# **CPS 475/575 - Secure Application Development, Spring 2020**

Dr. Phu Phung

Department of Computer Science,  
University of Dayton

Lab 3 - Secure Concurrent Programming in Java, GoLang, and Node.js

Instruction released: 2/4/2020

Deadline: 11:55 PM 2/18/2020.

Introduction

In this lab, you will gain hands-on experience in concurrent programming in Java, GoLang, and Node.js. You will also learn the data race security issues in concurrent programs and how to address these issues. This lab will be used as the foundation for Programming Assignment 1.

**Source code organization:** You need to push all your code for this lab under a folder labs/lab3 within your private repository. You need to include the URL of this folder at the beginning of your report. For example, in my case, the URL is:  
<https://bitbucket.org/phu-udayton/secad-pphung1/src/master/labs/lab3/>

**You will lose 5% of this lab grade if this URL is missing in your report and 5% if the code is not in your repository.**

This lab has three tasks of 30 points:  
Task 1 (8 points). Simple Multi-Threaded EchoServer.java

Task 2 (12 points). Concurrent EchoServer.go

Task 3 (10 points). Asynchronous Programming in Node.js

Report and submission

You need to write your report to submit via Isidore. Your report must be submitted in PDF file (reports not in PDF format will be returned without grading for this lab). Your report MUST have the course number, course name, the instructor, your name and UD email. You MUST organize your report as in this instruction, for example:

Task 1. xxx

1. Yyy

You need to demonstrate your experiments by screenshots, therefore, **it is important to capture screenshots when you perform the lab**. **Your screenshots must have a short description or a caption to explain the task.**

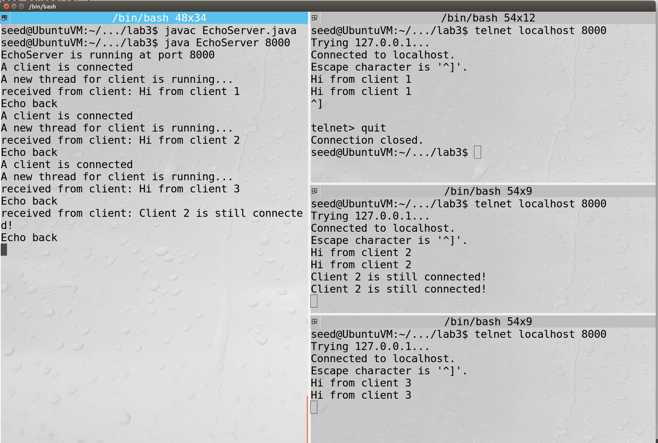
I recommend you to use Google Docs to write your report as it is convenient to include the screenshots for your report. **Do NOT copy the lab’s instructions in your report.** If code is required, **you need to provide your code as plaintext** (not screenshot) in your report. *If you use Google Docs to write the report, you can use an add-on such as “Code Blocks” (*[*https://gsuite.google.com/marketplace/app/code\_blocks/100740430168*](https://gsuite.google.com/marketplace/app/code_blocks/100740430168)*) to format and indent your code.*

## Task 1 (8 points). Simple Multi-Threaded EchoServer.java.

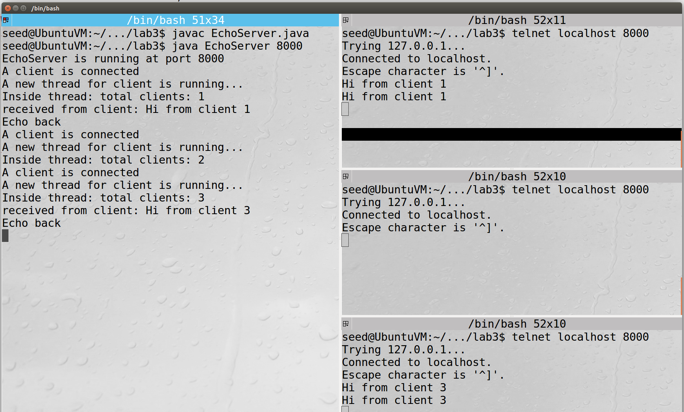
In this task, you need to implement the multi-threaded version of the EchoServer.java. Steps and code for this program is introduced in Lecture 7 (2/4/2020).

1. (4 points). Multi-threaded EchoServer.java

In this subtask, you need to revise the EchoServer.java provided in Lecture 3 to accept and communicate with multiple clients at the same time. The steps are provided in Lecture 9. Include your source code as text in your report, and capture a screenshot demonstrating the multiple connections similar as below. You need to print your name inside a System.out.println(..) in the application when it runs.



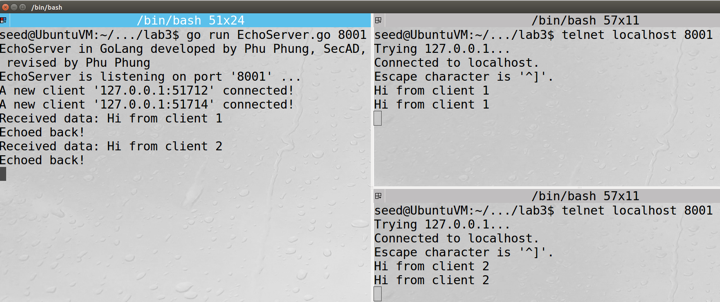
1. (4 points) Sharing among threads  
   In Java, a class instance must be created to share among the threads so that each thread and get an update the shared information. In this subtask, you will develop such a class to keep the list of active threads. The skeleton of this class is available in the course repo at labs/lab3/ThreadList.java

Follow the steps from slide 23-37 in Lecture 9 to implement the methods in this class and share the instance of the class among threads so that you can print out the number of connected clients inside a thread. In addition, analyze the issues of the data races and make the methods synchronized in ThreadList class if necessary so that it can prevent potential data races. Include your new code as text in your report and capture a screenshot demonstrating the test case:  
  


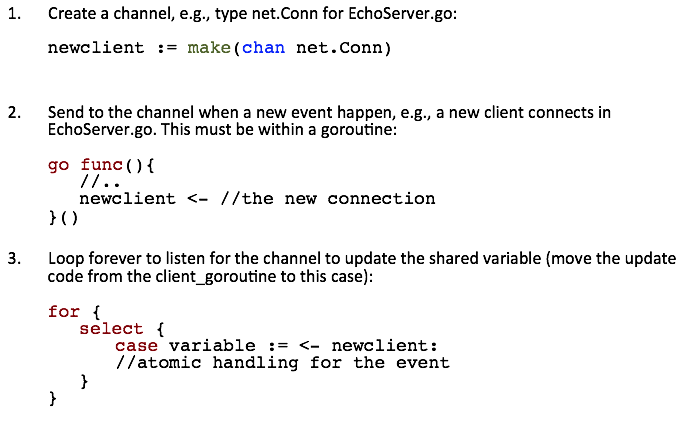
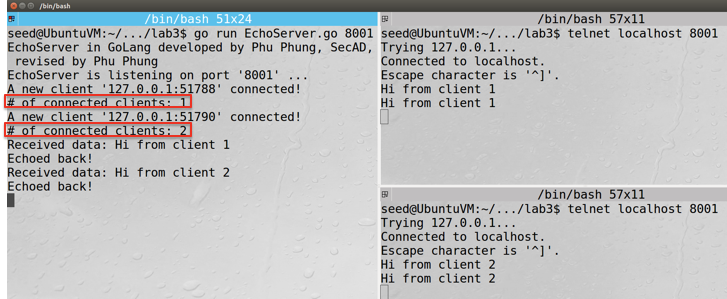
## Task 2 (12 points). Concurrent EchoServer.go

1. (3 points) Concurrent EchoServer.go

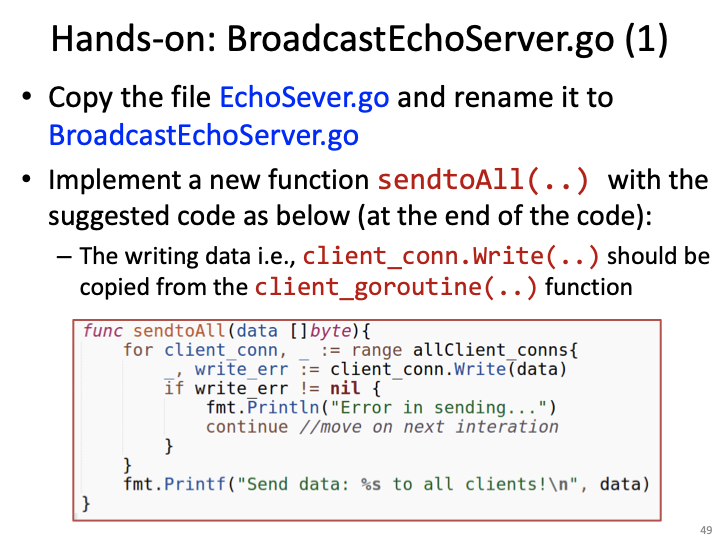
Follow the steps provided in Lecture 8 to implement the concurrent EchoServer.go program that can print out the number of connected clients within a goroutine. Include your source code as text **with a brief description** in your report, and capture a screenshot demonstrating the multiple connections and the number of connected clients similar to the below screenshot. You need to print your name in the application when it runs.

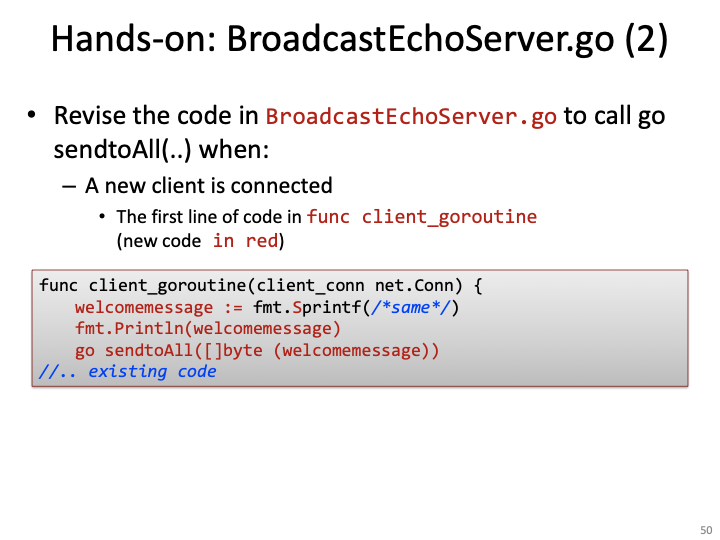


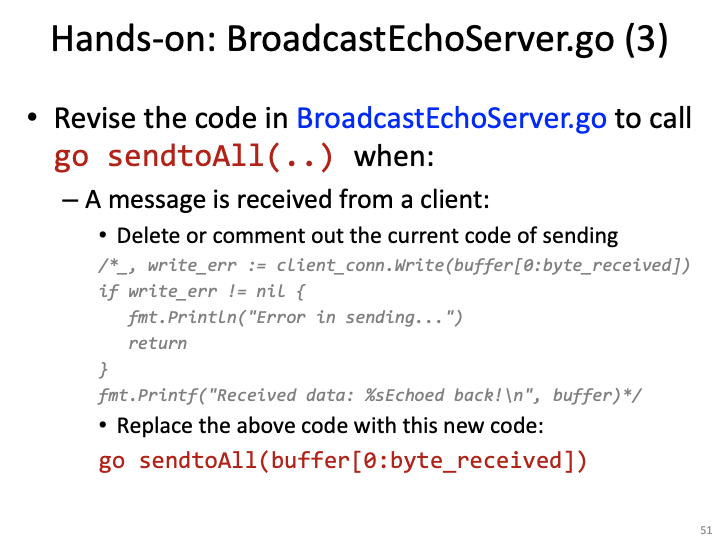
1. (3 points) channel in Go  
   In this subtask, you need to define a channel to synchronize among goroutines whenever a new client is connected. The steps are as follows (illustrated and demonstrated in Lecture 8).

  
Implement these mechanisms and include the code snippet **with a brief description** in your report. Capture a screenshot to demonstrate the cases when a new client is connected with this new channel implementation. Your screenshot should be similar below:  


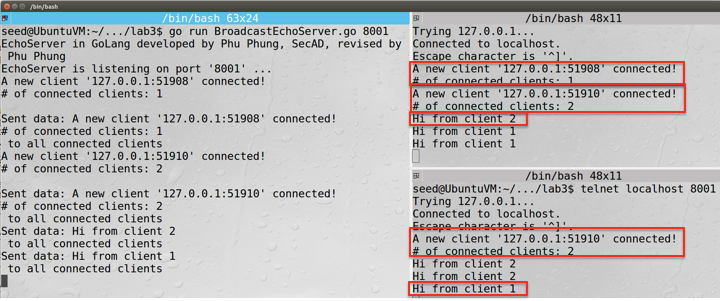
1. (3 points) Concurrent BroadcastEchoServer.go  
   In this subtask, you will revise the EchoServer.go program to broadcast (send to all) corresponding messages to the connected clients when a new client is connected **and** when a message from any client has arrived. Follow steps below to implement this new program (introduced in Lecture 8).







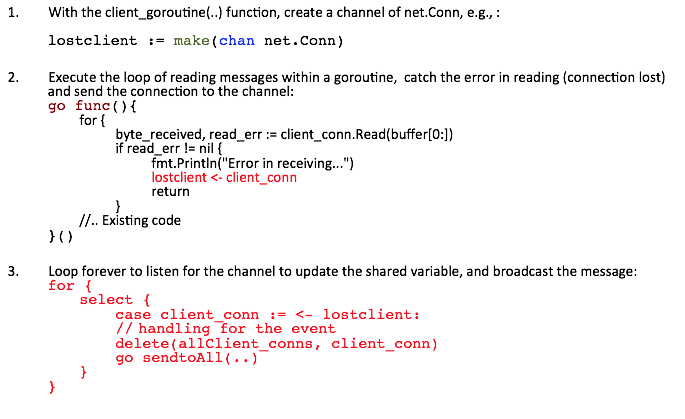
Include the code snippets of this program **with a brief description** in your report. And capture a screenshot to demonstrate the test cases are working as below.



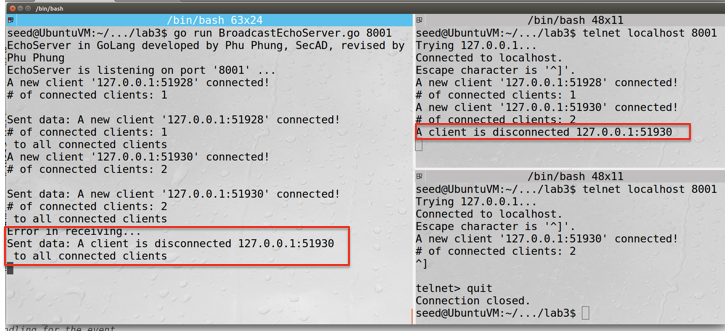
Screenshot - *Broadcasting when a new client is connected and a new message is received*

1. (3 points) Using channel to handle lost connections

In this subtask, you need to handle the event when a connection is lost in the BroadcastEchoServer.go program. Follow the steps below to implement the revision (introduced in Lecture 8).



You need to include the new code snippet **with a brief description** in your report and capture a screenshot demonstrating the case as below.

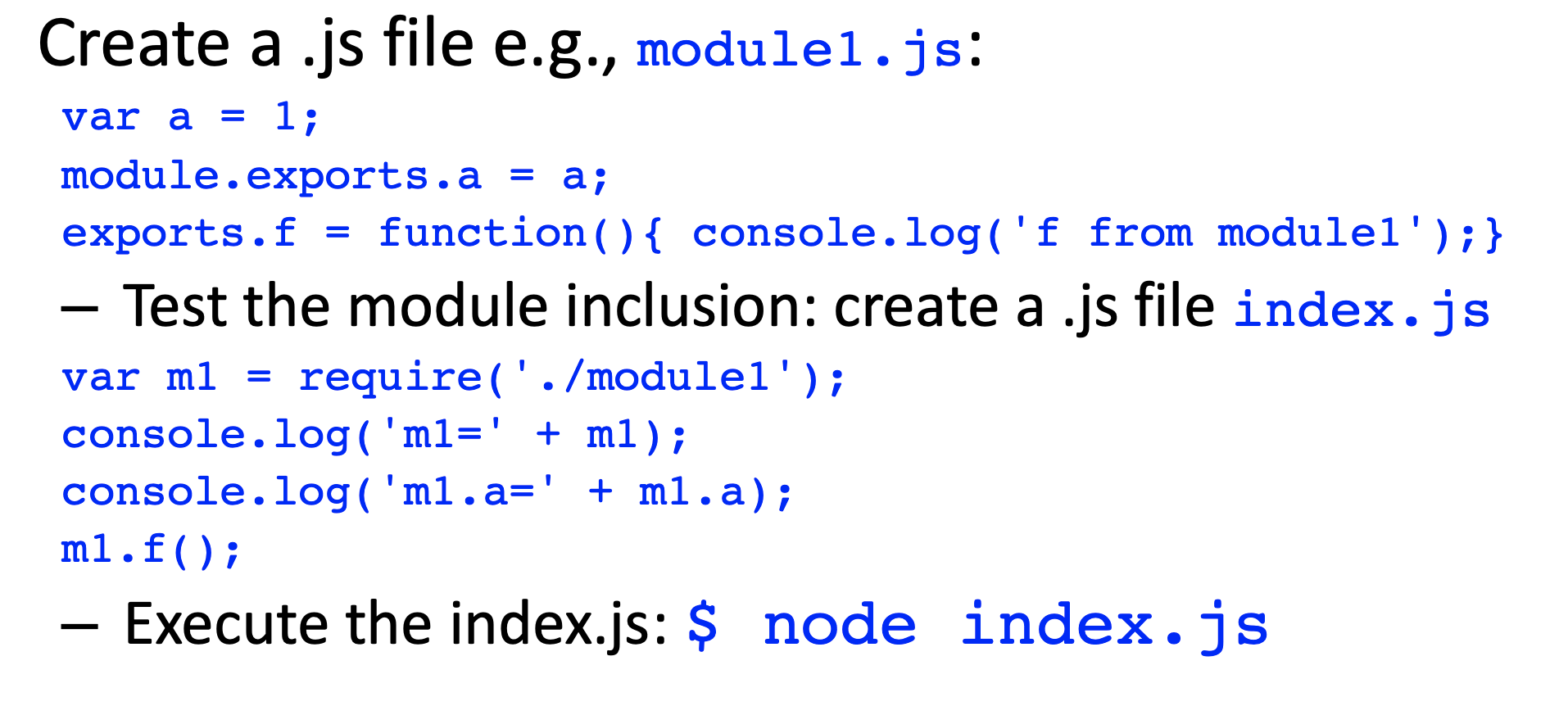


Screenshot - *A lost connection is captured, the new number of clients is updated and broadcast to all connected clients*

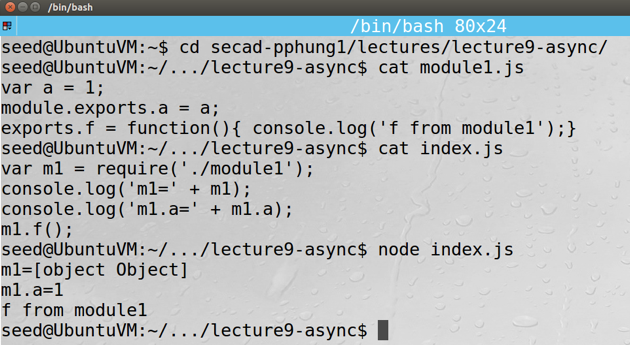
## Task 3 (10 points). Asynchronous Programming in Node.js

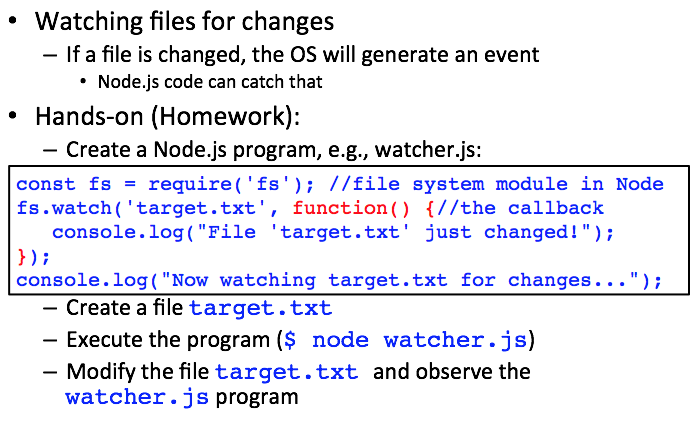
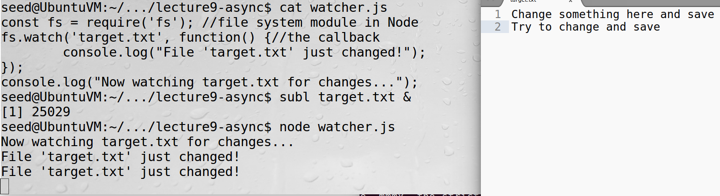
In this task, you will do hands-on experience in developing a non-blocking, asynchronous client (similar to simple telnet) in Node.js programming language.

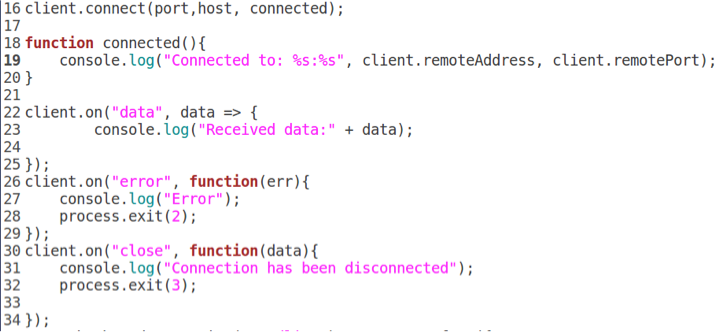
1. Common Node.js programs
   1. (1.5 points) Modules in Node.js

Code the two files module1.js and index.js as below   
(as introduced in Lecture 10).  


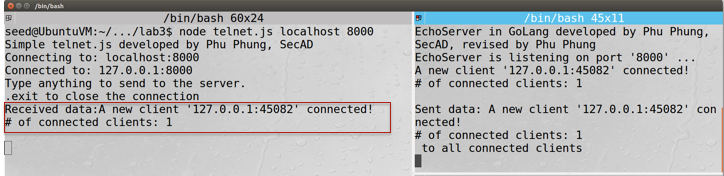
Execute the code to capture a screenshot and explain the result in your report. Your screenshot should be similar as below.



* 1. (1.5 points) Node.js Event Loop  
     Develop a Node.js program that watches the modification of the file system as below (Introduced in Lecture 10):   
       
     Create the file target.txt, execute the Node.js program, then open the file in gedit to modify it and observe the Node.js program result. Capture a screenshot and explain the results in your report. Your screenshot should be similar as below.  
     

1. A simple telnet.js client  
   The code skeleton telnet.js is provided in the course repo at secad/labs/lab3 and can connect with any TCP server. Your task is to develop the program further to fully function: can receive data from the server, and can get data from the keyboard and send it to the server.
   1. (4 points) Receiving data and handle other events  
      Follow steps in Lecture 10 to implement these handlers in callback functions. The code is given again here:  
      

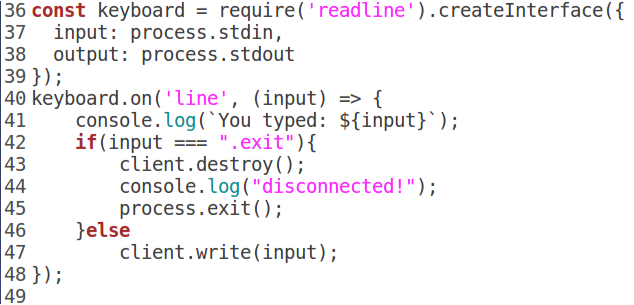
Test the program with the BroadcastEchoServer.go and another telnet client to capture a screenshot to demonstrate the execution. The screenshot should be similar to the below.

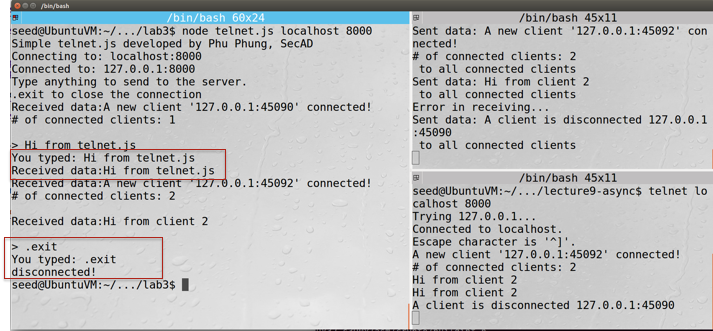


* 1. (3 points) Get user input and send to the server

In this subtask, you need to implement a keyboard interface to read the user input in order to send the data to the server. You also need to check if the input is equal to some predefined keyword e.g., .exit so that the client will disconnect the connection.

The code is introduced in Lecture 9 and provided again below.



Implement the code and test with the BroadcastEchoServer.go with several inputs from the keyboard. You need to have another client to test when the user typed the keyword for disconnecting. Capture and include a screenshot to demonstrate the programs with **a brief description** in your report as below.  


Screenshot - *a telnet.js client program can communicate asynchronously with a server and close connection upon user command*