TCP Simple Broadcast Chat Server and Client
ECEN 602 Network Programming Assignment 2
Due Oct. 8, 2018 NLT 5:00 pm Central

Introduction

In this assignment, you will implement the client and server for a simple chat service. The goals of the assignment are to develop a more sophisticated socket programming application than Assignment 1, reading and implementing a more complicated protocol specification, and participating in more extensive interoperability testing. The basic assignment is worth 100 points, but you can earn up to 50 additional points by implementing the Bonus features (see Bonus section below).

Assignments must be written in C or C++, and they are to be compiled and tested in a Linux environment. Because the goal of the exercise is to understand system calls to the socket layer, you are prohibited from using any socket "wrapper" libraries; however, you may use libraries for simple data-structures. It is also acceptable, to use the Unix Network Programming, Vol. 1, 3rd Edition "wrappers" for the basic networking function calls (e.g., `socket`, `bind`, `listen`, `accept`, `connect`, `close`, etc.…). These "wrapper" functions check for error returns from the network functions (see Supplementary References).

Protocol Specification

1. Overview

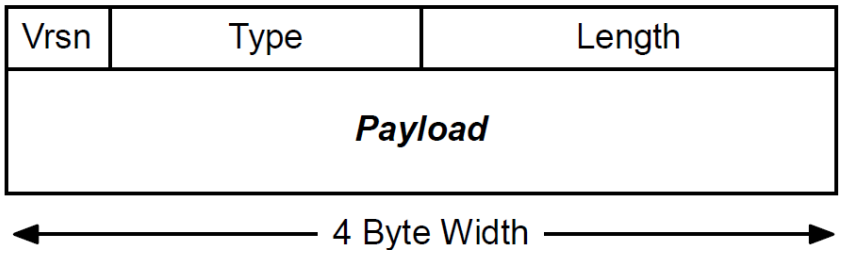
The Simple Broadcast Chat Protocol (SBCP) is a protocol that allows clients to join and leave a global chat session, view members of the session, and send and receive messages. An instance of the server provides a single "chat room," which can only handle a finite number of clients. Clients must explicitly JOIN the session. A client receives a list of the connected members of the chat session once they complete the JOIN transaction. Clients use SEND messages to carry chat text, and clients receive chat text from the server using the FWD message. Clients may exit unceremoniously at any time during the chat session. The server should detect a client exit, cleanup the resources allocated to that client and notify the other clients. Additionally, the client program will be able to detect idle clients and notify the server.

Some of the above mentioned features are bonus features, which are covered in the last part of the assignment.

2. Message Format

All SBCP messages share a common header format. The format for an SBCP message is:

SBCP Message



SBCP Header Fields

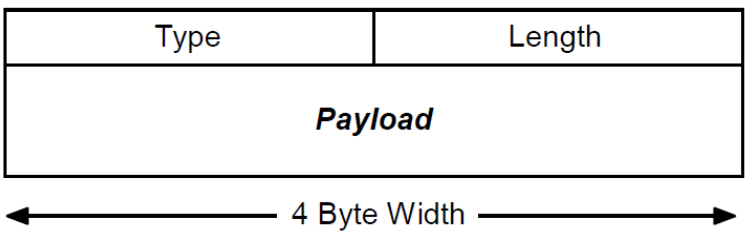
Field	Size	Description
Vrsn	9 bits	protocol version is 3
Type	7 bits	indicates the SBCP message type
Length	2 bytes	indicates the length of the SBCP message
Payload	0 or more bytes	contains zero or more SBCP attributes

SBCP Mandatory Header Types

NAME	Type	Description
JOIN	2	client sends to server to join the chat session
SEND	4	client sends to server, contains chat text
FWD	3	server sends to clients, contains chat text

The format for the SBCP attribute is given below. Note that zero or more SBCP Attributes are the ”payload” of an SBCP Message.

SBCP Attribute



SBCP Attribute Fields

Field	Size	Description
Type	2 bytes	indicates SBCP Attribute type
Length	2 bytes	indicates length of the SBCP attribute
Payload	0 or more bytes	contains the attribute payload

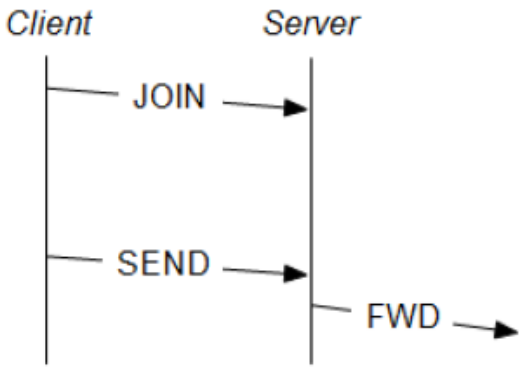
SBCP Attribute Payloads

Name	Attr. Type	Size	Description
Username	2	1-16 bytes	nickname client is using for chat
Message	4	1-512 bytes	actual chat text
Reason	1	1-32 bytes	text indicating reason of failure
Client Count	3	2 bytes	number of clients in chat session

3. Message Sequences

The following is a very brief sequence diagram showing a client JOIN a chat, SEND a message, which is broadcast (FWD) to any other client participants.

Mandatory Sequence



Message Attribute Table

SBCP	Sender	Receiver	Attributes
JOIN	Client	Server	username
SEND	Client	server	message
FWD	Server	client	message, username

4. Client Operations

#	Requirement
1	The client should connect to the server using the IP and port supplied on the command line. <./client username server_ip server_port>
2	The client should initiate a JOIN with the server using the username supplied on the command line.
3	The client should display a basic prompt to the operator for displaying received FWD messages, and typed message that are sent (SEND).
4	The client will need to use I/O multiplexing (select) to handle both sending and receiving of messages.
5	The client should discard any message that is not understood.

5. Server Operations

#	Requirement
1	The server should start on the IP and port supplied on the command line. <./server server_ip server_port max_clients>
2	A SEND received by the server will cause a copy of the MESSAGE text to be sent in FWD to all clients except the original sender.
3	A server may only accept JOIN SBCP messages from unknown clients. The server will not allow multiple clients to use the same username.
4	Clients may leave the chat session unceremoniously. The server should cleanup its resources (close socket, make the username available).
5	The server should discard any messages that are not understood.
6	The implementation can be an iterative server. A distinct TCP connection is used for each client-server transaction.

6. Input/Output Multiplexing – In this assignment, we need to handle several inputs at the same time in both the client and the server. Unlike Network Programming Assignment 1, where the server forks a child process to handle each request, a straight forward way to attack this assignment is with a single server process that can deal with a listening socket for new clients and additional connected sockets, i.e., one socket for each client that is connected. The client also has to deal with multiple inputs as it will be waiting on standard input for the local user to type a chat message, and at the same time waiting on a TCP socket for the server to send FWD messages from other chat clients.

   Input/Output (I/O) multiplexing is typically used in networking applications in the following scenarios [UNP vol. 1, 3rd Ed., Ch 6]:

   - When a client is handling multiple descriptors (normally interactive input and a network socket), I/O multiplexing should be used. This is the scenario for our chat client.
   - It is possible, but unusual, for a client to handle multiple sockets at the same time. The prime example of a client that does open multiple sockets at the same time is an important one, however, a web browser. Web browsers typically open six persistent TCP connections to a web server.
   - If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally used. This is the scenario for our chat server.
   - If a server handles both TCP and UDP connections, I/O multiplexing is normally used.
   - If a server handles multiple services and perhaps multiple protocols, I/O multiplexing is normally used. An example of such a server is the `inetd` super-server daemon in Unix systems (in recent years `inetd` has been replaced by similar types of super-servers that provide better security). The `inetd` daemon listens for connections for multiple services (e.g., FTP, POP3, echo, etc.), and it then starts single or multi-threaded programs to service the request.

   The most common model for I/O is the *blocking I/O model*, which is what you used in the first network programming assignment. By default, all sockets are blocking (e.g., a `read` socket call blocks until there is data, and an `accept` call blocks until a client connects to the server). In a blocking socket, you issue a socket operation, say a `read`, and then your process *blocks* (i.e., the process is put to "sleep") waiting for data. When data is received, the kernel copies the data into your process, and your program is awakened and starts executing after the `read`. The same thing happens when you call system I/O routines like `fgets` on standard input. Consider our chat client, which needs to simultaneously issue an `fgets`

for input from the local chat user and also issue a `read` to the socket connected to the server. If `fgets` is called first, your program will block waiting for the local user to type a message. If while you are waiting for `fgets` you receive a TCP message from the server, you will not be able to process it until `fgets` returns. A similar scenario occurs if you issue a socket `read` first. What we need for the chat client is some way to wait for either standard input *or* socket data to be received. The `select` system function provides this functionality and more.

The `select` system function allows a process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one or more of these events occurs or when a specified amount of time has passed. For example, we can call `select` and tell the kernel to return only when one or more of the following events has occurred [UNP vol. 1, 3rd Ed., Ch 6]:

- Any of the descriptors in the set {1, 4, 5} are ready for reading,
- Any of the descriptors in the set {2, 7} are ready for writing,
- Any of the descriptors in the set {1, 4} have an exception condition pending, or
- A time of 10.2 seconds has elapsed since calling `select`

You can find an example of using `select` in Beej's Guide and another example at http://www.binarytides.com/multiple-socket-connections-fdset-select-linux/. Also, Chapter 6 in UNP vol. 1, 3rd Edition has detailed information on the `select` function in network programming. The `select` function is a little complicated to set up, and there are four other system functions you will need to use to (1) configure the sets for reads, writes, and exceptions, and (2) to test for which events are set when select returns.

To use `select` in the chat server, for example, you first call `socket`, `bind`, and `listen` to set up your listening socket. Next, you set up the data structure for `select` and call `select` to wait for an event. Calling `select` causes your program to block until one of the events for which you are waiting is set. When `select` returns, you need to check what event took place (e.g., data is available to read on one of your sockets in the server) and then process the event (e.g., issue a `read` for the socket that has data). If there is activity on the listening socket, you will need to call `accept` to connect the new client. A new client will require that you add this new *connected socket* to your list of connected sockets, and that you update the `select` data structure before the next call to `select`. In this assignment, you can get away with just issuing `select` for descriptors that are ready for reading or accepting a connection and not worry about writing.

A socket is ready for *reading* or *accepting* a connection if any of the following conditions are true [UNP vol. 1, 3rd Ed., Ch 6]:

- The number of bytes of data in the socket receive buffer is greater than or equal to the current size of the "low-water mark" for the socket receive buffer. A read operation on the socket will not block and will return a value greater than 0. You can set the low-water mark using the `SO_RCVLOWAT` socket option. The low-water mark defaults to 1 for TCP and UDP sockets. You do not need to change the default for this assignment.
- The read half of the connection is closed. A read operation on the socket will not block and will return a 0 (i.e., EOF). For example, your server will need to check your `read` return value to determine if a client is exiting the chat session.
- The socket is a listening socket, and the number of completed connections is nonzero. An `accept` on the listening socket will normally not block. For this assignment, you do not have to worry about the special case where it can block.
- A socket error is pending. A read operation of the socket will not block and will return an error (-1) with `errno` set to the specific error condition.

A socket is ready for `writing` if any of the following conditions are true [UNP vol. 1 3rd Ed., Ch 6] (For this assignment, you can get away with not worrying about write's blocking since the amount of data you are sending is so small):

- The number of bytes of available space in the socket send buffer is greater than or equal to the size of the low-water mark for the socket send buffer and either: (1) the socket is connected or (2) the socket is a UDP socket. The low-water default for TCP and UDP is normally 2048 bytes.
- The write half of the socket of the connection is closed. A write operation to the socket will generate a `SIGPIPE` signal, which by default will terminate the process.
- A socket error is pending. A write operation on the process will not block and will return an error (-1) with `errno` set to the specific error condition.

A socket has an exception condition pending if there is out-of-band data for the socket or the socket is still at the out of band mark [UNP vol. 1, 3rd Ed., Ch 6]. You will not need to wait on an exception condition for this assignment. TCP does have something called "Urgent" data, but it is not used very often.

As was noted in class, a socket `read` may not return all the data in a single

call to `read`. In this assignment, because of the small message sizes, you will probably receive an entire message in a single `read`, but good network programming practice would be to make sure you have a complete message before processing. Each of the SBCP messages includes in the header a Length field you should compare to the number of bytes read from the socket. If the number of bytes read is less than expected, ideally you should wait to see if the rest of the message shows up in a future call to `select`. If the rest of the message does not show up in a few seconds, you could then discard the message. Worrying about multiple reads and having a timer complicates the program, but it is what you would want to do in production code. You are not required to implement this waiting feature, but you must check the Length field and discard a message where the Length does not match the number of bytes read from the socket.

There are other ways than using `select` to implement the required server and client functionality [UNP vol. 1, 3rd Ed., Ch 6]. For example, *multithreading with blocking I/O* closely resembles the I/O Multiplexing model above except that instead of using `select` to block on multiple file descriptors, the program uses multiple threads (one per file descriptor), and each thread is then free to call blocking I/O socket calls. Another option is to use *signal-driven I/O* where we implement a signal handler and then have the kernel notify us with a `SIGIO` signal when the descriptor is ready. Finally, some systems have calls that are similar to `select`, including `poll`, `pselect`, and the `libevent` event notification library mentioned in Beej's Guide that runs on Linux, BSD, Mac OS, Solaris and Windows. You are welcome to use the method your like the best. If you have never written a program using one of these approaches, however, I think `select` is the most straight forward.

Bonus Features

For the assignment you must implement the Mandatory features; however, you may implement the IPv4/6 Bonus and one or both of the other two bonus features for extra credit (the total possible additional bonus points is 50 = 10 + 20 + 20) . The bonus features that may be implemented include:

IPv4 and IPv6 (10 points)

Write your code so that it will work with either IPv4 or IPv6 networks. You will not be able to test the IPv6 functionality since the ECE Linux machines do not have IPv6 turned on. Nevertheless, we should all be writing network code today that works for both IPv6 and IPv4.

Feature 1 (20 points)

New message types: ACK, NAK, ONLINE, and OFFLINE. The server uses the ACK message to provide an explicit confirmation of the client's JOIN. The ACK includes a single Client Count attribute and zero or more Username attributes (Alternatively, the ACK may include a message attribute containing the client count and the list of clients as a string to be displayed at the client side. In this case, assume that all the usernames will fit into a single message.). The server may limit the number of active clients and refuse a JOIN with a NAK. The NAK includes a Reason attribute. Finally, the server can indicate when clients enter and leave the chat session through the ONLINE/OFFLINE messages. These messages contain a single username attribute indicating the client which is entering/leaving the chat.

Feature 2 (20 points)

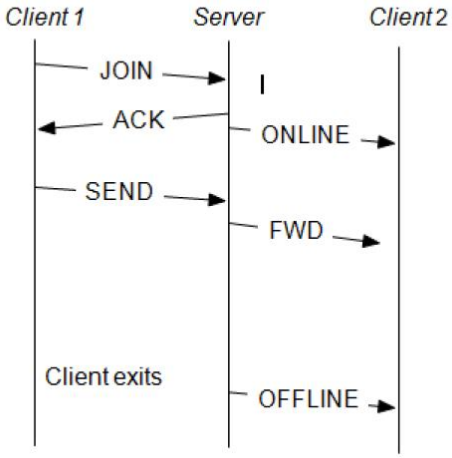
The client should keep track of the time for which a user has not used the chat session. If this interval is more than 10 seconds (fixed interval for this assignment), the client should send an IDLE message to the server. The server should in turn send an IDLE message to all other clients with the single username attribute indicating the client which is idle.

In summary, a client can be in one of three states: Online, Offline or Idle. The capacity of the chat-room is the total of Online and Idle clients. When a client goes offline, the server is responsible for cleaning up the resources.

SBCP Bonus Header Types

NAME	Type	Description
ACK	7	server sends to client to confirm the JOIN request
NAK	5	server sends to client to reject a request (JOIN, etc)
ONLINE	8	server sends to client indicating arrival of a chat participant
OFFLINE	6	server sends to client indicating departure of a chat participant
IDLE	9	Client sends to server indicating that it has been idle. Server sends to clients indicating the username which is idle.

Bonus Message Sequence



SBCP	Sender	Receiver	Attributes
ACK	server	Client	client count, username(s)
NAK	server	Client	Reason
ONLINE	*server*	*Client*	*username*
OFFLINE	*server*	*Client*	*username*
IDLE	*server*	*Client*	*username*
IDLE	*client*	*Serve*	*<none>*

1. The client should display the client count and username list of an ACK to its operator.
2. The server should acknowledge all well-formed JOIN requests with an ACK. If the request would cause the server to exceed its client limit then it should send a NAK with an appropriate reason.
3. The ACK should contain a client count attribute, and a list of username attributes. The client count should be inclusive of the requestor, while the username list should be exclusive.
4. Upon a successful JOIN, the server should send an ONLINE to its other clients. When a client gets disconnected, the server should send OFFLINE to its remaining clients.
5. When a client has not sent a chat message for 10 seconds, it is said to be idle. The Client should send an IDLE message without any attributes.
6. When the server receives the IDLE message, it should form another IDLE message with a username attribute. The username should be that of the idle client.
7. When a Client receives an IDLE message, it should display the username in the message attribute to stdout stating that “<username> is idle”.

Submission Guidelines

1. My expectation is that each team member will contribute equally to the network programming assignments. My recommendation for this assignment is that one of you develop the client and the test cases, and the other develop the server. Please include a statement in your README that describes the role of each team member in completing the assignment.
2. Test cases – Please develop a set of test cases to demonstrate that your client and server work correctly. Submit a short report describing how you tested your final implementation, and a description of your test cases along with screen captures of the test cases to document correct operation. At a minimum, include the following test cases: (1) normal operation of the chat client with three clients connected, (2) server rejects a client with a duplicate username, (3) server allows a previously used username to be reused, (4) server rejects the client because it exceeds the maximum number of clients allowed, and (5) separate test cases for any of the bonus features you implement.
3. The network programming assignments are due by 5:00 pm on the due date.
4. Your source code must be submitted to Turnitin.com using eCampus by 5:00 pm on the due date. Turnitin.com is plagiarism detection software that will compare your code to files on the Internet as well as your peers' code. Additional details on how to submit your code will be provided by the TA.
5. All programming assignments must include the following: makefile, README, and the code.
6. The README should contain a description of your code: architecture, usage, errata, etc.
7. Make sure all binaries and object code have been cleaned from your project before you submit.
8. Your project must compile on a standard Linux development system. Your code will be graded on a Linux testbed.
9. Explanation on the submission procedure will be provided by the TA.

Notes

1. Servers should not allow multiple clients with the same username to join a chat session.
2. Once a client exits, its username must be reusable by any new client.
3. The best way of learning new system calls is to build very small programs that demonstrate simple things. Unless you are familiar with the subject matter, you should experiment at first.
4. Once you are ready to begin you should start with a conceptual model/ architecture of your client and server? What is your basic data model? What

are the major decision points in the code? You should sketch these thoughts out on paper before you begin (it is also required with submission of the project).

5. Test your code as you write it (unit testing is a great thing).
6. Coding style is very important. People, including yourself, are more likely to understand your code if it follows a style that is simple and self consistent. We will not enforce any specific style; however, you should spend a little time on deciding on a style and sticking with it. Two good references are: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml, and http://lxr.linux.no/#linux+v2.6.35/Documentation/CodingStyle. What you pick is not as important as being consistent through your code.
7. A makefile is a standard tool for any software project. You will need to create one to build and clean your project. There are many online resources that can get you started.
8. A great way to do random testing is to partner with another team and perform interoperability testing. Your client should work with their server, and their server should work with your client. This is guaranteed to reveal implementation bugs, as well as possible design defects.
9. Come to the Recitation and/or the TA's Office hours to make sure you understand the assignment or are having trouble making progress.
10. You must comment your code.
11. Be sure to check all the error returns from the network functions and to check all buffer writes to avoid buffer overruns. You don't want to be the network programmer responsible for the next "Heartbleed" bug.