Homework 0
Network Programming in GO

CS-438 - Decentralized Systems Engineering, Fall 2020

Publish date: Friday, September 18, 2020
Due date: Tuesday, October 6, 2020 @ 23:55

General Guidelines to the Course Homeworks

In this course, you will build a peer-to-peer application in the **Go programming language** called ***Peerster***. With every homework, Peerster's functionality will become richer, starting with a simple and imperfect message broadcast, to gossip messaging, routing for one-to-one communication, file sharing à la BitTorrent, and finishing with an implementation to reach consensus on file names.

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 other homeworks 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 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, we will provide you with the skeleton files that contain the main interfaces to be implemented, test framework, and some useful functionality and implementation hints.

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. 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, we offer you an alternative. After each homework, you will receive 3 random anonymized submissions of your colleagues that you need to review (your reviews will be graded), 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 (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 3-10 hours per week on homeworks / project. It is, however, difficult to assess how time consuming bugs or concurrency issues will prove to be for each individual.

Prerequisites

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

Introduction

One core challenge in decentralized systems is ensuring efficient and scalable network communication. In this course, you will learn techniques to address this challenge and this homework takes a very first step in that direction. By the end of homework 0, peers will be able to broadcast messages through the network, albeit in an imperfect way. Indeed, messages can be lost or can loop around the network indefinitely, but that's ok, because you'll have the opportunity to fix this unreliable and inefficient broadcast in Homework 1. For now, the goal of this assignment is to get you familiar with Go in general and Go networking in particular.

Peerster Design

Peerster is a decentralized application, where several nodes, referred to as **gossipers**, communicate with each other. Each Gossiper **implements the functionality for network communication.** A Gossiper communicates with:

1) A client application over TCP (Transport Control Protocol). The client runs on the same machine as the gossiper.
2) Other Gossipers over UDP (User Datagram Protocol). These gossipers run on other machines, but for testing purposes you can also spawn local ones.

The Gossiper **exposes an API to web-based clients** that enables a client to:

1) Send messages to the local gossiper
2) Retrieve received messages from the local gossiper
3) Add Gossiper identifier for the local gossiper
4) Retrieve Gossiper Identifier from the local gossiper
5) Add connection details of another Gossiper
6) Receive connection details about other Gossipers

The **CLI client** is simpler than the web-based one; it can only send messages to the local gossiper.

The Gossiper listens to two transport layer ports: 1) UIPort and 2) GossipPort. Communication between the client and the Gossiper uses the UIport, whereas the communication between the Gossiper and other Gossipers uses the GossipPort. Figure 1 depicts this design.

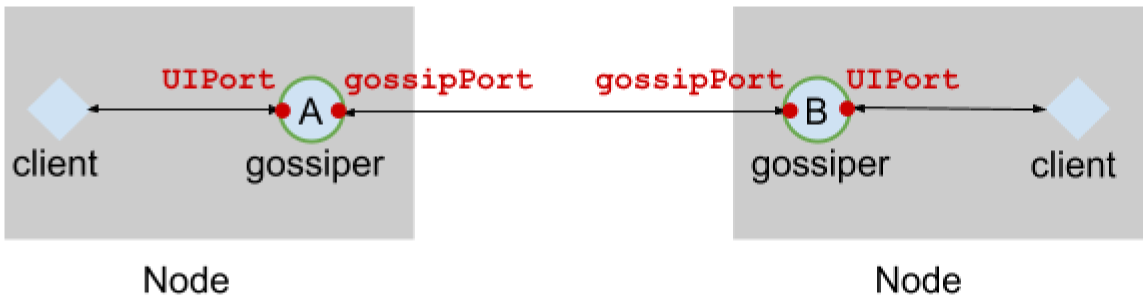

Figure 1

Your task

In this homework you will build the gossiper and the command line interface (CLI) client. We provide you with a web-based client, which you can use to test your Gossiper.

Network Programming in Go

Gossipers communicate with each other over UDP. You can use Go’s networking library. Your **Gossiper program** takes the following as the arguments.

1. The `UIPort`
2. The `GossipIP:GossipPort` (if you run in the localhost, GossipIP is `127.0.0.1`)
3. `Name` (name of the node)
4. List of one or more connection details of other gossipers, in the form `ip1:port1,ip2:port2`,etc (note that list elements are separated by a comma).

You must follow this input format in order to comply with automatic testing, otherwise tests will fail.

Below is an example of how to run a compiled gossiper

```
./peerster -UIPort=10000 -gossipAddr=127.0.0.1:5000 -name=nodeA
       -peers=127.0.0.1:5001,10.1.1.7:5002
```

Clients talk to gossipers over TCP. You can use Go's networking library. Your `local CLI client program` takes two arguments:

1. `UIPort` which is the port on which the gossiper listens for the messages that are sent by the client
2. `msg` which is the message sent to the gossiper. The msg is a JSON-encoded message (not plain text), described in the next section.

Following is how a compiled client sends a message to the gossiper.

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

Message formats

- **Client to Gossiper** message format

  JSON format: `{contents:<value>}`

  `ClientMessage` is the go struct that is used for Client to Gossiper communication, already defined in the code skeleton.

- **Gossiper to Gossiper** message format

  JSON format: `{originPeerName:<value>, relayPeerAddr:<value>, contents:<value>}`

  `SimpleMessage` is the structure of the messages that is used for Gossiper to Gossiper messaging, already defined in the code skeleton.

Moreover, we define a `GossipPacket`, the **ONLY** type of packets sent to other peers. In next homework, we will add other message types to it but for now it may only contain a `SimpleMessage`.

Message forwarding logic

We illustrate the message forwarding logic as follows.

When a `gossiper` receives a message (in simple broadcast mode, which is the only mode in this homework):

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

For message processing, the code skeleton that we provide enables you to register handlers. A handler is a function that can process a certain type of message. For each message type the Gossiper receives from other Gossipers (`SimpleMessage` in this homework), you should register a handler. In this homework, your task is to write the handler to process a `SimpleMessage` type. The handler must be called `Exec` and it is defined on the `SimpleMessage` struct. The signature of the handler is already in the code skeleton.

Printing format in the standard output

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.

      or

    - **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**

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 need to first convert the message to an architecture-independent format. Also upon receiving a message, the node should deserialize the message, before processing. For serializing and deserializing, you will use JSON, a standard approach for encoding messages on the internet. Go provides a library for JSON encoding.

How to manually test your program

Run multiple instances of the program on your local machine (use 127.0.0.1 as the IP with different ports) 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.

Automatic testing through GitLab

We will use GitLab for testing. Each of you will be provided with a GitLab repository that you can access with your credentials for gitlab.epfl.ch. The CI (continuous integration) tool will automatically run tests every time you push your code in the repo. For your convenience in implementation, you can inspect the tests and change them to debug your program. Unit tests are already available and integration tests will be available soon.

We will provide you with the basic code structure with interfaces, on top of which you can write your code. In order for your code to work with the testing infrastructure, ***we strongly advise you not to change the project structure***.

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 will grade your solutions via a combination of automatic testing, 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 (unit tests) and when communicating with our own instances (integration tests). You will be 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 will use the same tests but with *different* input data so only the implementations that correctly implement all the functionality will pass. We will 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 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.

------------------------------------This completes the homework---