University *of* Dayton
Department of Computer Science

CPS 475/575
Secure Application Development

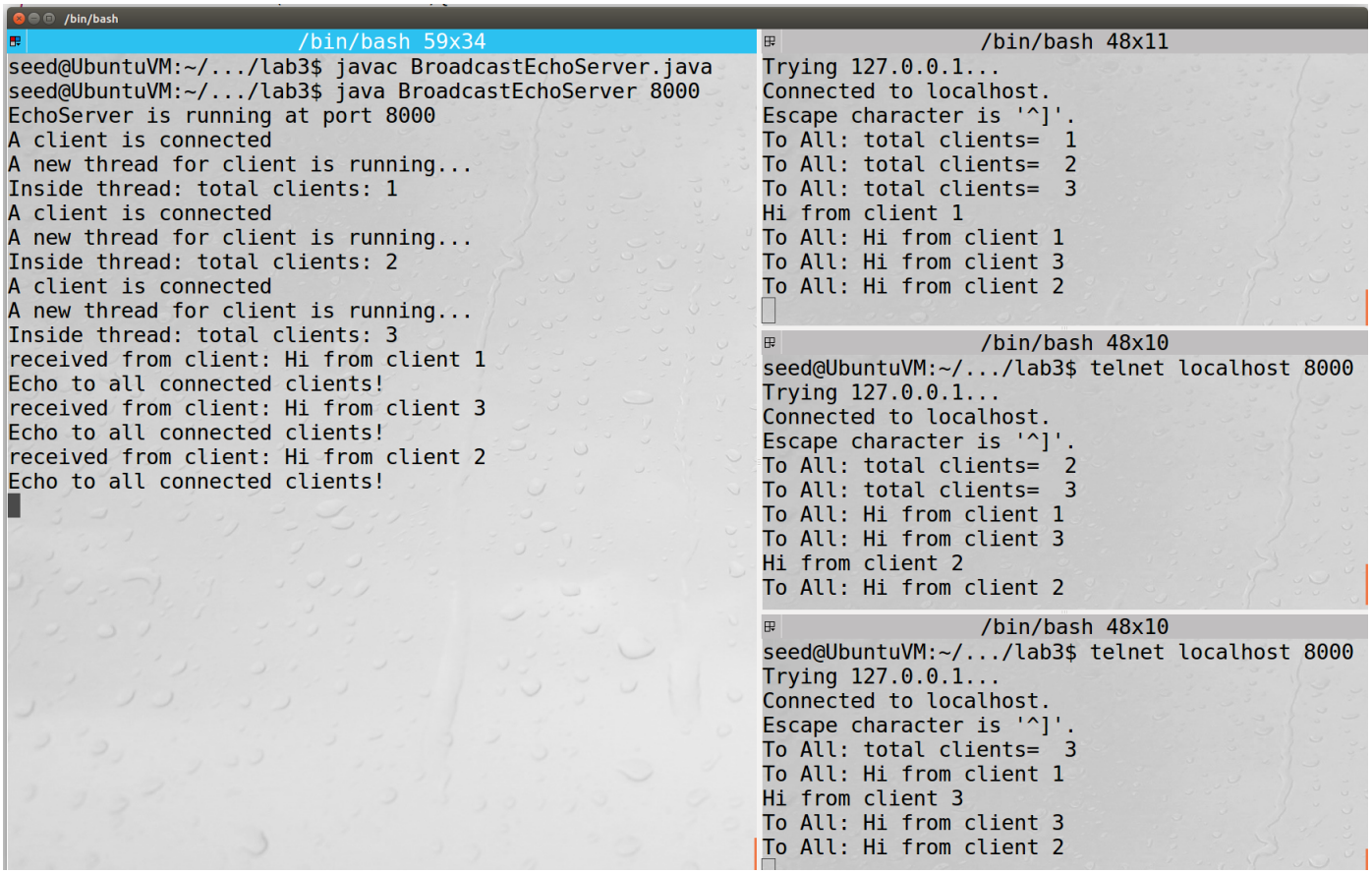
Lecture 8 – Preventing Data Races & Concurrent Programming in GoLang

Phu Phung
2/6/2020

Today's agenda

- Data Races and Preventing Data Races in Java
- Introduction to Go Programming Language
  - Hands-on: Data Races in Go
  - Hands-on: Simple EchoServer.go in Go
- Concurrent Programming in Go
  - Hands-on: Concurrent EchoServer.go
  - Hands-on: Concurrent BroadcastEchoServer.go

Review: Concurrent EchoServer.java Demo



Concurrent Programming Issues

```
class ThreadList{
   private ArrayList<EchoServerThread> threadlist = new
   ArrayList<EchoServerThread>();
   public ThreadList(){

   }
   public int getNumberofThreads(){
      return threadlist.size();
   }
   public void addThread(
                  EchoServerThread newthread){
      threadlist.add(newthread);
   }
   public void removeThread(
                     EchoServerThread thread){
      threadlist.remove(thread);
   }
}
```

Usage (create an instance and share among threads):

```
ThreadList threadlist = new ThreadList();
```

On-line Banking Example in Java

(Based on Raj Buyya's slides)

Several entities can access account potentially simultaneously (maybe a joint account, maybe automatic debits, …)

Suppose three entities each trying to perform an operation, either:

- deposit()
- withdraw()
- enquire()

Account methods

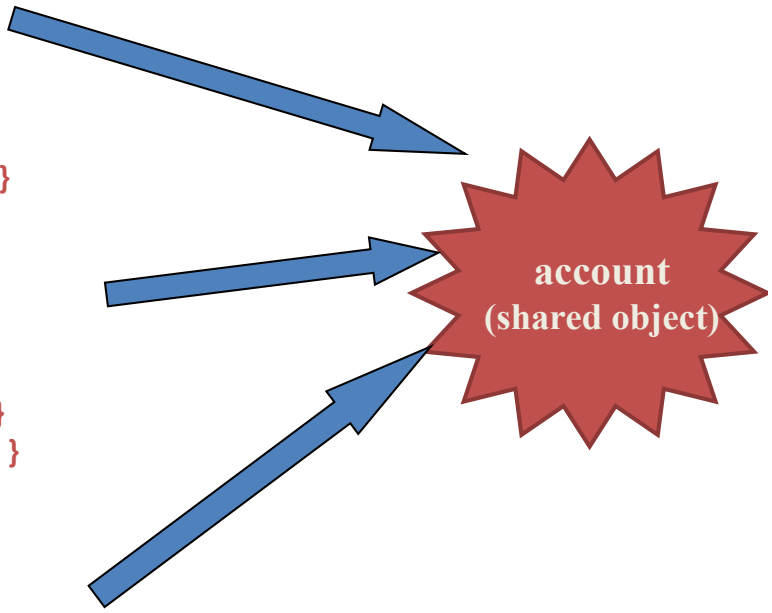
```
class Account {
    int balance;
    balance = //…
    public void deposit(int deposit_amount) {
            balance += deposit_amount;
    }

    public void withdraw(int withdrawal_amount) {
            balance -= withdrawal_amount;
    }
    public int enquire( ) {
            return balance.
    }
}
```

Three threads, one for each entities

```
class InternetBankingSystem {
      public static void main(String [] args  ) {
         Account accountObject = new Account ();
         Thread t1 = new Thread(new MyThread(accountObject));
         Thread t2 = new Thread(new YourThread(accountObject));
         Thread t3 = new Thread(new HerThread(accountObject));

         t1.start();
         t2.start();
         t3.start();
        // DO some other operation
     } // end main()
}
```

Shared account

```
class MyThread implements Runnable  {
        Account account;
       public MyThread (Account s) {  account = s;}
       public void run() { account.deposit(x); }
} // end class MyThread

class YourThread implements Runnable  {
        Account account;
       public YourThread (Account s) { account = s;}
       public void run() { account.withdraw(y); }
} // end class YourThread

class HerThread implements Runnable  {
        Account account;
       public HerThread (Account s) { account = s; }
       public void run() {display account.enquire(); }
} // end class HerThread
```



Online-banking example

- Account modeled as a global variable
- Withdrawal modeled as threads running

```
balance = balance – x
```

- In reality the machine code is more like

```
temp = balance – x
balance = temp
```

  - where temp is some thread-local variable

Data Races Example

Consider the following interleaving:

ATM1	ATM2
$temp_1$ = balance – x	
	$temp_2$ = balance – y
balance = $temp_1$	
	balance = $temp_2$

- Result: balance = balance – y
  - should be: balance = balance – (x + y)
- This is a race condition (data race) on global "balance"
- It only happens on rare interleavings

Data Races

- *Data races* are a multithreading bug
  - At least two threads access a shared variable
  - At least one of the threads writes the variable
  - The accesses are (potentially) simultaneous

- Races are usually undesirable
  - Source of nondeterminism
    - Program state depends on timing
  - Very hard to reproduce bugs

A data-races consequence: time-of-check-to-time-of-use (TOCTOU)

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

A Race-Avoidance Discipline

- Any access to a shared variable unprotected by a lock or similar is an error
- Use race-avoidance mechanisms/tools supported in programming languages to avoid and detect data-races, e.g.:
  - Lock, synchronization in Java
  - sync.WaitGroup, channel, mutex in GoLang
    - GoLang also has a built-in race detector (-race)

Preventing Data Races using Locks

- Locks
  - Special objects with two states: locked, unlocked
  - A locked lock is owned by the thread that locked it
  - At most one thread can own the lock at one time
    - The other threads are blocked and waiting

- Example:

```
lock(balanceLock)
balance = balance - 1
unlock(balanceLock)
```

Synchronization in Java

- Synchronization in Java can be used to prevent data races
  - Java provides two basic synchronization idioms
    - Synchronized Methods
    - Synchronized statements
  - Ensure that
    - At most one invocation of synchronized methods/statements on the same object at the same time
    - There is no the synchronized methods/statements must be blocked until the first thread is done
- More reference:
  https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html

Data Races-Free Example

```
class Account {
    int balance;
    balance = //…
     public synchronized void deposit(int deposit_amount) {
            balance += deposit_amount;
      }

     public synchronized  void withdraw(int
       withdrawal_amount) {
            balance -= withdrawal_amount;
     }
     public synchronized int enquire( ) {
            return balance.
     }
}
```

To Do as a Developer

- Be aware and take responsibility for potential bugs when developing multi-threaded/process resource sharing applications
- Keep yourself up-to-date with tools in the state-of-the-art

A. Is a multithreading bug happening when at least two threads read a shared variable simultaneous (concurrently)
B. Can be avoided using locks or synchronization during the development phase
C. Happen even within a sequential (single-threaded and synchronous) programs
D. A and B
E. A, B, and C.

Lab 3 – Detailed Outline

Lab's instructions: http://bit.ly/secad-s20-lab3 (Available on Isidore 11:55 PM 2/4/2020)

Task 1 (8p): Multi-Threaded EchoServer.java

a. Develop the program (Lecture 7)
b. Sharing among threads (Lecture 7) and avoiding Data races (Homework – just introduced)

Task 2 (12p): Concurrent EchoServer.go (This lecture - Lecture 8)

a. Develop the program
b. Channel for a new connection
c. Broadcast messages to all clients
d. Channel for a lost connection

Task 3 (10p): Asynchronous Programming in Node.js (Lecture 9)

a. (3p) Common Node.js programs
b. Simple telnet.js program
    i. (4p) Receive data asynchronously
    ii. (3 p) Get user inputs and send to the server

Agenda

- Data Races and Preventing Data Races in Java
- Introduction to Go Programming Language
  - Hands-on: Data Races in Go
  - Hands-on: Simple EchoServer.go in Go
- Concurrent Programming in Go
  - Hands-on: Concurrent EchoServer.go
  - Hands-on: Concurrent BroadcastEchoServer.go

Introduction to Go/GoLang

- Go is a compiled programming language designed at Google, announced in 2009, and the first version released in 2012
  - https://golang.org: *"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."*
- Go's motivations: "*address criticism of other languages in use at Google, while retaining their useful characteristics*" to use in production at Google
  - Syntax is similar to C, but flexibility like Python and JavaScript
    - More secure: memory safety, garbage collection, structural typing, and communicating sequential processes (CSP)-style concurrency.
  - Support high-performance networking and multiprocessing

Notable Users of Go (except Google)

- Docker, a set of tools for deploying Linux containers
- Couchbase, Query and Indexing services
- Dropbox, migrated critical components from Python to Go
- Ethereum, a cryptocurrency
- MongoDB
- Netflix
- Uber, geofence-based queries

. . .

- Comprehensive list: https://github.com/golang/go/wiki/GoUsers

Hands-on: “Hello World!” in Go

- https://gobyexample.com/hello-world

hello.go

```go
package main

import "fmt"

func main() {
    fmt.Println("hello from Phu Phung - SecAD")
}
```

- Install the Go compiler from SEED VM:
  $ `sudo apt install golang-go`
- Get the code from the course repo:
  $ `cd ~/secad && git pull`
  $ `cd lec<tab>/lecture8<tab> && ls`
- Copy all of the code to your private repo to edit the `hello.go` file to add your name and execute:
  $ `go run hello.go`

“Hello World!” in Go - DEMO

- go run
  - Will compile the source to a binary program in a temporary folder and execute it

```
seed@UbuntuVM:~/.../lecture8-golang$ ls
hello.go  norace-example.go
seed@UbuntuVM:~/.../lecture8-golang$ go run hello.go
hello from Phu Phung - SecAD
seed@UbuntuVM:~/.../lecture8-golang$ cat hello.go
package main

import "fmt"

func main() {
    fmt.Println("hello from Phu Phung - SecAD")
}seed@UbuntuVM:~/.../lecture8-golang$
```

Single-thread vs multi-threaded program examples in Go

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main(){
 6     var i int
 7     func(){
 8         i = 1
 9     }()
10     fmt.Println(i)
11 }
```

```
 1 package main
 2
 3 import "fmt"
 4
 5 func main(){
 6     var i int
 7     go func(){
 8         i = 1
 9     }()
10     fmt.Println(i)
11 }
```

- Hands-on: run the first program (provided in norace-example.go):
  `$ go run norace-example.go`
- Copy the norace-example.go file and rename it to race-example.go, modify the code as in the right figure. Run it to observe the results
  `$ go run race-example.go`

Demo: Data Races in Go

```
}seed@UbuntuVM:~/.../lecture8-golang$ cat norace-example.go
package main

import "fmt"

func main(){
        var i int
        func(){
                i = 1
        }()
        fmt.Println(i)
}
seed@UbuntuVM:~/.../lecture8-golang$ go run norace-example.go
1
```

```
seed@UbuntuVM:~/.../lecture8-golang$ cp norace-example.go race-example.go
seed@UbuntuVM:~/.../lecture8-golang$ subl race-example.go
seed@UbuntuVM:~/.../lecture8-golang$ go run race-example.go
0
seed@UbuntuVM:~/.../lecture8-golang$ cat race-example.go
package main

import "fmt"

func main(){
        var i int
        go func(){
                i = 1
        }()
        fmt.Println(i)
}
seed@UbuntuVM:~/.../lecture8-golang$
```

`goroutine` and concurrent programming in Go

- `goroutine` is a primary concurrency construct in Go as a lightweight thread
  - Java can run tens of 1000's threads while Go can run tens of millions of goroutines concurrently*
- To start a new goroutine, invoke a function with the go keyword in prefix, e.g.:

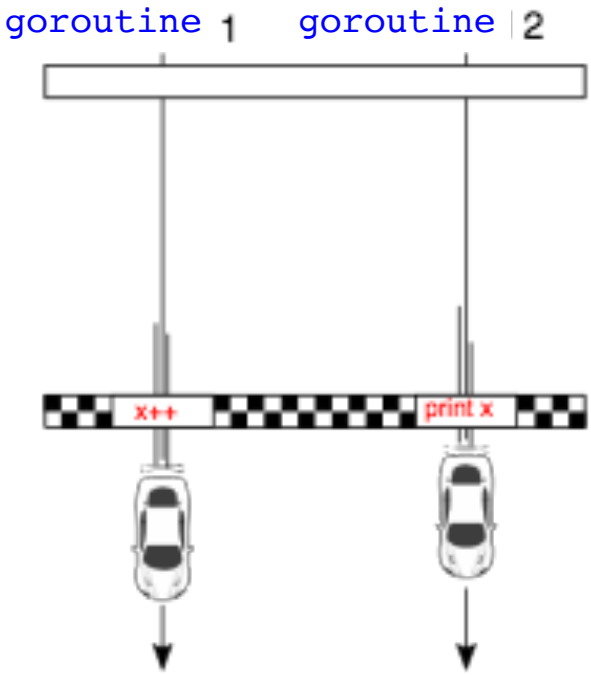
```
go func
```

```
func

go
```

*https://rcoh.me/posts/why-you-can-have-a-million-go-routines-but-only-1000-java-threads

Data Races Example in Go Explained

- Similar to Java, Data Races happen in Go programs when
  - Two or more goroutines access the shared resource, and
  - At least one goroutine modify/write to the resource



Credit: https://programming.guide/go/data-races-explained.html

Detecting Data Races in Go

https://golang.org/doc/articles/race_detector.html

- Go provides a built-in data race detector: use –race in go command, i.e.,:
  $ `go run –race race-example.go`
  - However, it only works on several physical architecture, VM is not supported

```
aswx041171:~ pphung1$ cat race-example.go
package main

import "fmt"

func main(){
        var i int
        go func(){
                i = 1
        }()
        fmt.Println(i)
}
aswx041171:~ pphung1$ go run race-example.go
0
aswx041171:~ pphung1$ go run -race race-example.go
0
==================
WARNING: DATA RACE
Write at 0x00c000018060 by goroutine 6:
  main.main.func1()
      /Users/pphung1/race-example.go:8 +0x38

Previous read at 0x00c000018060 by main goroutine:
  main.main()
      /Users/pphung1/race-example.go:10 +0x88

Goroutine 6 (running) created at:
  main.main()
      /Users/pphung1/race-example.go:7 +0x7a
==================
Found 1 data race(s)
exit status 66
aswx041171:~ pphung1$
```

Methods to avoid data races in Go

- The general discipline to avoid data races is to synchronize access to all mutable data shared between threads, i.e., goroutine
- Go provides 3 methods
  - channel
  - lock
  - synchronization (in `"sync"` and `"sync/atomic"` packages)

https://yourbasic.org/golang/data-races-explained/

`channel` in Go

- A channel is a mechanism for goroutines to **synchronize execution** and **communicate** by passing values [yourbasic.org/golang]
- To create a channel, use `make` function, e.g.,:
  ```
  i := make(chan int)
  ```
- To send/write a value to a channel, use `chan <- value`, e.g.,:
  ```
  i <- 1
  ```
- To read/receive the value from a channel, use `<-chan`, e.g.::
  ```
  value := <-i
  ```

Hands-on: Create a channel for `race-example.go`

- Copy the race-example.go and rename it to chan-example.go, and revise this new file

```
$ cp race-example.go chan-example.go
$ subl chan-example.go &
```

```go
package main

import "fmt"

func main(){
    var i int
    go func(){
        i = 1
    }()
    fmt.Println(i)
}
```

race-example.go

```go
package main

import "fmt"

func main(){
    i := make(chan int)
    go func(){
        i <- 1 //write to the channel
    }()
    //value_i := <-i //read from the channel
    fmt.Println(<-i)
}
```

chan-example.go

Hands-on: chan-example.go

- Run the new program and compare the result

```
seed@UbuntuVM:~/.../lecture8-golang$ go run chan-example.go
1
seed@UbuntuVM:~/.../lecture8-golang$ cat chan-example.go
package main

import "fmt"

func main(){
        i := make (chan int)
        go func(){
                i <-1
        }()
        fmt.Println(<-i)
}
seed@UbuntuVM:~/.../lecture8-golang$ go run chan-example.go
1
seed@UbuntuVM:~/.../lecture8-golang$
```

chan-example.go with Go race detector

- No data race is detected (compared with slide 28)

```
aswx041171:~ pphung1$ go run -race chan-example.go
1
aswx041171:~ pphung1$ cat chan-example.go
package main

import "fmt"

func main(){
        i := make(chan int)
        go func(){
                i <- 1
        }()
        fmt.Println(<-i)
}
aswx041171:~ pphung1$ go run chan-example.go
1
aswx041171:~ pphung1$
```

Overview of basic Go packages for Internet Applications

- `"fmt"`
  - formatted I/O with functions analogous to C's printf and scanf
- `"net"`
  - a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets
- `"os"`
  - a platform-independent interface to operating system functionality.
- All available packages: https://golang.org/pkg/

Client-side programming in Go

- https://golang.org/pkg/net/

The Dial function connects to a server:

```
conn, err := net.Dial("tcp", "golang.org:80")
if err != nil {
        // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

Server-side programming in Go

- https://golang.org/pkg/net/

The Listen function creates servers:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
        // handle error
}
for {
        conn, err := ln.Accept()
        if err != nil {
                // handle error
        }
        go handleConnection(conn)
}
```

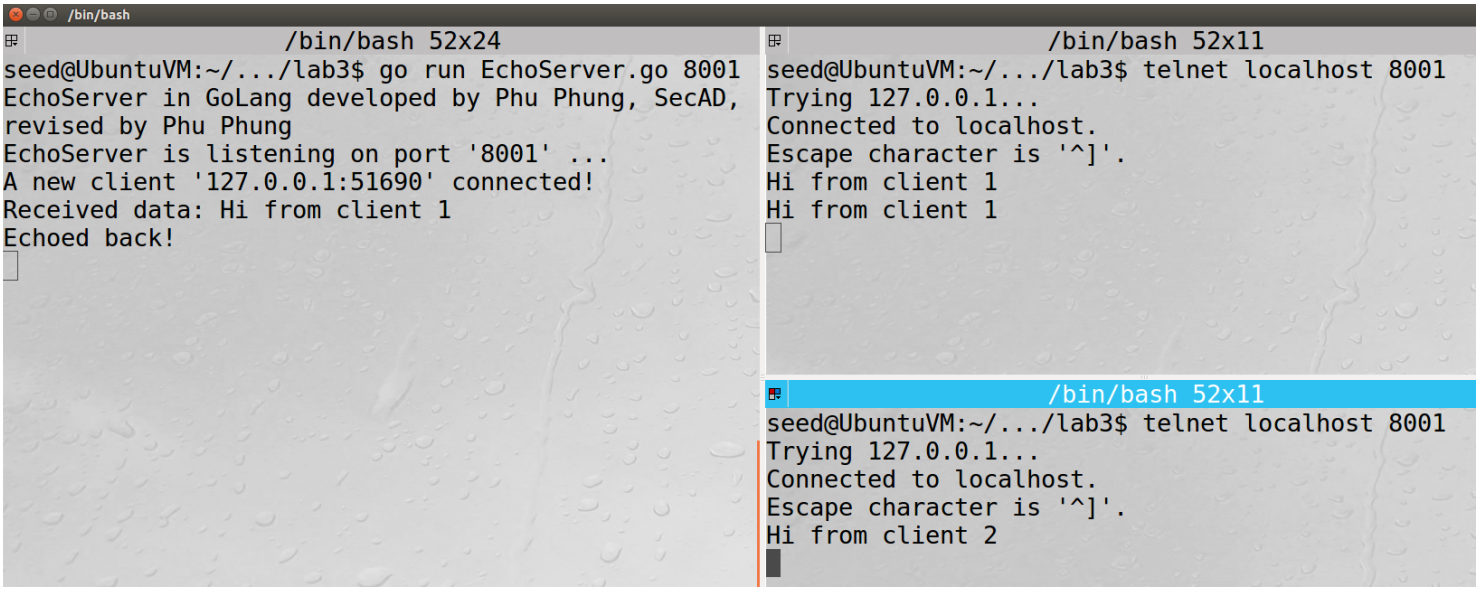
Hands-on: EchoServer.go

- Provided in the course repository at labs/lab3
  - Copy the file to your private repo and run (it is similar to the original EchoServer.java)

```
22      server, err := net.Listen("tcp", ":"+port)
23      if err != nil {
24          fmt.Printf("Cannot listen on port '" + port + "'!\n")
25          os.Exit(2)
26      }
27      fmt.Println("EchoServer in GoLang developed by Phu Phung, SecAD, revised by Your Name")
28      fmt.Printf("EchoServer is listening on port '%s' ...\n", port)
29      client_conn, _ := server.Accept()
30      fmt.Printf("A new client '%s' connected!\n", client_conn.RemoteAddr().String())
31      var buffer [BUFFERSIZE]byte
32      for {
33          byte_received, read_err := client_conn.Read(buffer[0:])
34          if read_err != nil {
35              fmt.Println("Error in receiving...")
36              return
37          }
38          _, write_err := client_conn.Write(buffer[0:byte_received])
39          if write_err != nil {
40              fmt.Println("Error in sending...")
41              return
42          }
43          fmt.Printf("Received data: %sEchoed back!\n", buffer)
44      }
45  }
```

Hands-on: Execution of EchoServer.go

- It can talk with only one client



Exercise 1:
Making EchoServer.go Concurrently

- Move the key statements for a client to the body of a new function, so that we can create a new goroutine with "go", e.g.:

```
func client_goroutine(client_conn net.Conn){
    /*key statements for a client will be
      pasted here*/
}
```

/*key statements for a client*/ in the main func will be replaced with:

```
go client_goroutine(client_conn)
```

Hands-on: Concurrent EchoServer.go

- Step 1:

```
22      server, err := net.Listen("tcp", ":"+port)
23      if err != nil {
24          fmt.Printf("Cannot listen on port '" + port + "'!\n")
25          os.Exit(2)
26      }
27      fmt.Println("EchoServer in GoLang developed by Phu Phung, SecAD, revised by Your Name")
28      fmt.Printf("EchoServer is listening on port '%s' ...\n", port)
29      client_conn, _ := server.Accept()
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45  }
```

Move this piece of code (from line 30 to line 44) to the inside of a new function:

```
func client_goroutine(client_conn net.Conn){
        /*code here*/
}
```

Then, replace with:

```
go client_goroutine(client_conn)
```

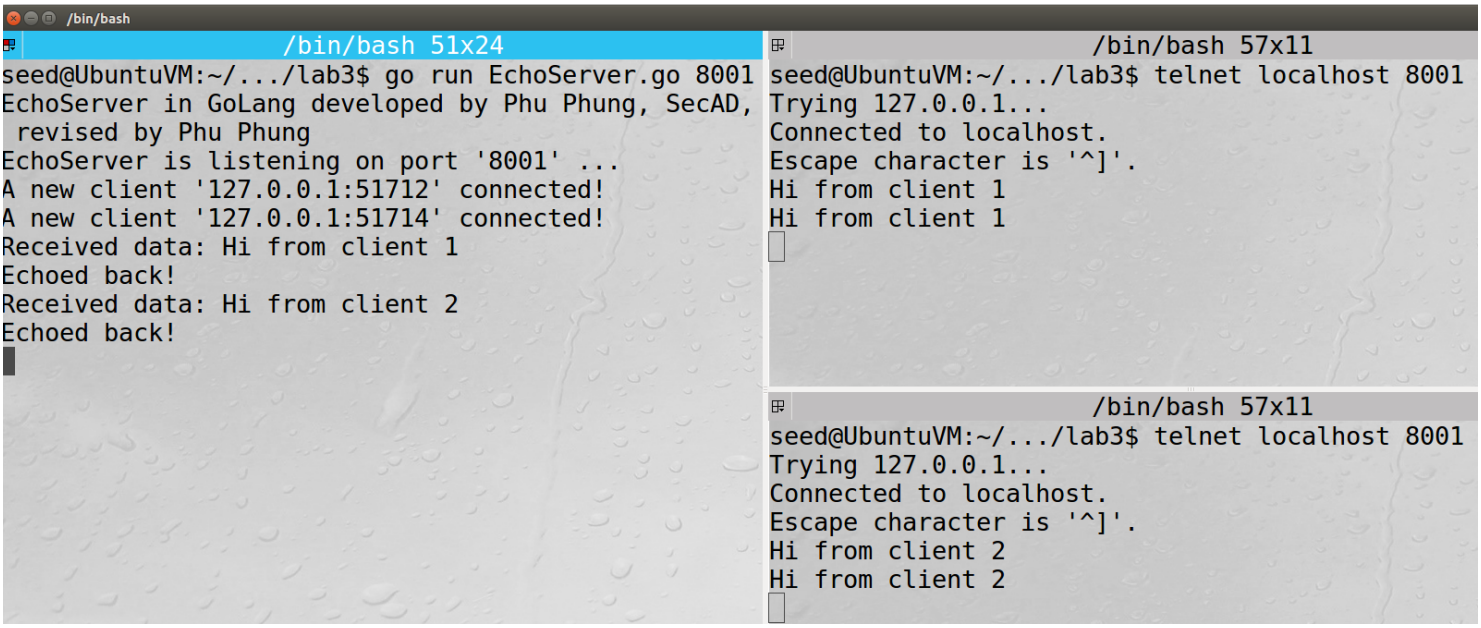
Hands-on: Concurrent EchoServer.go

- Step 2:
  - Put the accept and goroutine within a forever loop

```
for {
    client_conn, _ := server.Accept()
    go client_goroutine(client_conn)
}
```

Testing: Concurrent EchoServer.go

- It should accept multiple clients at the same time



Exercise 2: Printout the number of connected clients

- A possible solution:
  - Store all client connections in a map (outside and before `func main`):

    ```
    var allClient_conns = make(map[net.Conn]string)
    ```

  - Inside the for loop of `func main`,
    before `go client_goroutine` code,
    update/access the map in each go routine:

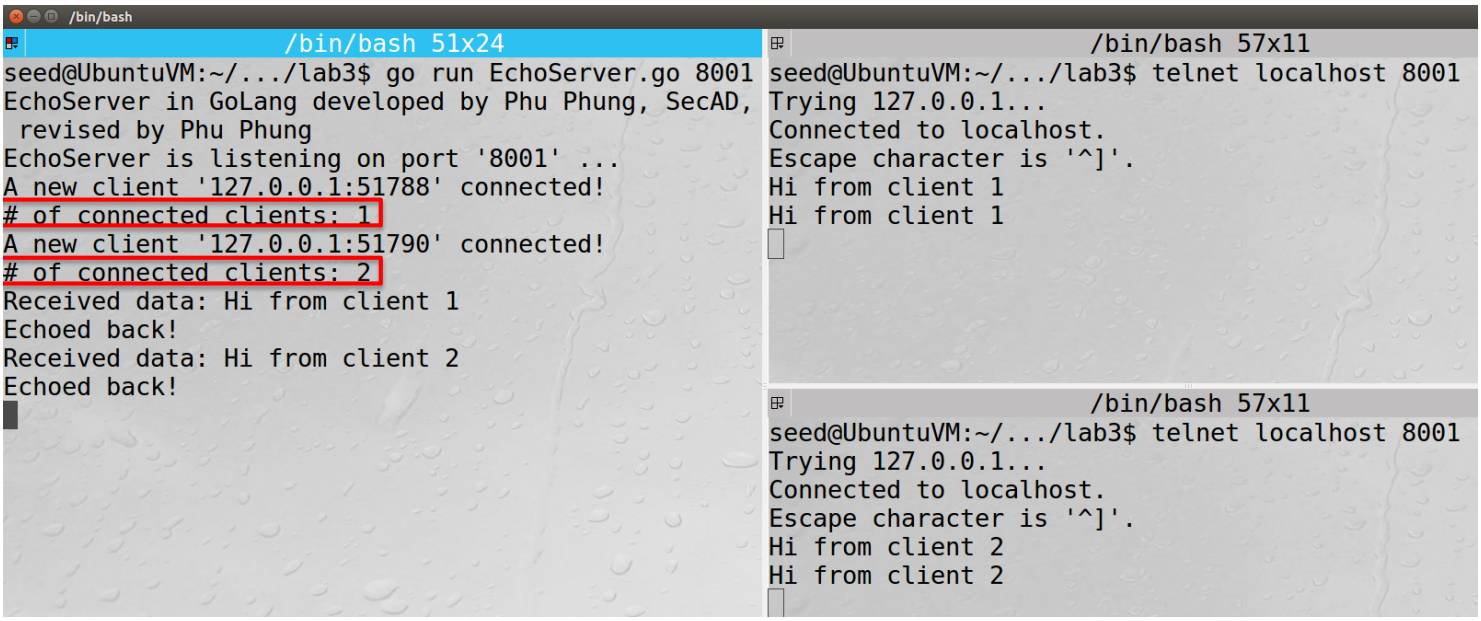
    ```
    allClient_conns[client_conn] =
                    client_conn.RemoteAddr().String()
    ```

  - Inside the `fmt.Printf` in the func `client_goroutine` (new code in red):

    ```
    fmt.Printf("A new client '%s' connected!\n# of connected clients: %d\n",
                client_conn.RemoteAddr().String(),
                len(allClient_conns))
    ```

Demo: # of connected clients

- Each goroutine can access the shared map to update and print:



- Discussion: is there any issue with the # of connected clients?

Next classes

- Tuesday, February 11
  - Lecture 9:
    - Concurrent Programming in Go
      - Hands-on: Concurrent EchoServer.go with channel
      - Hands-on: Concurrent BroadcastEchoServer.go
    - Asynchronous Programming in Node.js