's address are randomly chosen and displayed in the logs. Note that you can also specify those addresses in the CLI. Once your peer and its proxy are running, you can open the file /gui/web/index.html and type the proxy's address. From there you can try to run multiple proxies and exchange messages. Figure 6 illustrates the web frontend you should see once you can successfully exchange chat messages between peers.

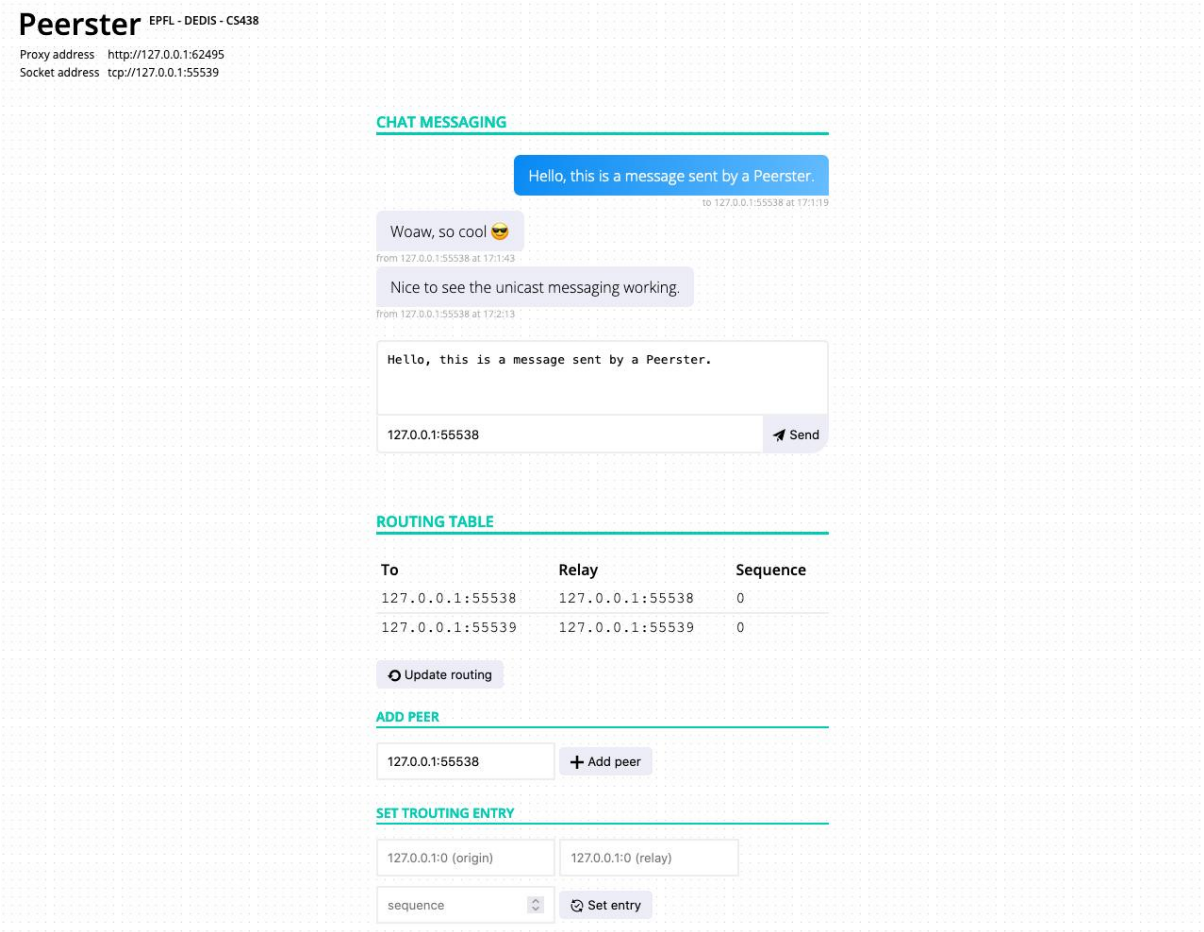

Figure 6: Illustration of the web front end exchanging chat messages.

Automatic testing through GitLab

GitLab is configured to automatically run the tests each time changes are committed.

Hand-in Procedure and Grading

For each homework, you will receive a grade out of 6. 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 each homework assignment document. **You can always update your submission on moodle until the deadline, so please start submitting early. Late submissions are not possible.**

We grade your solutions via a combination of automatic testing, code inspection, and code plagiarism detection. We run automated tests on your implementation and make sure it works as required, both when communicating with other instances of itself (unit tests) and when communicating with the reference implementation (integration tests). You are provided with the test framework, so that you can test your code yourself before submitting. 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.

For the actual grading, we use the same tests but with *different* input data so only the implementations that correctly implement all the functionality will pass. We also use a few hidden tests that test the scalability of your implementation in a large system (e.g., 20 nodes). Thus, your implementation must be *reasonably efficient*, e.g., by processing messages in parallel instead of one-at-a-time. We strongly encourage you to test your implementation with the implementations of your classmates by having them communicate with each other.

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 don't reduce 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.

Appendix A - Tests definitions

Note to authors: Do not directly modify the test definitions without updating the implementation. Leave a comment if needed.

Socket implementation

Socket creation/close

Pre-condition	**Action**	**Expectation**
(0-1) A socket factory using UDP.	A socket is created with “fake” as an address.	The CreateSocket returns an error.
(0-1) A socket factory using UDP.	A socket is created with “127.0.0.1:0” as address.	The CreateSocket doesn’t return an error. The GetAddress() of the socket doesn’t return “127.0.0.1:0”.
(0-1) A socket using UDP.	Close the socket.	The close shouldn’t return an error.
(0-1) A socket factory that already created and closed a socket.	A socket is created with “127.0.0.1:0” as address.	The CreateSocket shouldn’t return an error. The close shouldn’t return an error.

Simple send/recv

Pre-condition	**Action**	**Expectation**

(0-2) Two sockets, S1 and S2, were created with two UDP factories.	S1 sends a random packet to S2. S2 calls Recv(). S1 sends a random packet to S2. S2 calls Recv(). S1 sends a random packet to S1.	S2 should receive the packet from S1. No errors should be returned. S1 should receive the packet from S2. No errors should be returned. S1 should receive the packet from itself with no errors.

Multiple send/recv

Pre-condition	Action	Expectation
(0-3) Two sockets, S1 and S2, were created with two UDP factories. S1 and S2 are listening on incoming messages and storing them in R1, R2 respectively.	Start a goroutine that makes S1 send 10 packets. Do the same for S2. Wait for the sending goroutines to be done. Sleep 1 second.	R1 should contain all the messages sent by S2. R2 should contain all the messages sent by S1.

Peer implementation

AddPeer function

Pre-condition	Action	Expectation
(0-4) A peer P1 is started.	Get routing table R1 from P1.	R1 should contain one element, the peer's address and relay to itself.

	Call AddPeer() on P1 with P1's address as argument. Get routing table R1 from P1. Call AddPeer() on P1 with A1, A2 as argument. Get routing table R1 from P1.	R1 should not have been updated and be of length 1. R1 should be of length 3 and contain A1, A2

SetRoutingEntry

Pre-condition	**Action**	**Expectation**
(0-9) A peer P1	Call SetRoutingEntry(A, B) on P1 Call SetRoutingEntry(A, C) on P1 Call SetRoutingEntry(A, "") on P1	P1 should have one routing entry to itself P1 should have two routing entries, one to itself and one A -> B P1 should have two routing entries, one to itself and one A -> C P1 should have one routing entry to itself

RegisterNotify

Pre-condition	**Action**	**Expectation**
(0-5) A peer P1 is started. P1 registered the random packet callback.	Call RegisterNotify on the message registry with C1 as argument. Send a random packet to P1.	P1 called C1. P1 called the random packet callback.

Unicast function

Pre-condition	Action	Expectation
(0-5) Two peers P1, P2, started. P2 registered the random packet callback. P1 added P2 with AddPeer.	Call Unicast on P1 to P2's address with a random packet.	P1's socket should've sent the random packet. P2's socket should've not sent any packet. P2's socket should've received the random packet. P2 should've called the random packet callback.
(0-6) Two peers P1, P2, started.	Call Unicast on P1 to P2's address.	P1's address should return an error since it doesn't know about P2. P1 should've not sent any packet. P2 should've not received any packet.

Relays of packet

Pre-condition	Action	Expectation
(0-7) Three peers P1, P2, P3 started. P1 knows P2. P2 knows P3. P1 knows it can relay messages to P3 via P2, using SetRoutingEntry. P3 has registered the random packet callback.	Call Unicast on P1 to P3's address with a random packet. Sleep 1 second.	P1 didn't' receive any packet. P2 received a packet from P1. P2 sent a packet to P3. P3 received P1's packet from P2. RelayAddress of the packet is P2's address and source is P1's. P3 didnt' send any packet.

		P3 called the random packet callback.

Integration tests

Unicast

Pre-condition	**Action**	**Expectation**
(0-8) 10 reference peers started. 10 student peers started.	Every peer sends a message to every other peer, including itself.	Every peer sent 20 packets. Every peer received 20 packets.