Exercise in subject:	**DAT520 Distributed Systems**
Title:	**Lab 2: Networking programming in Go**
Deadline:	**Wedensday Jan 22 2013 23:59**
Expected effort:	**10 hours**
Remarks:	Links in this document are clickable.

University of Stavanger

Faculty of Science and Technology

1 Introduction

The goal of this lab exercise is to get you started with network programming in Go. The overall aim of the lab project is to implement a distributed fault tolerant application. Knowledge of network programming in Go is naturally a prerequisite for accomplishing this. This lab exercise consist of three parts. Part 2 introduces some background material. Part 3 and 4 include implementing a simple key-value store using both TCP and RPC.

The most important packages in the Go standard library you will use in this assignment is the `net` and `net/rpc` package. It is recommended that you actively use the documentation available for the packages during your work on this lab exercise. Another good source of information is Jan Newmarch's book *Network programming with Go*, available at http://jan.newmarch.name/go/.

Handin: Use the procedure explained during the lab hours. A written summary is also available on itslearning.

2 Background

You should read chapters 1, 3, 4 and 13 of Jan Newmarch's book. Large parts of Chapter 1 should already be familiar knowledge from an introductory network course.

3 Client-Sever Key-Value Store using TCP

In this part of the exercise you are going to implement a client-server key-value store. The server holds a map where both keys and values are regular strings. A client can issue read or write requests to the server and its map. The clients and the server should use the TCP protocol for network communication. You can run both clients and the server on the same machine to test your code.

The skeleton code available for the client and server implementation is shown in Listing 1 and 2. You should use this code as a starting point. The code is available in the `handouts` repository at GitHub. The code contains the following packages:

- `server-tcp`: The server implementation using the TCP protocol.
- `client-tcp`: The client implementation using the TCP protocol.
- `cmd`: The `Request` and `Response` structs used by both clients and the server. Two kinds of request are supported: Read and Write.

- `kvs`: The `KeyValueStorage` type definition and associated method stubs (these method stubs should only be used with RPC in Part 4).

The TCP client-server implementation should have the following properties:

- The server should be able to handle multiple clients concurrently, i.e. the server should handle each client in a separate goroutine.
- A client issues a request to the server using the `Request` struct in the `cmd` package. The server responds using the `Response` struct in the same package. A request can either be a read or write request. A read request has an empty `Value` field. A write response has an empty `Value` field in response to a write request. For a write response the boolean `OK` field indicates if a key was found in the map. This field should by true in the response of a successful write command. If the server encounters any error it should respond with `OK` set to false and an error message in the `Value` field.
- Both the clients and server should use the gob format to encode their messages. See the `encoding/gob` package for more information.
- The server should synchronize access to its map by using a reader/writer mutual exclusion lock. See the `sync.RWMutex` type in the `sync` package. You can also use channels to achieve this synchronization.
- The command line client should print both the request sent and the response received. You can choose to implement an interactive or hardcoded client. A hardcoded client should do a couple of successive (possibly random) reads and writes to demonstrate your program.

Listing 1: TCP Server

```go
package main

import (
  "log"
  "net"

  "handouts/lab2/kvs"
)

var keyVal = kvs.NewKvs()

func main() {
  if err := run(); err != nil {
    log.Fatalln("Error:", err)
  }
}

func run() error {
  return nil
}

func handleConn(conn net.Conn) {
}
```

Listing 2: TCP Client

```go
package main
```

```
import "log"

func main() {
  if err := run(); err != nil {
    log.Fatalln("Error:", err)
  }
}

func run() error {
  return nil
}
```

Listing 3: Request and Response structures

```
package cmd

import "fmt"

type Operation int

const (
  Read Operation = iota
  Write
)

var Operations = [...]string{
  "Read",
  "Write",
}

func (op Operation) String() string {
  return Operations[op]
}

type Request struct {
  Op    Operation
  Key   string
  Value string
}

type Response struct {
  OK    bool
  Value string
}

func (req Request) String() string {
  switch req.Op {
  case Read:
    return fmt.Sprintf("Read␣request␣for␣key␣%q", req.Key)
  case Write:
    return fmt.Sprintf("Write␣request␣for␣key␣%q␣with␣value␣%q",
      req.Key, req.Value)
  default:
    return "Unkown␣request␣type"
  }
}

func (resp Response) String() string {
  return fmt.Sprintf("Reponse␣with␣OK:␣%t␣and␣value:␣%q",
    resp.OK, resp.Value)
}
```

4 Client-Server Key-value Store using RPC

A very popular way to design distributed applications is by means of remote procedure calls. RPC is explained in Chapter 13 of Jan Newmarch's book. The task for this part of the lab is to implement the client-server key-value described in Part 3 using RPC instead of TCP. The skeleton code you should use is available in the `server-rpc`, `client-rpc` and `kvs` package. The code is shown in Listing 4, 5 and 6. Note that you do not need to explicitly use gob encoding for this task (the `net/rpc` package handles this for you). You will for this part of the exercise also need to implement the two empty method stubs for the `KeyValueStore` type in the `kvs` package.

Listing 4: RPC Server

```
package main

import (
  "log"

  "handouts/lab2/kvs"
)

var keyVal = kvs.NewKvs()

func main() {
  if err := run(); err != nil {
    log.Fatalln("Error:", err)
  }
}

func run() error {
  return nil
}
```

Listing 5: RPC Client

```
package main

import "log"

func main() {
  if err := run(); err != nil {
    log.Fatalln("Error:", err)
  }
}

func run() error {
  return nil
}
```

Listing 6: The KeyValueStorage type

```
package kvs

import (
  "sync"

  "handouts/lab2/cmd"
)

type KeyValueStorage struct {
  Storage map[string]string
  RWLock  sync.RWMutex
}

func NewKvs() *KeyValueStorage {
  return &KeyValueStorage{
    Storage: make(map[string]string),
  }
}

func (kvs *KeyValueStorage) Read(key string, resp *cmd.Response) error {
  return nil
}

func (kvs *KeyValueStorage) Write(req *cmd.Request, resp *cmd.Response) error {
  return nil
}
```