

Introduction + Homework 0

Network Programming in Go

Author	DEDIS lab, EPFL
Revision	September 2023 - 1.2.0
Publish date	Friday, September 22, 2023
Due date	Tuesday, October 10, 2023 @ 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
Distributed systems, defensive programming, and quality	4
General structure of homeworks	5
GitHub Classroom	6
Git setup	6
Continuous integration (CI)	7
Programming environment	8
Logging, Performance and Benchmarks	8
Homework 0	9
End result	9
Objectives	9
Peer communication	9
REST API and CLI	10
Your tasks	11
Task 0: Explore the codebase	11
Task 1: Implement the basic peer functionalities	12
Task 2: Implement a simple routing table	12
Task 3: Implement a unicast function	13
Task 4: Implement a chat messaging mechanism	13
Task 5: Implement a socket based on UDP	13
How to test your program	14
Automatic testing	14
Manual testing	14
Automatic testing through GitHub	15
Hand-in Procedure and Grading	16
Appendix A - Tests definitions	17
Socket implementation	17
Socket creation/close	17
Simple send/recv	17
Multiple send/recv	18
Peer implementation	18
AddPeer function	18

SetRoutingEntry	19
RegisterNotify	19
Unicast function	20
Relays of packet	20
Integration tests	21
Unicast	21

Introduction to all the CS438 homeworks

Gossip protocols

Gossip protocols are distributed protocols for robust information exchange. Those protocols are typically deployed networks with high churn. Example 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 gossiping 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.

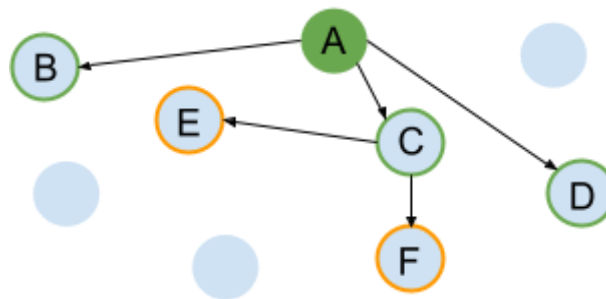


Figure 1: A gossip protocol in action. A is the rumor starter and sends a rumor to its known addresses B, C, D. Node C will then spread the rumor it received from A to its known addresses E, F. C will also learn the address of node A.

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 gossipier¹. 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 in the manner of BitTorrent, and finishing with an implementation to reach consensus on file names.

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

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.

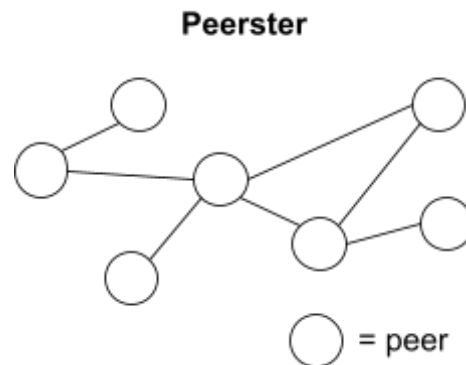


Figure 2: Global view of the Peerster system, which is made of connected peers that can exchange messages. Each peer in Peerster acts as a gossipier. Your job is to implement a peer that follows the provided specification.

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. You can also use the Moodle forum to ask questions to the TAs and discuss among students.

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 work diligently and make sure you fully implement every homework assignment. If it so happens that you were unable to finish an assignment in due time, the grading still rewards you for improving it later on. For more information on this, see the information sheet on grading.

Workload

This is a software systems course and you are expected to build a working system. On average, you should be spending 8-12 hours per week on the homework and project. This is an average over the semester, and while initial weeks might seem lighter, you can expect later homework to be more difficult. As such, try to manage your time accordingly and tackle the work early on.

As the homework can be quite complex, and bugs such as concurrency issues can prove time consuming to debug, try to resist the temptation of getting a perfect score without regard for the work it entails. Instead of aiming for a perfect score as if you had infinite time available, try to aim for a grade matching your abilities within the allotted time. You should ask yourself: how many extra hours are you willing to spend on this class for a (small) grade improvement ? To better answer this question, [we provide a detailed explanation of the grading scheme for the class](#).

To save yourself extra work: make sure you understand the core concepts from class, take the time to read the homework thoroughly, then design what your solution will look like ahead of implementing it.

Prerequisites

The major prerequisite for this project is a good understanding of the Go language. If you are familiar with a language like C and an object-oriented programming language like Java, then you should be able to quickly get familiar with the Go language.

If you are not familiar with Go, we suggest you first complete [the Tour of Go tutorial](#).

Furthermore, here is a list of topics with Go that we advise you to get **strongly** familiar with:

- Go concurrency primitives ([routines, channels, select, etc.](#))
- Composition vs. inheritance ([embedding](#))
- The synchronization primitives in the [sync package](#).

Distributed systems, defensive programming, and quality

As part of this course, you are going to be building a distributed system. Distributed systems are *hard*. They have, by definition, many moving parts and any of them can go wrong, which makes both their development and debugging particularly complex. Things “going wrong” can happen unintentionally, due to bugs, network issues, protocol ambiguity, etc., or it can be intentional, as part of a cyber attack.

We obviously don’t expect you to build secure, safety-critical systems, but **we will be assessing whether your system can withstand a few things “going wrong”**. In any case, we will always explicitly indicate when it is safe for you to make assumptions.

To achieve a degree of robustness, we *strongly* encourage you to adopt a *defensive programming* approach. This will allow you to detect bugs earlier in the development, and it will also ensure that your system performs reliably even in the face of network issues, invalid data, and scenarios that were not explicitly tested. Furthermore, it will make it easier for you to detect bugs during development.

Defensive programming is a vast subject (feel free to read up on it!), but it does entail at the very least a few things:

- Validating all inputs to your software system: user input, network messages, etc.
- Adding guard clauses / checking pre-conditions: validating your data before performing operations on it
- Explicitly handling all errors raised by the functions you call
- A slightly paranoid attitude, anticipating what could go wrong
- Writing clean / simple enough code that it is either obviously correct or broken

To help with your debugging endeavors, we also suggest that you make abundant use of logging, using logging levels as needed, to understand the runtime behavior of your software. This will be essential when you see bugs appearing on systems’ other than your own computer, such as those arising in the continuous integration environment, where starting the debugger isn’t an option. However, be mindful that **you need to support a log-free mode** (c.f. “Logging, Performance and Benchmarks”).

Furthermore, the project is provided to you pre-configured with tools that check the software quality, spot obvious bugs, warn of excessive code complexity and verify that you respect Go best practices. This will support your software engineering effort and a subset of **static analysis results will also contribute to your homework assessment**. For this reason, **you are not allowed to modify the static analysis configuration or to disable errors inline**.

General structure of homeworks

You are going to start with a skeleton of code that you build on. The skeleton can be found on GitHub, in the [2023-hw-skeleton](#) repository (each branch corresponding to a specific homework). The skeleton entry point can be found in `/peer/impl/impl.go`. It's in this `impl` package that you are going to implement your peer.

We provide an abstraction of a peer, i.e. 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/peer.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` and `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.

GitHub Classroom

As part of this class, you will be using GitHub to host your code and work on it. If you don't have one already, please create a GitHub account. You will be provided with a GitHub repository for each homework, that you can access *after* accepting the GitHub Classroom Assignment.

For each homework, please proceed as follow:

1. First accept the assignment at the link provided with each homework, which will look like `https://classroom.github.com/a/{UNIQUE_CODE}`. This step will automatically create a repository for you.
2. Once accepted, wait a little bit to refresh the page and see a link to access the Git repository that was created for you, named with the following convention: `https://github.com/cs438-epfl/{ASSIGNMENT_NAME}-{GITHUB_USER}`.

Please note that you are not allowed to publish your code on any publicly accessible service without the written consent of the lab, even once the semester is over !

Git setup

To start with, clone the homework skeleton locally:

```
git clone --origin skeleton git@github.com:cs438-epfl/2023-hw-skeleton.git
```

The `--origin skeleton` argument names the Git remote "skeleton" instead of "origin". This makes its usage much clearer later on, as well as when working with other remotes.

Now, to set up a homework assignment locally, follow the next 3 steps inside the repository's directory. These can be repeated for new homeworks, by changing `hw0` into `hw1`, `hw2`, etc...

1. Add your homework repository as a [Git remote](#):

```
git remote add hw0-repo git@github.com:cs438-epfl/2023-hw0-{USERNAME}.git
```

2. Then create a new branch:

```
git checkout -b hw0
```

3. Integrate the homework skeleton into the newly created branch

```
git pull skeleton hw0
```

4. Finally set up the upstream repo for the newly created branch

```
git push --set-upstream hw0-repo hw0
```

At this point, you are ready to start working on your homework and can commit and push the changes to the correct repository.

In any case, please be mindful of where your code is pushed throughout the semester. You will be graded based on the content of your homework repository on the deadline.

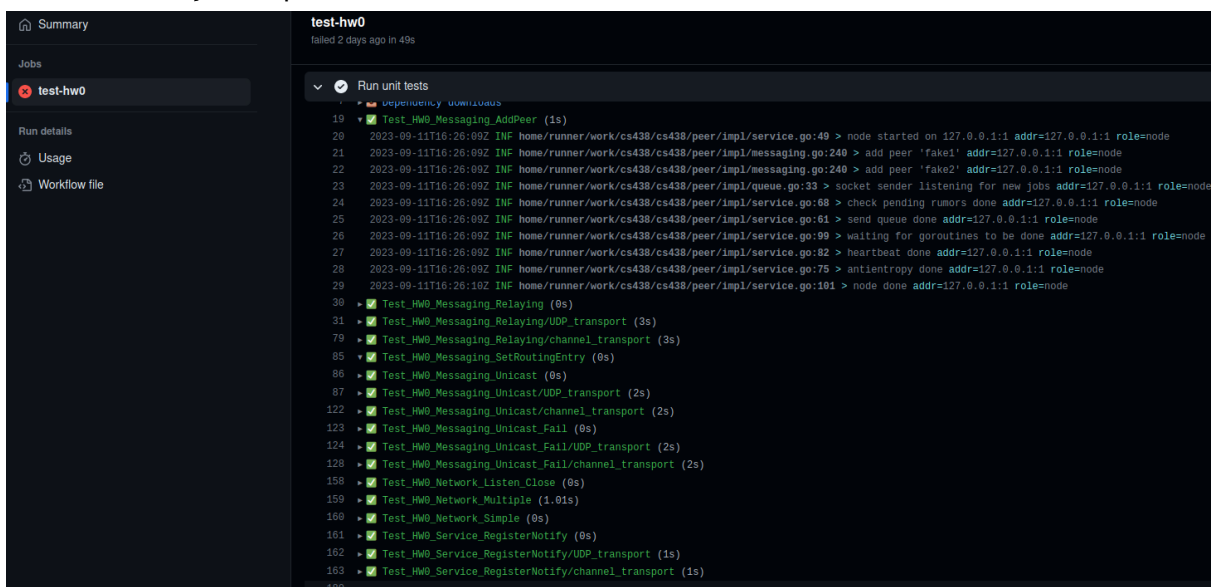
Code that does not compile or was pushed to the wrong repository will not be graded.

Continuous integration (CI)

A continuous integration environment is provided to you to help you catch bugs during your development. You should feel free to add any tests you deem necessary to ensure that your implementation behaves correctly, though they won't be taken into account when grading.

Note that the CI is not supposed to replace local testing, debugging and other quality efforts. The CI provides a fixed environment with its own computational resources and timings. By testing both locally and on the CI, you significantly increase your chances of catching bugs that are related to concurrency and timing issues.

The CI runs automatically when you push new code in your repository. No pull request is required, as this is a personal repository. The results of the tests can be found in the GitHub Actions tab of your repo, and look like this:



The screenshot shows the GitHub Actions interface for a workflow named 'test-hw0'. The left sidebar contains links for 'Summary', 'Jobs', 'Run details', 'Usage', and 'Workflow file'. The 'Jobs' section shows 'test-hw0' with a failed status. The main panel displays the 'Run unit tests' job, which failed 2 days ago in 49s. The log shows a series of successful test runs for 'Test_HW0_Messaging' and 'Test_HW0_Network', followed by a failure for 'Test_HW0_Network_Listen_Close' at line 158. The failure message is 'panic: to be implemented in HW0 [recovered]'.

```
Summary
Jobs
test-hw0
Run details
Usage
Workflow file

test-hw0
failed 2 days ago in 49s

Run unit tests
- dependency download
19 Test_HW0_Messaging_AddPeer (1s)
20 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:49 > node started on 127.0.0.1:1 addr=127.0.0.1:1 role=node
21 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/messaging.go:240 > add peer 'fake1' addr=127.0.0.1:1 role=node
22 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/messaging.go:240 > add peer 'fake2' addr=127.0.0.1:1 role=node
23 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/queue.go:33 > socket sender listening for new jobs addr=127.0.0.1:1 role=node
24 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:68 > check pending rumors done addr=127.0.0.1:1 role=node
25 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:61 > send queue done addr=127.0.0.1:1 role=node
26 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:99 > waiting for goroutines to be done addr=127.0.0.1:1 role=node
27 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:82 > heartbeat done addr=127.0.0.1:1 role=node
28 2023-09-11T16:26:00Z INF home/runner/work/cs438/cs438/peer/impl/service.go:75 > antientropy done addr=127.0.0.1:1 role=node
29 2023-09-11T16:26:10Z INF home/runner/work/cs438/cs438/peer/impl/service.go:101 > node done addr=127.0.0.1:1 role=node
30 Test_HW0_Messaging_Relaying (0s)
31 Test_HW0_Messaging_Relaying_UDP_transport (3s)
79 Test_HW0_Messaging_Relaying_channel_transport (3s)
85 Test_HW0_Messaging_SetRoutingEntry (0s)
86 Test_HW0_Messaging_Unicast (0s)
87 Test_HW0_Messaging_Unicast_UDP_transport (2s)
122 Test_HW0_Messaging_Unicast_channel_transport (2s)
123 Test_HW0_Messaging_Unicast_Fail (0s)
124 Test_HW0_Messaging_Unicast_Fail_UDP_transport (2s)
128 Test_HW0_Messaging_Unicast_Fail_channel_transport (2s)
158 Test_HW0_Network_Listen_Close (0s)
159 Test_HW0_Network_Multiple (1.01s)
160 Test_HW0_Network_Simple (0s)
161 Test_HW0_Service_RegisterNotify (0s)
162 Test_HW0_Service_RegisterNotify_UDP_transport (1s)
163 Test_HW0_Service_RegisterNotify_channel_transport (1s)
168
```

Example of a failed test:



The screenshot shows a failed test in the CI environment. The test is 'Test_HW0_Network_Listen_Close' and it failed at line 158. The failure message is 'panic: to be implemented in HW0 [recovered]'. The test is part of a workflow named 'Run unit tests' which took 27s to complete.

```
Run unit tests 27s
1 ▶ Run set -euo pipefail
7 ▶ Dependency downloads
14 ▼ Test_HW0_Network_Listen_Close (0s)
15   panic: to be implemented in HW0 [recovered]
16     panic: to be implemented in HW0
17
```

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).

Multiple IDEs exist for the Go language:

- [JetBrains' GoLand](#), by far the most advanced and feature-complete. You can obtain a [student license](#) for it with your EPFL e-mail.
- Visual Studio Code provides [good support for Go](#)
- Vim and vim-based editors can be used, but do make sure that you have a [strong](#) command of the Go debugger in your environment, if you choose to go this route.

If you're on a Windows-based machine, your IDE can be configured to run the tests in the Windows subsystem for Linux (WSL) environment. Look up the documentation !

In term of software packages, here are the dependencies:

- Go 1.21
- make (if you want to use the makefile)

Logging, Performance and Benchmarks

As part of the evaluation of your homework (starting with Homework 1), **the performance of your solution will also be evaluated**. As such, we expect your code to have a completely log-free mode, so that the system's execution is not artificially slowed down by generating or displaying the logs.

We ask that you support the log-free mode by disabling logging when the environment variable `GLOG` is set to `no`. You can test this setting on any UNIX-compatible system by prepending your command with "`GLOG=no` " (including whitespace). On a Windows system using Powershell, you might want to set the environment variable by issuing the following command "`$env:GLOG = 'no'`" ahead of the test/benchmark execution.

You can take example on how the `/internal/binnode` implementation handles a similar requirement with the `BINLOG` environment variable.

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.

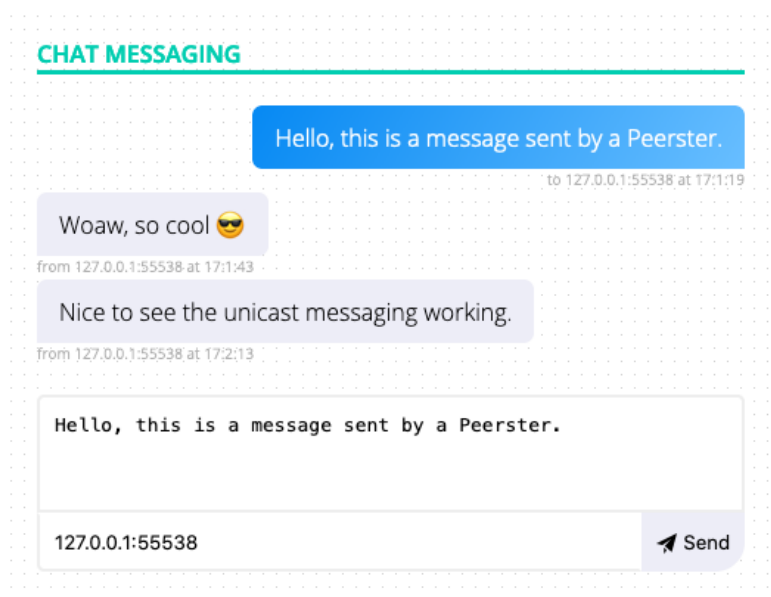


Figure 3: Minimalistic web-based GUI to play around with the chat functionality that HW0 implements. This is the end result of HW0.

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), the Peerster architecture, the 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.

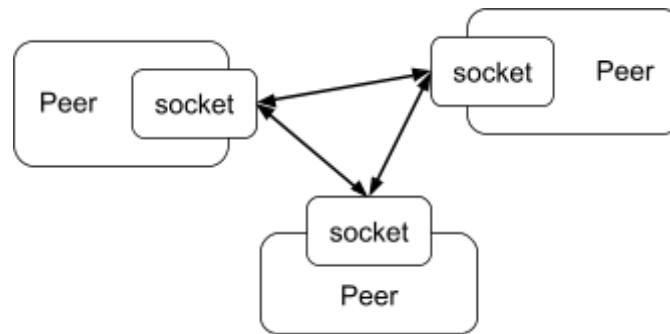


Figure 4: Three peers are exchanging messages using the socket they got in their configuration argument. Sockets are the only way for peers to communicate with the outside world.

A socket is yet another abstraction that is defined in `transport/transport.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 API² 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.

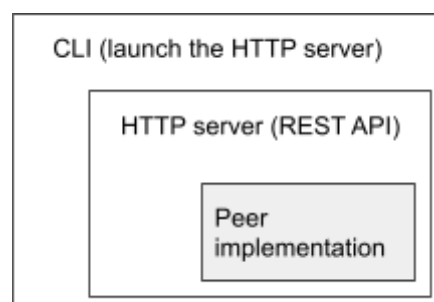


Figure 5: We provide an HTTP wrapper that offers a REST API for a peer. This wrapper is launched by a CLI.

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.

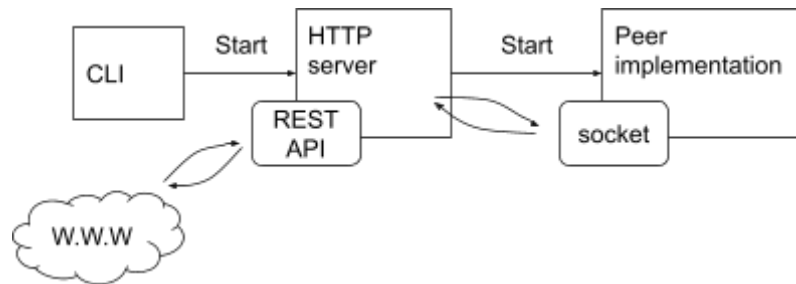


Figure 5.2: Another view of the CLI and HTTP server that offer a REST API for a peer. The HTTP server uses the peer's socket to talk to the peer, based on the call it gets from its HTTP endpoint, which come from anyone that can access the REST API.

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

```
go run gui.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 behaviour.

You should start by implementing the start and stop functions you can find in the skeleton (`/peer/impl/impl.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 (i.e. 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 (in `docs/README.md`) 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.TimeoutError(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.

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/transport.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/transport.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 gui.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.

Peerster EPFL - DEDIS - CS438

Proxy address http://127.0.0.1:62457
Peer address tcp://127.0.0.1:61480

MESSAGES RECEIVED ON THE CHAT ↑

Hello, this is a message sent by a peer 🍌

to 127.0.0.1:60982 at 17:15:7

Hi mate !

from 127.0.0.1:60982 at 17:15:48

So cool to see the unicast messaging
working 😊

from 127.0.0.1:60982 at 17:16:20

UNICAST ↑

CHATMESSAGE

Hello, this is a message sent by a peer 🍌

127.0.0.1:60982

Send

ROUTING TABLE ↑

To	Relay
127.0.0.1:60982	127.0.0.1:60982
127.0.0.1:61480	127.0.0.1:61480

See graph

Update routing

ADD PEER

127.0.0.1:60982

+ Add peer

SET TROUTING ENTRY

127.0.0.1:0 (origin)

127.0.0.1:0 (relay)

Set entry

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

Automatic testing through GitHub

GitHub 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 submit your fully-working code on GitHub by the due date, which you can find at the beginning of each homework assignment document. **You can always update your submission on GitHub until the deadline, so please start submitting early. Late submissions are not possible.**

Also please remember that you need to support log-free mode for benchmarking purposes, when the GLOG environment variable is set to no, as per the section "Logging, Performance and Benchmarks".

We grade your solutions via a combination of automatic testing, static analysis, 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 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 different properties of your system (scalability, robustness, etc.). Thus, your implementation must be *reasonably efficient and robust*, e.g., by processing messages in parallel instead of one-at-a-time, by validating inputs, etc. We strongly encourage you to test your implementation with the implementations of your classmates by having them communicate with each other. Finally, we also take into account a subset of the static analysis output to ensure that your code is reasonably clean and well-designed.

Our very first test is that your code must build. **No points will be given if your code does not compile and build.**

As a final recommendation, 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

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. S1 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.

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

SetRoutingEntry

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

RegisterNotify

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

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 didn't 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.