

Assign-04 : The “Can We Talk” System

Who

This assignment is best completed with a partner (but can be completed individually if you like). Before you begin on the assignment, please use the **A-04 Group** sign-up (under the *Groups* option in *Course Tools*) to organize yourselves into partnerships. Remember that **both members** of the partnership **need to sign-up** and enroll in the **same group**.

Description

TCP/IP is **the most heavily used communications protocol** for inter-process communication. It is broadly supported by the socket programming *paradigm* across multiple platforms. In this assignment, you will write a **chat program** to demonstrate the basics of socket-based TCP/IP communications.

Objectives

- Practice C Programming techniques for socket-level programming as well as multi-threaded solutions
- Practice the integration and use of a 3rd party library (NCURSES)

Requirements

1. In this assignment – you are creating a system called CHAT-SYSTEM. This system is comprised of 2 applications – the server (called chat-server) and the client (called chat-client). These names must be used and reflected in your system development structure. (Please refer to the “Linux Development Project Code Structure” document in the course content)
2. Your solution architecture must be a central server model written in ANSI C.
 - The server must be multi-threaded. A new thread will be created each time a new user joins the conversation. This thread will be responsible for accepting incoming messages from that user and broadcasting them to the other chat users. Call the server “chat-server”.
 - **QUESTION:** How will each of the threads learn about all of the other threads and their IP addresses in order to send the message? Think of a data structure that might be used to hold all client information (IP Address, the user name, etc.) within the server and will be visible and shared among all the communication threads.
 - Your client application will be written to make use of the ncurses library in order to facilitate the multiple windows. Call the client program “chat-client”.
 - I have provided some sample ncurses programs in the ncurses-samples.tar archive
 - You may also want to experiment with threading the client program – one thread to handle the outgoing message window and one thread to handle the incoming messages window.
 - The chat functionality must operate across computers that are in the same subnet
3. Your chat solution must be able to **support at least 2 users being able to chat** with each other.
 - This means that each person can see the messages in the conversation as it goes back and forth – so each user’s message must be *tagged* with their name (or userID)
 - Your server design must be able to support a maximum of 10 clients.
4. While running the CHAT-SYSTEM the minimum configuration must be:
 - The chat-server application must run on a Linux VM (MACHINE-A)
 - One of the chat-client applications must run on a different Linux VM (MACHINE-B)

Assign-04 : The “Can We Talk” System

- The second chat-client application can run on the same Linux VM as the server (MACHINE-A)
- It is recommended that your chat-client application take at least 2 command-line switches as follows:

```
chat-client -user<userID> -server<server name>
```

Please see the section title “Finding the Server’s *Name*” below for more details and hints ...

5. The chat solution client’s UI only needs to be very basic and simple.
 - You will need to incorporate the use of the ***ncurses*** library to do this
 - The minimum UI requirement is show in Figure 1 at the end of the document. The basic UI consists of a prompt area (to allow a user to input a message) as well as a dedicated area on the screen to display the conversation. The message is sent to the recipient when the carriage return is pressed.
 - As you can see by the extra *ncurses* resource links (at the end of the assignment) – you are also able to draw windows on the screen. You could use this concept to *soup up* your UI (as shown in Figure 2)
6. Your solution must enforce and **parcel all messages** being input **at the 40 character boundary**
 - The user should be allowed to enter a message of up to 80 characters. When the user hits the 80 character input boundary – the UI should stop accepting characters for input
 - As mentioned, the chat program will parcel up and send the outgoing message into 40 character lengths.
 - So if the user happens to enter a message that is 56 characters in length before pressing ENTER (to send) – then one message of length 40 will be sent and another message of length 16 will be sent immediately following it.
 - An example of this kind of message is shown in Figure 2 below
 - As a usability factor – it would nice for your chat program to break the message at/near the 40 character boundary based on a space character (i.e. between words)
 - **QUESTION: Is this parceling best handled by the originating client sending the message? Or by the central server before it broadcasts to all clients? Or on the client receiving the message end?**
7. The client’s incoming message window of your UI:
 - Should display each of the incoming messages in a specific format. This format is detailed in the *Incoming Message Formatting* section below and as well is also shown in the sample screenshots.
 - **HINT: The fact that there are starting and stopping positions for fields in the message output should indicate the potential solution for you ...**
8. The client’s incoming message window should be able to show the history of at most the last 10 “lines” from the messages sent and received. Please note that a message is allowed to take up 2 “lines” in the output window (i.e. one line for each of the 40 character messages) ... this requirement indicates that a maximum of 10 lines of output are present in the message window before being scrolled ...
9. Your solution should be architected such that messages should be received as soon as they are sent. And as well, if a message is received when the user is typing another message, it must not interrupt the message being currently composed.
10. When the user enters the message “>>bye<<” – their client application will shut down properly.

Assign-04 : The “Can We Talk” System

- The server can end the thread that is connected to this client and as well clean up any information dealing with the client.
 - When the number of threads reaches zero in the server, it may shutdown properly
11. There should be no debugging messages being printed to the screen in your final client and server programs.
 12. Your solution must be programmed to handle any and all errors gracefully.
 13. Your solution must be programmed to handle any and all shutdowns gracefully.
 14. If there are command line parameters available in either your client or server programs then make sure that a **usage** message appears if the parameters are incorrect or missing.
 15. Include the completed A-04: Test Report in your submission
 - This document does not have to be part of your cleaned, submitted TAR file
 16. Make sure to submit your commented, cleaned TAR file to the appropriate drop-box by the due date and time

What About the Message?

You need to think about what needs to be sent in the message and how it will be formatted. When you are using sockets (or any low-level communication mechanism) one of the most exciting things is that you are in control of the messaging protocol! You get to create the format of, program and enforce your own communication scheme. So let's consider what needs to be placed in the actual message – what pieces of information need to be sent between the chatting parties?

- The IP address of the incoming message can be gotten through the accept() function call– or you could include it in your message
- What about the name (5 characters) of the person sending the message?
- What about the actual message contents? Should it be parceled on the client before the original send? Or on the server before broadcasting?

Please document your messaging scheme and data structure used to manage the multiple client connections in your server code file header comments.

- Make sure to include where/how the server will gain knowledge of the client IP and client's user name
- Be sure to include documentation on how the server will handle the >>bye<< message and shutdown / clean-up after the client. And as well clean-up after itself (when all clients are gone)
- Also ensure to indicate how your server data structure is managing the list of all clients – do this by describing the structure / elements / values being stored for each client

ncurses Resources

Here are a couple of extra resources that you can use to do more in-depth research on the functionality and capabilities of the ncurses library.

- [Installing ncurses on Your Own Linux Installation](#)
- [Programmer's Guide to ncurses](#)
- [Another Programmer's Guide \(of sorts\) - but with sample programs](#)
- [Simple "Hello World" Tutorial](#)

Assign-04 : The “Can We Talk” System

Incoming Message Formatting

Here is the layout of each of the chat messages being sent from and/or received by your program – as well, an explanation of the format follows.

```
0          1      2  2          6
1          7      5  8          9

XXX.XXX.XXX.XXX[AAAAA]>>aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa(HH:MM:SS)
--- IP ADDR --- -USER-----MESSAGE----- --TIME--
```

- Should display the IP address of the message’s source
 - in positions 1 through 15 of the output line
- Positions 16, 24, 27 and 68 should be spaces (as highlighted above)
- Should show the name of the person sending the message (max 5 characters) enclosed in square brackets (“[” and “]”)
 - in positions 17 through 23 (including the brackets)
- Should show the directionality of the message (i.e. >> for outgoing, << for incoming)
 - in positions 25 and 26
 - you should see >> on your client if you sent the message
 - you should see << on your client if the message came from another client
- Should show the actual message (again max 40 characters)
 - in positions 28 to 67
- Should show the a 24-hour clock received timestamp on the message enclosed in round brackets (“(” and “)”)
 - in positions 69 to 78 (including the brackets)
 - this should reflect the time that the client received the message from the server

Finding the Server’s Name

As was discussed during the Module on Sockets, the underlying TCP/IP protocol being used in this solution works by knowing the name / identity or address of the computer you are trying to communicate with.

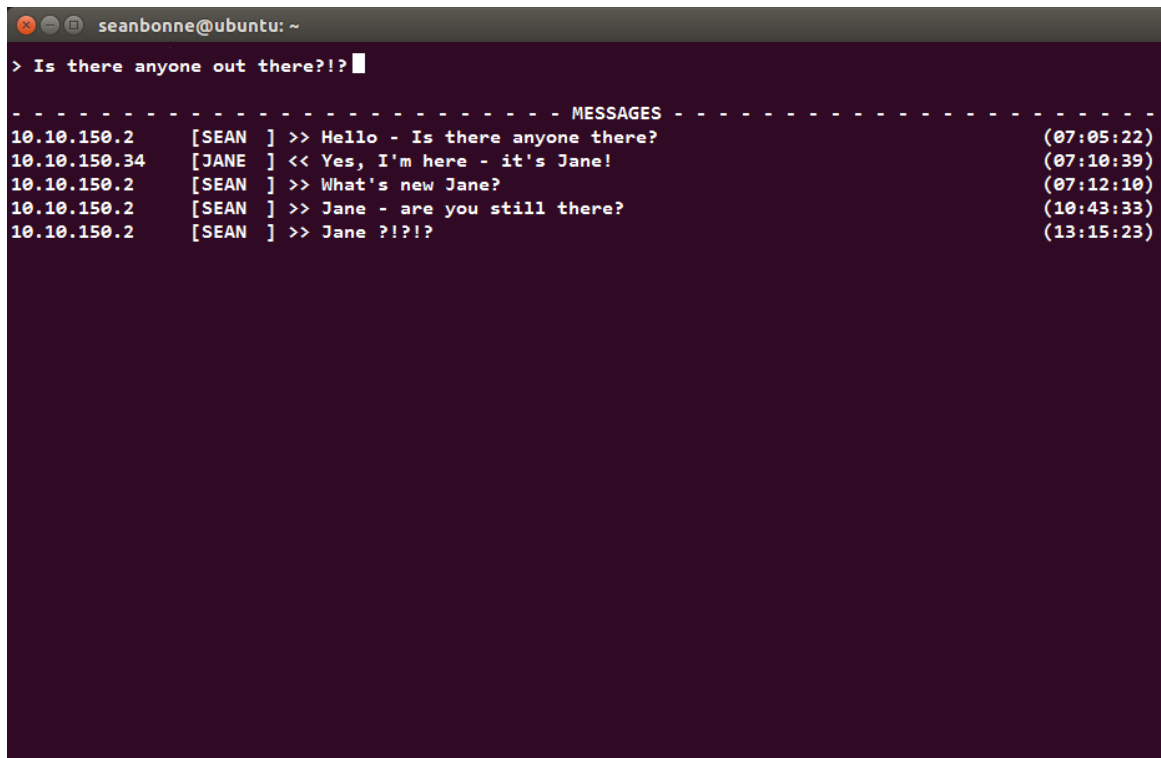
As you learned in OSF – computers can be given names (e.g. B03-213) and computers can be networked and be made part of a domain (e.g. conestogac.on.ca). If we were trying to communicate with this computer within the domain then – we could open a TCP/IP communication to B03-213.conestogac.on.ca – and we would find ourselves talking to that computer!

You also learned in OSF that each computer on a network is assigned an IP address (IPv4 and IPv6). This IP address also serves as the name of the computer when talking across any of the TCP/IP protocols. In Windows, you learned about the ipconfig command which allows you to find the computer’s IP address. In Linux the comparable command is ifconfig.

It is recommended that when constructing and running the CHAT-SYSTEM you find the server’s name by choosing the IPv4 address (referred to as the inet address (not inet6)) of the networking devices shown in the ifconfig command. This command will show you many networking devices – the ethernet networking connection device will most likely be called something like eth0.

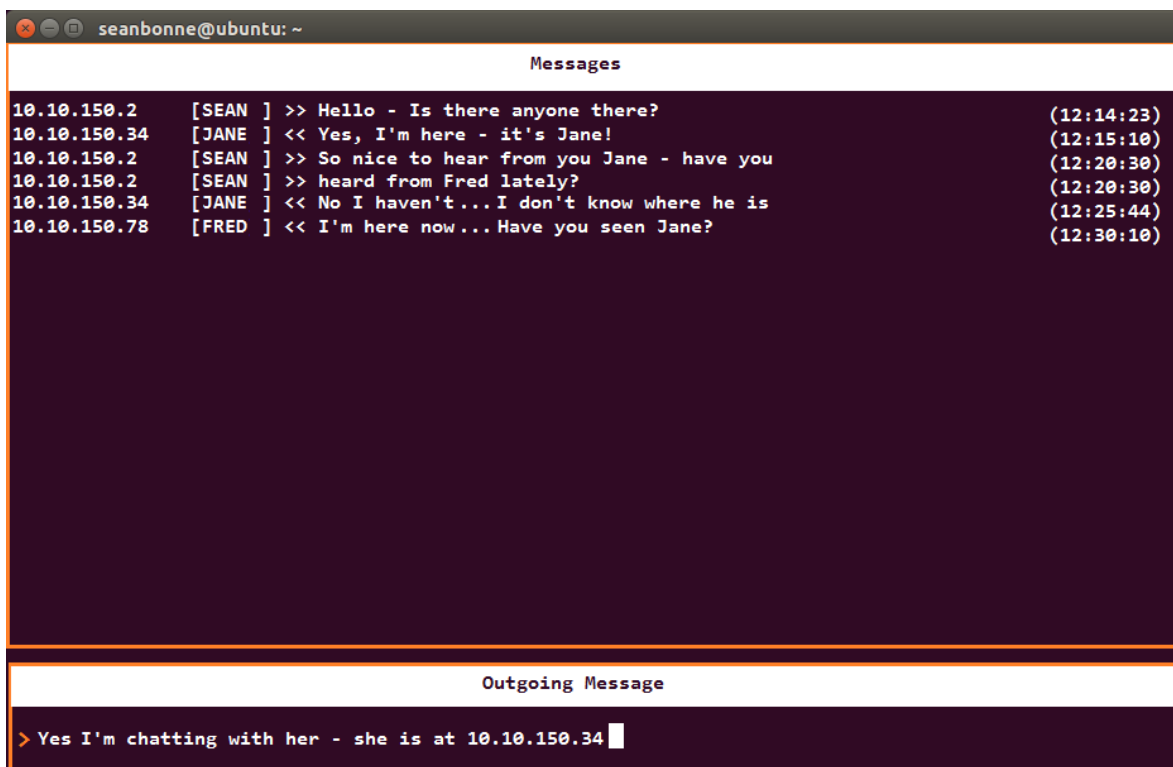
Assign-04 : The “Can We Talk” System

Example Client UI's



```
seanbonne@ubuntu: ~  
> Is there anyone out there?!?  
  
----- MESSAGES -----  
10.10.150.2    [SEAN ] >> Hello - Is there anyone there?           (07:05:22)  
10.10.150.34   [JANE ] << Yes, I'm here - it's Jane!           (07:10:39)  
10.10.150.2    [SEAN ] >> What's new Jane?               (07:12:10)  
10.10.150.2    [SEAN ] >> Jane - are you still there?    (10:43:33)  
10.10.150.2    [SEAN ] >> Jane ?!?!?                   (13:15:23)
```

Figure 1 : The Basic UI



```
seanbonne@ubuntu: ~  
  
Messages  
10.10.150.2    [SEAN ] >> Hello - Is there anyone there?           (12:14:23)  
10.10.150.34   [JANE ] << Yes, I'm here - it's Jane!           (12:15:10)  
10.10.150.2    [SEAN ] >> So nice to hear from you Jane - have you (12:20:30)  
10.10.150.2    [SEAN ] >> heard from Fred lately?               (12:20:30)  
10.10.150.34   [JANE ] << No I haven't...I don't know where he is (12:25:44)  
10.10.150.78   [FRED ] << I'm here now ... Have you seen Jane?    (12:30:10)  
  
Outgoing Message  
  
> Yes I'm chatting with her - she is at 10.10.150.34
```

Figure 2 : A More Advanced UI

Assign-04 : The “Can We Talk” System

Demonstration, Submission and Testing

You and your partner will need to demonstrate your solution (however much has been completed) in the lab period prior to your final submission the source code. This small demonstration counts towards the final mark on this assignment.

A sample Test Plan (title "Can We Talk" - Sample Test Plan) will become available at least one week prior to the demonstration in eConestoga. The idea behind the tests is twofold:

1. For you and your partner to gain experience in following a set of test specifications and to be exposed to examples of the different types of tests that may be run on a system solution
2. To gain experience in documenting, capturing required output and completing a Test Report. This type of activity is a crucial skill of any Software Engineer.

When you and your partner are ready to submit your final solution:

1. TAR up your system development directory structure and submit to the drop-box
2. You and your partner will also be required to run through a System Test Plan (titled A-04: System Test Plan) and complete the **A-04: Test Report**.
 - a. This final test plan will become visible in the course material after the demonstration of your code during the lab time.
 - b. Make sure to submit your completed **A-04: Test Report** into the drop-box along with your TAR file.