# The *Can We Talk* System

## Product Requirements Document

**Thorton P. Snodgrass III**

**TPI-III Industries**

# 1. Overall System Idea

## The Team

This assignment is best completed within a team (but can be completed individually if you like). You are allowed to work in teams up to 4 members. 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 teams. Remember that **all members** of the partnership **need to sign-up** and enroll in the **same group**.

## System 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.

### Coding 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)

# 2. System 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 all 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 about using a data structure on the server that might be used to hold all client information (the IP Address, the userID, etc.). Just remember that this structure may need to be shared among all the communication threads.
    - Your client application will also be written in ANSI C and will 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.
        - You do not need to concern yourself in the `chat-client` dealing with or handling the delete, backspace or arrow keys
    - The CHAT-SYSTEM functionality must be designed to operate across different 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 should be:
   - The chat-server application must run on one Linux VM (MACHINE-A)
   - One of the chat-client applications must run on a different Linux VM (MACHINE-B)
   - The second chat-client application can run on the same Linux VM as the server (MACHINE-A)
     - **COVID-19 Update : partners will not necessarily be working and debugging their solution together (i.e. physically) and on the same subnet – so the allowed configuration for testing is now:**
       - **The chat-server application must run on a Linux VM (MACHINE-A)**
       - **Both of the chat-client applications can run on the same Linux VM as the server (MACHINE-A)**
       - **Just make sure that your system design does not assume/depend on the fact that both clients and server are running on the same machine**
   - Your chat-client application must take 2 command-line switches as follows:
     ```
     chat-client –user<userID> –server<server name>
     ```
     Please see the section 4 "**Finding the Server's *Name***" below for more details and hints …
5. The chat solution client's UI only needs to be basic and simple.
   - You will need to incorporate the use of the ***ncurses*** library to do this
   - The minimum UI requirement is shown in Section 6, Figure 1.  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 Section 6, Figure 2)
6. Your system must enforce and **parcel all messages** being received by the chat-client **at a 40 character boundary**.
   - In the chat-client 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 must stop accepting characters for input
   - The design choice of where to parcel the message into 40 character "chunks" is up to you. You can choose to parcel it up in the chat-client  before sending the messages to the chat-server – or you can choose to send the 80 character message to the chat-server and parcel it up there before broadcasting to the chat-clients.
   - It doesn't matter which application does the parceling so long as the  40 character parceling is done.
     - 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 received by the clients and another message of length 16 will be received immediately following it.
       - An example of this kind of message is shown in Section 6, Figure 2
     - 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)

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 <u>at most</u> 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 <u>maximum of 10 lines of output</u> are present in the message window before being scrolled …

9. Your solution should be architected such that messages should be received in all clients as soon as they are sent. And as well, if a message is received when the user is typing another message, it <u>must not interrupt</u> the message being currently composed.

10. When the user enters the message ">>bye<<" – their client application will shut down properly.
    - 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 must shutdown properly and clean up any and all resources
    - This ">>bye<<" message must not be broadcast to the other clients

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 <u>A-04: Test Report</u> 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

17. In this System's development, you may choose to build/copy the `chat-client` and `chat-server` applications into the `Common/bin` folder, or you can leave them in their own respective `bin` folders – the choice is yours.

# 3. Chat System 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. You get to decide which application (i.e. the client or the server) performs which operations and enforces the required messaging guidelines.

Here's a reminder (as indicated in the *Incoming Messaging Format* section below) of the pieces of information that make up messages:

- Each message being received within a client needs to contain the sending client's IP address. So where best to get that information? It could be gotten from the client that originates the message and it could be part of the message being sent to the central server – or it could be gathered by the server as a client initially connects [through the `accept()` function call]. The design choice is yours …
  - The `accept()` function gives you an integer representation of your client's IP address – you'll need to investigate and use the `inet_ntop()` or (`inet_ntoa()`) function to translate the integer IP into a human readable IP … See Module-07 for more information.
- Each message being received within a client needs to contain the sending client's user-name (up to 5 characters). This information is provided on the `chat-client` command-line, but needs to find its way to the server (for use in message broadcasting) – so how will it get from the command-line of the client to the server? Will you have an initial / startup message that is sent from the client to the server in order to register?
- What about the actual message contents being sent from a client? Should it be parceled (into 40 byte chunks) 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 userID

- 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

## *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]_>>_aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa_(HH:MM:SS)
--- IP ADDR ---   -USER    --------------MESSAGE-----------------  --TIME--
```

- Should display the IP address of the message's source (i.e. the IP address of the client sending it)
  - in positions 1 through 15 of the output line
- Positions 16, 24, 27 and 68 must 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 <u>client received the message from the server</u>

# 4. Finding the Server's Name

As was discussed during the Module on <u>Sockets</u>, the underlying TCP/IP protocol being used in this solution works by knowing the (true) <u>name</u> or <u>IP Address</u> of the computer you are trying to communicate with.

In this assignment, when you launch the `chat-client` application you need to specify the server's *name* as a command-line argument. I want you to design your `chat-client` to accept both a server's (true) name and also a server's IP Address as this *name* command-line argument.

If you followed / used the default installation of Linux – then your computer's (true) <u>name</u> is most likely **ubuntu**. You can verify the (true) <u>name</u> of our computer by looking in the `etc/hosts` `file`. Chances are that this <u>name</u> is tied/bound too the *loopback* IP address of `127.0.0.1`.

As you learned in OSF each computer on a network is assigned an IP address (IPv4 and IPv6). This IP address can also serve 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 if `ifconfig`. If you execute the `ifconfig` command you will find the computer's IP Address by choosing the IPv4 address (referred to as the <u>inet address</u> (not `inet6`)) of the networking devices shown in the output. This command will show you many networking devices – the ethernet networking connection device that you are looking for will most likely be called something like `eth0` or `ens33`.

To be clear then about the chat-client application's -server command-line argument – you need to be able to handle and support the server's (true) name as well as the server's IP Address. For example:
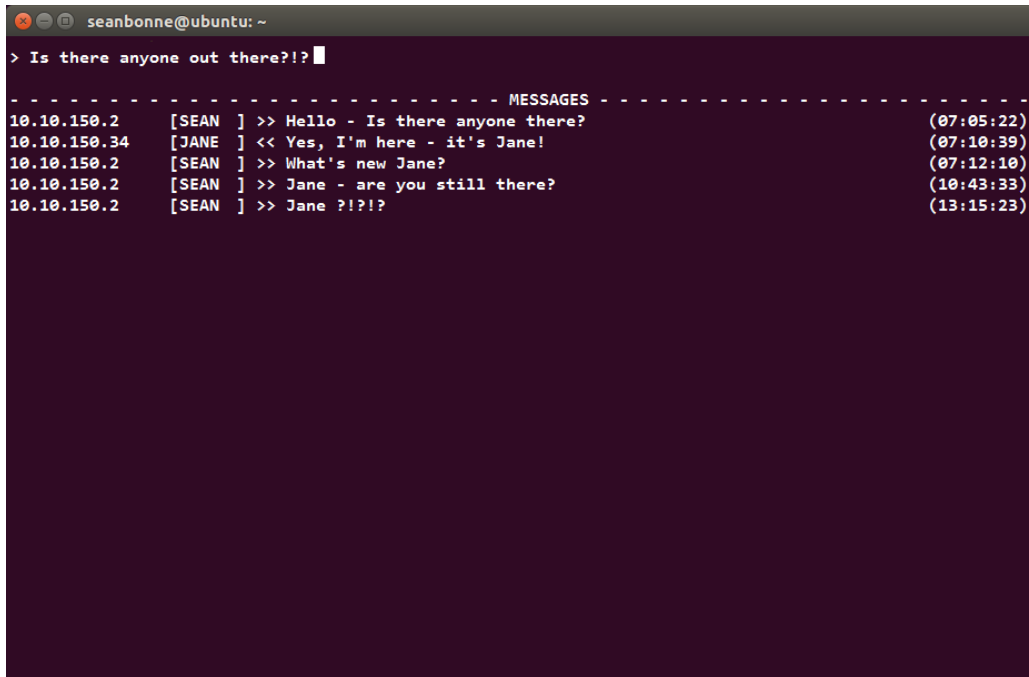
```
chat-client –userSean –serverubuntu          or
chat-client –userSean –server192.168.244.128
```

# 5. 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.
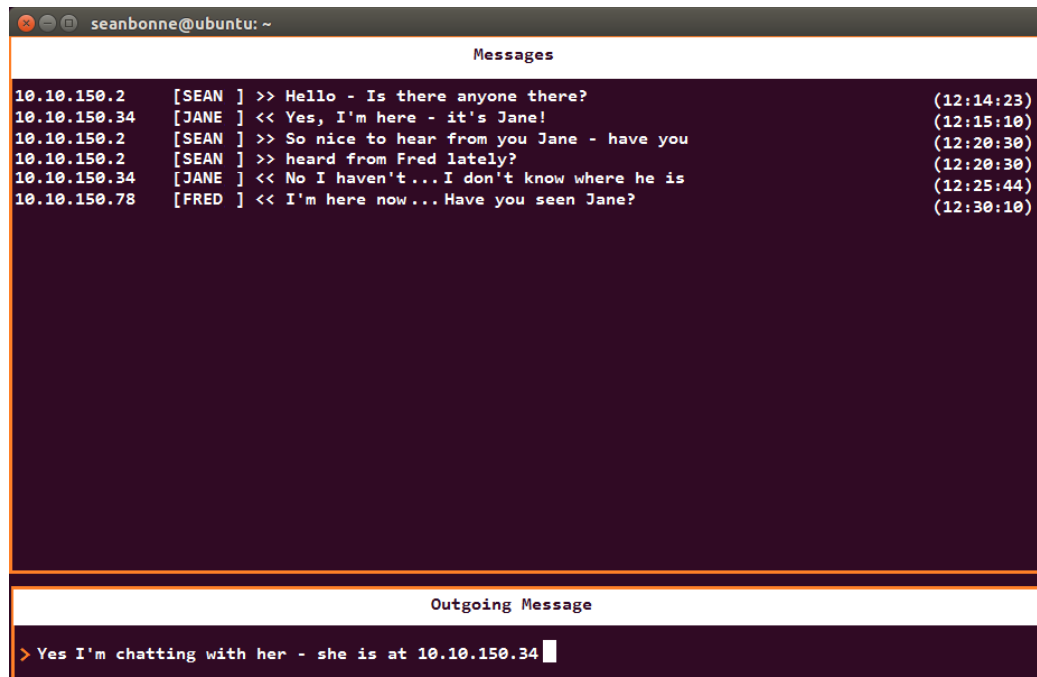
- [Installing ncurses on Your Own Linux Installation](#)
- [Programmer's Guide to ncurses](#)
- [Another Programmer's Guide (of sorts) - but with sample programs](#)
- [Simple "Hello World" Tutorial](#)

# 6. Example Client UI's



**Figure 1 : The Basic UI**



**Figure 2 : A More Advanced UI**

# 7. Submission and Testing

A sample Test Plan (title <u>"Can We Talk" - Sample Test Plan</u>) will become available at least one week prior to your submission.  The idea behind the tests is threefold:

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.

3. For you and your partner to be able to troubleshoot your solution and fix any errors or omissions prior to the actual submission of the system.

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

1. TAR up your *cleaned* 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 <u>A-04: System Test Plan</u>) and complete the **A-04: Test Report**.

    a. This final test plan will become visible in the course material at least three (3) days prior to the due date.

    b. Make sure to submit your completed **A-04: Test Report**  into the drop-box along with your TAR file.

# 8. System Design Hints and Thoughts

Through the years of using this assignment, students have asked a number of questions each year.  The following information is offered as a FAQ for you about design thoughts and choices from previous students.  As is the case with any system or solution that you need to create – the more thought you put into the design, the better, more robust your solution.

----- Message to Students

When student begin work on their *Can We Talk*  System design and solution a definite trend begins to happen with the questions and code that students begin to ask about.  Too many people begin to unknowingly over-complicate their system and server design.  This might happen due to simply *Googling* how-to information about threads, sockets, etc. and blindly *Frankenstein-ing* the code together in the hopes that it works.  But in reality, this behaviour and these choices begin to dig yourself a huge, scary hole.

In order to prevent you from entering these "scary places" – here is some information that I want you to read in order to ask yourself (and think about) what the server is and what really needs to be done ... the information below makes some statements and poses some questions to you about the requirements and about what you should know about top-down design approaches.

**Food for Thought** (Thinking about the system, the server and the client from a more simple (high level) perspective)

1.  Your chat-server needs to be continually listening for more connections
2.  Once a connection is made with the server - you need to spawn off a thread to interact with that client and receive the messages from that client
    a.  Should your server wait for each of the threads to finish and "join" them? Is that necessary? Or is it good enough in this server's design and purpose to simply spawn the thread and allow it to exit when the client "hangs up" and says goodbye?
3.  A system-level design choice that needs to be addressed and decided upon is this – when a client sends a message to the server:
    a.  Should that client also be designed to display that message in its own display window (meaning that the server must have been designed to not broadcast a message back to the client that sent it)?  Or should the client send the message and the server simply broadcasts to all clients (including the one that sent the message)?
4.  Regarding server broadcasting (or sending) each message that is received to all clients :
    a.  How do you do this? How can you spawn off a single task to handle this sending of the messages? [HINT: think about sewing - you need a needle and _____ ]
    b.  The question is where should you spawn off this task? In relation to the spawning of the client threads that is …
5.  As is the case with most system design -- what about the data?  What is the data being maintained and managed in the server?
    a.  You need some sort of "master list" of client connections, IP Addresses and userIDs - in order to know who you need to broadcast to.
        i.  When would you add to this list? When a new client connection comes in I would think...
        ii.  When would you shrink/maintain this list? I would think that you would do that when the individual client says "goodbye" and close their connection
    b.  Would it be a good idea to maintain some sort of data structure to handle the list of incoming messages (their content and originator client ID) until they are broadcast? Would this list need to be able to grow and shrink? You bet it would … it would grow with each client's incoming message and shrink once it was broadcast.  Following the server threads and design idea I've been speaking about here - that would mean that this queued message list would grow (be added to within the various client handling threads) and would shrink within the broadcasting thread.
        i.  I don't believe that my information oversimplifies the server design - basically your server needs

to create/maintain (grow and shrink) a couple of lists (for client connections and incoming messages) and must have one thread for each client connection - and possibly (as suggested above) another thread to handle the broadcasting.

ii. Given that these lists need to be accessible by potentially all threads - where should they be created? [Meaning are they local variables to the various methods? Or is this finally a case that justifies the need for GLOBAL variables]?

These are just my thoughts - and by no means are meant to give you the one and only design that you (and your team) could use to satisfy the assignment requirements ... they are merely offered to you as ideas for you to think about and perhaps simplify your design before you code yourself into a corner...