Lab 2: Message Passing system and Leader election

September 27, 2022

1 Message passing system

In the first part of this lab, we will learn how to program, in Go, a Synchronous Message passing system and a random walk. First you can have a look at the code in Figure 1.

1/ Execute the code. What is happening if time.Sleep(100 * time.Millisecond) of the player 1 is now equal to time.Sleep(200 * time.Millisecond)? Is this code is event driven or not? Can you simplify this code?

2/ Modify the code above such that the exchange of a ball between the two nodes is becoming the exchange of a vector. Each time a node receives the vector, it is adding k to the k-th component of this vector, for every k.

In the next two questions, we will study how to randomly elect a leader, by simply using a random walk. A random walk is a standard scheme used to sample nodes and edges in a network. A simple random walk in a distributed system is defined as follows: In each step, the token, with the extra data, goes from the current node to a random neighbor, uniformly chosen.

3/ What is the probability that node v receives the token at step n, knowing that that the node u had the token at step $n-1$ and node u has N_u neighbors.

4/ Implement the random walk for a general topology (for this question, you can assume that a node can have three neighbors at maximum).

- Hint 1: The topology is bi-directional;
- Hint 2: You define three types of nodes. The nodes with one neighbor, the nodes with two neighbors, and the nodes with three neighbors;
- Hint 3: You will have to use the select and switch statement and the math/rand package (see https://zetcode.com/golang/random/).

2 Leader election on a ring

The goal of the first part of the lab is to program the leader election algorithm introduced in the class. Please go to through the slides if you don't remember the exact algorithm. You need to complete the code in Figure 2. To help you, please answer the following questions.

1/ What should contain the structure Message? Complete the structure message in Figure 2.

2/ How many channels should you create in a ring composed of 3 nodes? (Remember that each node should be associated with a channel which displays its final decision). Complete the main() in Figure 2..

3/ To complete the function node in in Figure 2, using condition statements, switch and select. This function captures the behavior of a node during the election process. Look at the slides of the course to understand the different cases.

4/ Run the entire code (call the prof). (Bonus: Can you suggest an improvement of the code in case one node is failing?)

3 Bonus: Leader election in General Synchronous Graphs

We consider an arbitrary connected undirected graph $G = (V, E)$ having I nodes. We assume that the diameter of the network is known (you can imagine that an upper bound on the diameter is known). We also assume that each process is associated with a unique identifier. The algorithm works as follows:

- Every process maintains a record of the maximum id it has seen so far (initially its own).
- At each round, each process propagates this maximum on all of its outgoing edges.
- After diam rounds, if the maximum value seen is the process's own id, the process elects itself the leader. Otherwise, it is a non-leader.

1/ Implement the algorithm and try on a general topology with 5 nodes. Remember that the communication between the nodes, in this case, is bidirectional. To test whether or not your algorithm is working you should observe that the process with the highest index will output leader and the other processes will output non-leader.

4 Leader election

We define the k-neighborhood of a process i in the ring to be the set of processes that are at distance at most k from i in the ring (either to the left or to the right). The algorithm works in phases. In phase 0, all processors are winners. In phase k, a process i that is a phase $k-1$ winner sends ¡probe¿ messages with its identifier to the $f(k)$-neighborhood (one in each direction). We assume that $f(k)$ is strictly increasing in k.

- A processor does not transmit a $< probe >$ message to his neighbor if the message contains an identifier that is smaller than its own identifier.
- If the message arrives at the last process in the neighbourhood, then that last process sends back a $< reply >$ message to i.
- If i receives replies from both directions, processor i becomes a phase k winner, and it continues to phase $k + 1$.
- When a processor receives its own ¡probe¿ message, he becomes the leader of the ring. He will send a termination message around the ring.

1. For every k, what is the upper bound on the number of processes that are phase k winners?
2. How many phases are needed to have only one leader?
3. When $f(k) = 2^k$, what is that the total number of messages exchanged to elect a leader?

```
package main

import (
        "fmt"
        "time"
        )

type Ball struct{ hits int }

func main() {
    table := make(chan *Ball)
    go player1("Alice", table)
    go player2("Bob", table)

    table <- new(Ball) // game on; toss the ball
    time.Sleep(1 * time.Second)
    <-table // game over; grab the ball
}

func player1(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
func player2(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(200 * time.Millisecond)
        table <- ball
    }
}
```

Figure 1: Code Ping Pong

```
type Message struct {
        // to complete
}

func node(id int, channelinput chan *Message,
    channeloutput chan *Message, Decision chan *Message) {

        State := new(Message)
        State.terminate = "No"
        State.max = id
        Counter := 1
        for {
                select {

                }
        }
}

func main() {
  // Add all the channels

        go node(1,) //Add all the channels
        go node(2,) //Add all the channels
        go node(3,) //Add all the channels
        counter_main := 1
        Input := <- //to complete
        println(Input.max)
}
```

Figure 2: Code: Leader election