Exercise in subject:	**DAT320 Operating Systems**
Title:	**Lab 5: Processing Channel Zaps**
Deadline:	**Sunday Oct 27 2013 23:59**
Expected effort:	**30 hours**
Remarks:	Links in this document are clickable.

University of Stavanger
Faculty of Science and Technology

1 Introduction

This is the main project in this course. It will take you through some interesting challenges, and hopefully you will be able to use much of the stuff that you have learnt in the previous lab exercises. The project must be written in Go.

2 Collecting Channel Zaps

Imagine that you are working for ZapBox, an Internet and Cable service provider. ZapBox has deployed a huge number of set-top boxes at customer homes that allows them to watch TV over a fiber optic cable. The TV signal is distributed to customers based on a multicast stream for each available TV channel in ZapBox's channel portfolio. Recently, ZapBox commissioned a software update on their deployed set-top boxes. After this software update, the set-top box will send a UDP message to a server every time a user changes the channel on their set-top box. In addition to channel changes, a few other items of interest may also be sent. Thus, a message sent by a set-top box may contain information about either channel changes, volume, mute status, or HDMI status. The content depends on the actions of the different TV viewers. Below is shown a few samples of the message format:

```
2013/07/20, 21:56:13, 252.126.91.56, HDMI_Status: 0
2013/07/20, 21:56:55, 111.229.208.129, MAX, Viasat 4
2013/07/20, 21:57:01, 111.229.208.129, Mute_Status: 0
2013/07/20, 21:57:48, 98.202.244.97, FEM, TVNORGE
2013/07/20, 21:57:44, 12.23.36.158, Canal 9, MAX
2013/07/20, 21:57:46, 81.187.186.219, TV2 Bliss, TV2 Zebra
2013/07/20, 21:57:42, 61.77.4.101, TV2 Film, TV2 Bliss
2013/07/20, 21:57:42, 203.124.29.72, Volume: 50
2013/07/20, 21:57:42, 203.124.29.72, Mute_Status: 0
```

Each line above represents an event, triggered by a single TV viewer's action, either to change the channel on their set-top box, or adjust the volume and so forth. These set-top box events are sent in text format shown above. The fields are separated by comma and have the meaning shown in the table below. Note that the message format with 5 fields represents channel change events, while a message with only 4 fields contains a *status change* in the 4th field, and no 5th field.

	Field No.	Field Name	Description
	1	`Date`	The date that the event was sent.
	2	`Time`	The time that the event was sent.
	3	`IP`	The IPv4 address of the sending set-top box unit.
	4	`FromCh`	The previous channel of the set-top box.
	5	`ToCh`	The new channel of the set-top box.
	4	`StatusChange`	A change in status on the set-top box.

A *StatusChange* may contain one of the following entries:

StatusChange	Value range	Description
`Volume:`	0-100	The volume setting on the set-top box.
`Mute_Status:`	0/1	The mute setting on the set-top box.
`HDMI_Status:`	0/1	The HDMI status of the set-top box indicates whether or not a TV is connected to the set-top box and powered on.

3 Traffic Generator

For the purposes of this lab project, we have built a traffic generator to simulate the set-top box events generated by ZapBox's customer set-top boxes. The traffic generator resends set-top box events loaded from a large dataset obtained from real traffic. The IP addresses have been scrambled and do not represent a real set-top box. The traffic generator works by synchronizing the timestamp obtained from the dataset with the local clock on the simulator machine. The date is not synchronized.

In a real deployment, the traffic would typically be sent from set-top boxes using UDP and received at a single UDP server, where the data can be processed. However, to make the simulator scale to multiple receiver groups (you the students), we have instead set up the traffic generator on a single machine multicasting each set-top box event to a single multicast address.

4 Part 1: Building a Zap Event Processing Server

The objective of this part is to develop a UDP multicast server that will process the events that are sent by the set-top box clients (in our case the traffic generator). The server can run on one of the machines in the Linux lab. Your server should be able to receive UDP packets from the traffic generator using multicast address and port:

224.0.1.130:10000

Note that since the traffic generator is continuously sending out a stream of zap events, it may be difficult to work with this part of the lab on your own machine. The multicast stream is only available on the subnet of the Linux lab. It is therefore recommended that you work on the lab machines, either physically or remotely using ssh.

Heads up!

Before you begin, it is probably useful to read through the whole document. This will perhaps help you plan your design, so that you can separate code into separate files and make packages and separate structs and so forth that will help you design a good piece of software.

Tasks and Questions:

(a) (5 points) Build a UDP server that listens to the IP multicast address and port number specified above. Your server *must not* echo anything back (respond) to the traffic generator. Your server should only receive zap events in a loop. In this task, the server only needs to print to the console whatever it receives from the traffic generator. Hint: `net.ListenMulticastUDP`

(b) (5 points) Develop a data structure for storing individual zap events. The struct must contain all the necessary fields to store channel changes (ignore storing status changes for now). Provide a constructor for the struct which can be used by your server when it receives a zap event. In addition the struct should have the following methods.

Method	**Description**
`String() string`	Return a string representation of your struct.
`Duration(provided ChZap) time.Duration`	Return the duration between two zap events: the receiving *zap* event and the *provided* event.

Hints: `time.Time` package, Methods: `time.Parse()`, `strings.Split()`, `strings.TrimSpace()`, Layout: `const timeLayout = "2006/01/02, 15:04:05"`

(c) (10 points) The next task is to use the simple in-memory storage shown in Listing 1 to store the channel changes received on your server.

Listing 1: Simple slide-based ChZap storage.

```
package ztorage

import (
  "zaplab/chzap"
)

type Zaps []chzap.ChZap

func NewZapStore() *Zaps {
  zs := make(Zaps, 0)
  return &zs
}

func (zs *Zaps) StoreZap(z chzap.ChZap) {
  *zs = append(*zs, z)
}

func (zs *Zaps) ComputeViewers(chName string) int {
  viewers := 0
  for _, v := range *zs {
    if v.ToChan == chName {
      viewers++
    }
    if v.FromChan == chName {
      viewers--
    }
  }
  return viewers
}
```

1. Use the API of the simple storage to compute the number of viewers on `NRK1` periodically, once every second. Explain what you observe from the output.

2. Implement the same for `TV2 Norge`. Is there any correlation between the data for NRK1 and TV2?
3. Implement another function that prints the number of entries in the storage. This function should be run every five seconds.
4. Use the memory profiler and run your server for 10 minutes. Explain what you observe?
5. What conclusions can you draw from this analysis?

In your handin document, please include the relevant code snippets for your solution to 1-3 (do not include the full zapserver code in the document.)

(d) (10 points) *Read also the next question before starting this one.* Implement a function that can compute a *list of the top-10* channels. Call this function periodically, once every second.

(e) (20 points) Implement a data structure that avoids the problems that you should have identified with the simple slice-based storage solution. Implement the data structure so that it can support you with keeping track of the top-10 list of channels. *Note that if you have problems with this task, you can skip to the next task. Or ask for help!*

(f) (20 points) In your UDP zapserver, implement an RPC-based subscription interface that one or more external clients can query to subscribe to a stream of viewer statistics (the top-10 list).

- The `subscribe()` method should take an argument, *refresh rate* that a specific RPC client wishes to be notified.
- The RPC server is serving statistics based on zap events from the zap storage, while continuously updating the server's storage (the state of the server).
- The RPC server and the zapserver part receiving zap events should be implemented as separate goroutines.
- Assume that the refresh rate is one second or more.

1. How would you characterize the access pattern to the server's state?
2. With this access pattern (workload) in mind. How would you protect the server's state to avoid returning a statistics computation that is incorrect or otherwise malformed?
3. Implement the corresponding RPC client. The client should display the viewership updates as they are received from the RPC server.

(g) (20 points) Now we want to analyze the duration between channel change clicks. To do that, we need to store the previous zap event for each IP, so that you can use the `Duration()` method that you developed earlier. Make a method that periodically sends durations to the RPC client.

Further details will follow later in a revised version of the text.