The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2021
Assignment 4 (version 1.1)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 3:59pm 28th May, 2021

Specification changes since version 1.0 are shown in red and are summarised at the end of the document.

Introduction

The goal of this assignment is to test and demonstrate your knowledge of socket programming and thread multiprocessing, and to further strengthen your general C programming capabilities through more advanced program design and implementation.

Put simply, this assignment is a networked/threaded version of the chat system you created for Assignment 3. You are welcome to re-use any code from your previous submission, although the core connection and thread management code will all be new.

You are to create two programs, one server (`server`) and one client (`client`), which together form a networked chat messaging system. The server is responsible accepting network connections from clients and broadcasting messages between clients participating in the chat. The clients will connect to the server, and present a line-based interface allowing messages to be sent and received, and a few chat commands. The clients and server will communicate using a text-based messaging protocol over TCP/IP. This protocol is similar but not identical to the protocol from Assignment 3.

Because we are using threads and sockets, the chat system is much more dynamic than Assignment 3. Clients can join and leave the chat at any time. You interact with the client via `stdin`/`stdout`, and the use of sockets means you can actually use it to chat with your classmates on moss. It's basically IRC-lite.

Student conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design and coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository, and do not allow others to access your computer - you must keep your code secure.

Uploading or otherwise providing the assignment specification to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and they cooperate with us in misconduct investigations.

You may use code provided to you by the CSSE2310/CSSE7231 teaching staff **in this current semester** and you may use code examples that are found in man pages on moss. If you do so, you **must** add a comment in your code (adjacent to that code) that references the source of the code. If you use code from other sources then this is either misconduct (if you don't reference the code) or code without academic merit (if you do reference the code). Code without academic merit will be removed from your assignment prior to marking (which may cause compilation to fail) but this will not be considered misconduct.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site: `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`

Specification

Server

Command Line Arguments

Your program (`server`) is to accept command line arguments as follows:

```
./server authfile [port]
```

where

- *`authfile`* is the name of a text file that contains a single line authentication string, which must be provided by a client to be permitted to connect
- *`port`* an optional argument describing the port number to listen on. If no port is specified on the command line, the server should create an ephemeral port. If *`port`* is 0 then the server shall use an ephemeral port.

The format of the `authfile` is as follows:

- One newline terminated line containing a string of any length
- An empty authfile implies no authentication

e.g.

```
ToPSeKrit
```

See Table 1 for error messages and exit codes for the server.

Server behaviour

The server will create and listen on a TCP/IP socket (either the port specified on the command line, or an ephemeral one if none is specified) and accept connections from clients.
General notes on required server functionality:

- the server must use threads and sockets to handle concurrency and communication.
- the server must be able to accept new client connections at any time, and permit clients to disconnect at any time.
- the server must accept an arbitrary number of clients (up to system limits of maximum sockets/threads etc, which we will not test for). A session with a single client is also valid, if somewhat lonely.
- the server will need to track the state of each client independently, and communicate using the protocol described in Table 3, ensuring the correct messages are sent according to the client state.
- the server must emit the listening port number (followed by a newline) on `stderr` when it is ready to accept connections on that port.
- the server must echo all chat messages to `stdout` in the format specified below.
- upon receiving the `SIGHUP` signal, the server should emit to `stderr` statistics regarding the current chat session. The exact format is described below.
- as clients quit or are kicked the server must detect this, free up resources and no longer attempt to communicate with them. (clarified from spec v1.00)
- any poorly formatted or invalid messages received by the server must be silently ignored.
- the server should expect no input on `stdin`.
- once the server is listening for connection requests it should never terminate unless it receives SIGQUIT/SIGKILL. We will not test error codes in this case.

The following shows example output captured from a server session.

```
$ ./server auth.txt 0
45962
(fred has entered the chat)
fred: Hello is anybody here?
(wilma has entered the chat)
wilma: Hi there
fred: OHAI wilma
(barney has entered the chat)
wilma: I'm outta here
(wilma has left the chat)
barney: what's her problem?
fred: I think it's because you smell bad
(barney has left the chat)
(fred has left the chat)
(wilma has entered the chat)
wilma: Oh good, he's gone
...
```

Listing 1: Sample server `stdout` and `stderr` output

Authentication and name negotiation

Upon accepting a connection from the listening socket, the server shall issue an "`AUTH:`" challenge, and expect an "`AUTH:`" response back:

```
(server) AUTH:
(client) AUTH:string
```

If the auth string received from the client does not match the server's auth string, then the connection is terminated immediately. No retries are permitted.

Upon succesful authentication, the server replies with an "`OK:`" message

Once authentication is complete, name negotiation begins.

```
(server) WHO:
(client) NAME:name
```

If there is already a user in the chat with that name, the server responds with "`NAME_TAKEN:`", and repeats the "`WHO:`" message. This process continues until the client specifies a unique name (or the client disconnects).

```
(server) WHO:
(client) NAME:Fred
(server) NAME_TAKEN:
(server) WHO:
(client) NAME:Barney
(server) OK:
```

Once a unique name is received, the server responds with "`OK:`", and the client is officially in the chat.

The server immediately broadcasts an "`ENTER:`*`name`*" message to all connected clients (**including the newly connected one**).

At this point the server emits `(`*`name`*` has entered the chat)` to its `stdout`.

Names provided by clients must be at least one character long. Zero-length names from clients shall receive a `NAME_TAKEN:` response (new in spec revision 1.1)

Messages and commands accepted by the server

Once a client is connected to the server, the server waits for messages from the client. Valid messages are as follows:

- `SAY:`*contents* – the client has something to say. This message must be rebroadcast (in a `MSG:` message) to all connected clients (including the sending client). See the client specification below for details of how this message must be formatted.
- `LIST:` – the client is requesting a comma seperated list containing the names of all current chatters, **in lexicographical (alphabetic) order**. The server's reply is in the form:
  - `LIST:name1,name2,name3` (Note there is no trailing comma in the list of names).
- `KICK:`*name* – client requests to kick the named user. If no such user is present, do nothing. A client may kick itself. Upon receiving a `KICK:` request, the server shall send a "`KICK:`" message **only to the named client** (if it exists).
- `LEAVE:` - the client is leaving. The server should immediately close the network connection to that client and clean up any threads and data structures associated with that client. No further activity is permitted on that socket. Upon receiving a `LEAVE:` message from a client, the server must broadcast a `LEAVE:` message to all other clients indicating that a particular client has left:
  - `LEAVE:`*name*

  At this point the server emits (*name* `has left the chat`) to its `stdout`.

Server signal handling

Upon receiving `SIGHUP`, the server shall emit to `stderr` a table of statistics relating to the current session. Statistics reported are:

- A list of all currently connected clients (in lexicographical order) and the number of `SAY:`, KICK: and `LIST:` messages received by the server from that client
- The total number of certain message type received from all clients **since the server started**

This output is in the following format (to `stderr`):

```
@CLIENTS@
Barney:SAY:25:KICK:4:LIST:0
Fred:SAY:25:KICK:0:LIST:10
Wilma:SAY:100:KICK:3:LIST:10
@SERVER@
server:AUTH:25:NAME:10:SAY:465:KICK:25:LIST:10:LEAVE:5
```

Notes:

- If there are no currently connected clients, the server should still output the `@CLIENTS@` header.
- The server statistics won't necessarily be the sum of client statistics, as earlier clients may have disconnected before the `SIGHUP` is received.
- Subsequent `SIGHUP` signals should report cumulative statistics – **do not reset statistics counters** upon receiving `SIGHUP`.
- If a client leaves and subsequently reconnects, their statistics are reset – you don't have to remember whether they were previously connected or not.
- Count all messages of a particular type, regardless of whether they were actioned. For example a `KICK:` message sent about a non-existent user should still be counted, even if no such user was present at the time. Similarly `NAME:` and `AUTH:` attempts that were subsequently rejected should all be counted. This should make your life a bit easier.

Client message rate limiting

Client-handling server threads should sleep for 100 milliseconds after processing a client's latest message, in order to prevent one client from swamping the chat. You can use the `usleep()` C library function to achieve this. We will not test this delay precisely, but you need to include it.

Unexpected client disconnections

(Updated in spec v1.1)

Sometimes clients will just disappear – for example if the process dies. Your server must handle this gracefully by detecting `eof` or an error when communicating with a client, and emitting a `LEAVE:name` message to all remaining clients. If you are using `FILE *` streams, then `ferror()` will be useful, otherwise look for error returns from `read()` and `write()`. The server must clean up any threads and data structures etc. associated with that client.

Your server should probably block `SIGPIPE` – you will get them ~~sometimes~~ when clients disconnect unexpectedly. If you don't block it, your server will terminate.

Clients

Command Line Arguments

Your program (`client`) is to accept command line arguments as follows:

`./client name authfile port`

where

- *`authfile`* is the name of a text file that contains a single line authentication string in the same format as used by the server.
- *`name`* is the name this client wants to use. See below for how to handle `NAME_TAKEN:` messages.
- *`port`* is the port number on the server (`localhost`) to connect to.

See Table 2 for error messages and exit codes for the client.

Client behaviour

In general, clients communicate with the server over a socket, and take line input from `stdin` which is used to send messages and other chat commands to the server.

The client shall first attempt to connect to the identified socket on `localhost`. If this fails, the client exits with a communications error (see Table 2 for details).

Apart from error messages, the client emits all of its chat log/output to `stdout`.

An example log from a client session may look something like the following. Note that Fred's messages are sent back to him by the server and displayed like any other chatter. When you run `client` on the console, you will see your messages twice - once as you type them in because of terminal echoing, and once when they come back as a ~~"`SAY:`"~~ "MSG:" message. The local terminal echo is ommitted in the following example.

```
$ ./client Fred authfile 42195
(Fred has entered the chat)
Fred: Hey is anybody here?
(current chatters Wilma,Fred)
Wilma: Oh hey fred, how's it going?
Fred: Good thanks U?
Wilma: Oh not bad - Barney was here earlier, he smells bad
(Barney has entered the chat)
Wilma: Yay - Barney's here
Fred: Hi Barney
(current chatters: Barney,Fred,Wilma)
(Wilma has left the chat)
(current chatters: Barney,Fred)
(Wilma has entered the chat)
(current chatters: Barney,Fred,Wilma)
```

Note the "`(current chatters)`" output - these were in response to Fred sending "`LIST:`" commands to the server.

Client `stdin` handling

The client shall read lines from `stdin`, and process them as follows:

- input starting with an asterisk ('`*`') is to be treated as a verbatim command to be sent to the server (minus the '`*`'). e.g. entering "`*LIST:`" on the terminal will send "`LIST:`" to the server, requesting a list of current chatters. "`*KICK:Wilma`" would send "`KICK:Wilma`" and so on. **Your client does not need to check the validity of these command messages**, simply send them as-is.
- all other lines entered on `stdin` are to be converted to "`SAY:`" messages according to the format in Table 3. This means in an interactive client session you can just type messages and the client will convert them into the right format for transmission.

Client message handling

The client shall accept and respond to the following messages from the server:

- `AUTH:` – authentication challenge from the server. Respond with `AUTH:`*`authstring`*
- `WHO:` – naming request from the server. Respond with `NAME:`*`name`*
- `NAME_TAKEN:` – the requested name is already in use. The server will re-issue the `WHO:` challenge.
- `OK:` – the client's `AUTH:` or `NAME:` response has been accepted.
- `ENTER:`*`name`* – a client called *name* has entered the chat. Emit "`(`*`name`*` has entered the chat)`" to `stdout`.
- `LEAVE:`*`name`* – a client called *name* has left the chat. Emit "`(`*`name`*` has left the chat)`" to `stdout`.
- ~~`SAY:MSG:`~~*`name`*`:`*`message`* – a client called *name* said *message*. Emit the message to `stdout` in the format "*`name`*`:` *`message`*" (note the single space after the colon). Updated in Spec revision 1.1
- `KICK:` – the client is being kicked and must close the connection immediately and terminate with the appropriate error message to `stderr` and exit code (see Table 2).

Error handling

The client does not need to gracefully handle the unexpected disappearance of the server, we will not test for this. The client shall silently ignore any bad messages it receives.

Name negotiation

Clients first attempt to connect and identify using the name passed on the command line. Similarly to Assignment 3, if a `NAME_TAKEN:` response is received from the server, then the client shall retry by appending an integer, starting from zero, to their name and use this in their response to the next `WHO:` message received from the server. Each client will repeat this with an increasing integer until the server is satisfied that their name is unique (e.g. `fred`, `fred0`, `fred1`, ...)

A client shall terminate in the following conditions. The exit codes and required output for these different cases are specified in Table 2:

- After sending a `LEAVE:` message (this is a normal exit).
- Upon reaching end of file on `stdin`, e.g. if `client` has its `stdin` redirected from a file or a pipe (which is how we will be testing). Do not send a `LEAVE:` message, just terminate immediately (this is a normal exit).
- Command line error (bad arguments) or a missing auth file.
- Authentication failure.
- Upon receiving a `KICK:` message. Do not send a `LEAVE:` message, just terminate immediately (this is a kicked exit).

Implementation and testing notes

As per the lectures, we recommend that you have **one server thread for accepting connections**, and then **spawn a new thread to handle each client**. There might be other ways of doing it, but they will be more difficult and probably involve `select()` or `poll()`, both of which are forbidden from use in the server.

Your client will probably require two threads – one for listening for messages from the server and handling/displaying them, and another for handling `stdin` and sending messages/commands to the server. Again, there may be other ways of doing it but they will be more difficult or lead to your client blocking on `stdin` and preventing the receipt and processing of messages from the server. Please don't be tempted to play with O_NONBLOCK, you don't need it and will just make things much more difficult than they need to be.

Spec v1.1 update: You may use `select()` in your client if you wish, however it is not compulsory and there are no additional marks for doing so. You client must still behave correctly handling simultaneous network and console I/O. **Do not use `select()` with your server.**

Consider a dedicated signal handling server thread for `SIGHUP`. `pthread_sigmask()` can be used to mask signal delivery to threads, and `sigwait()` can be used in a thread to block until a signal is received. You will need to do some research and experimentation to get this working.

Due to the nature of this task there is potentially non-determinism in the precise order of message receipt. We will compensate for this in testing and through testing specific features in isolation, however dropped messages etc will be detected and penalised appropriately.

You should expect some stress testing as well - many clients connecting to a server simultaneously and sending lots of messages.

Error messages and exit codes

Server

Exit	Condition	Message (to `stderr`)
0	Normal exit	
1	Incorrect number of args or unable to open authfile	`Usage: server authfile [port]`
2	Communication error (unable to create socket etc)	`Communications error`

Table 1: Server errors, messages and exit codes

Client

Exit	Condition	Message (to `stderr`)
0	Normal exit	
1	Incorrect number of args or unable to open authfile	`Usage: client name authfile port`
2	Communications error (unable to open socket, lost connection with server (added in rev 1.1))	`Communications error`
3	Client was kicked	`Kicked`
4	Client failed authentication	`Authentication error`

Table 2: Client errors, messages and exit codes

Message protocol details

Messages between the clients and the server take the the form

`CMD:arg1:arg2:...\n`

where `CMD` is an uppercase command word, and `arg1` .. `argN` are arbitrary text parameters to the command. These arguments, and the command word, will not contain a colon character. Different commands have different numbers of arguments – see the later tables for details. Some commands have a different format depending on whether they are from client to server or vice versa.

Messages are sent in plain text over socket connections established between the clients and the server. Each message is terminated by a single newline (\n).

The following table describes the messages sent between the server and clients. Note that broadcast messages (i.e. messages to all) are sent to all clients participating in the chat session.

Direction	Format	Detail
server → client	`WHO:`	Name request from the server. Receiving client should reply with its name (`NAME:` message).
server → client	`NAME_TAKEN:`	Somebody is already using that name (server will follow up with another `WHO:` message).
server → client	`AUTH:`	Authentication challenge – receiving client should reply with the auth string (`AUTH:` message).
server → client	`OK:`	Server acknowledgement of a good `NAME:` or `AUTH:` response).
server → client	`KICK:`	The receiving client is being kicked from the chat – they must terminate immediately with the appropriate error code.
server → client	`LIST:`*`name1,...,nameN`*	Response to a client `LIST:` command, comma seperated list of current chatters in lexicographical order. No trailing comma e.g. `LIST:barney,fred,wilma`
server → all (Updated in Spec revision 1.1)	~~`SAY:`~~ `MSG:`*`name`*`:`*`text`*	Server is broadcasting that the client called *`name`* sent message *`text`* e.g. ~~`SAY:`~~ `MSG:fred:Hello world!`
server → all	`ENTER:`*`name`*	The named client has entered the chat e.g. `ENTER:Fred`
server → all	`LEAVE:`*`name`*	The named client has left the chat e.g. `LEAVE:Fred`
client → server	`NAME:`*`name`*	Client informing the server of their name e.g. `NAME:Fred`
client → server	`AUTH:`*`authstring`*	Authentication response to an `AUTH:` challenge from server
client → server	`SAY:`*`msg`*	Client sends a chat message e.g. `SAY:Can I get an F in the chat?`
client → server	`KICK:`*`name`*	Ask the server to kick the named client out of the chat. The server will generate a `KICK` message to the relevant client, if they are in the chat.
client → server	`LIST:`	Request a list of current chat participants from the server. Server will respond with a populated `LIST:` message.
client → server	`LEAVE:`	Client is informing the server that it's leaving the chat.

Table 3: Client / server communication protocol

Notes on special characters, message and name lengths (new in Spec rev 1.1)

The following restrictions are to be implemented server side - that is the server must detect and handle these situations. No checking is required on the client side:

- in general, names and messages can contain spaces, and be of arbitrary length
- messages may contain colon characters, however we will test this as a special case so if you aren't able to implement it then it won't cause you to fail a large number of tests
- the server shall convert any non-printable characters (character value lower than 32) in names and messages to question marks ('?') before re-broadcast.

Other relevant notes:

- Names cannot contain colon characters due to the message protocol. This case will not be tested.
- As noted earlier, names provided by clients must be at least one character long.

Style

You must follow version 2.0.4 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

Hints

1. Consider starting with `client`.
2. You can test and develop client and server programs in isolation using netcat, e.g.
   - fake server: `nc -l port` / `./client name auth port`
   - fake client: `./server auth port` / `nc localhost port`
3. We will make demo client and server binaries available on moss to test against.
4. Be careful to flush output streams.
5. Consider creating data structures and APIs for common functionality shared between client and server – in particular message formatting and parsing.
6. When developing the server, start simple to get the data flow right, before worrying about the meaning and behavior of specific messages. The simplest server would basically be a broadcast relay – accept connections and copy anything received on any connection out to all other connections.
7. Be careful to only use thread-safe versions of C library functions – `strtok()` vs `strtok_r()` for example. `sscanf()` might be simpler, and it's thread safe.
8. You'll need something like a linked list to manage the client list. Make sure it's thread-safe with appropriate locking. If you manage the list insertions well then you won't have to worry about sorting the `SIGHUP` statistics and `LIST:` responses, it will just work naturally.
9. We will test your clients and servers against reference implementations, however we will also test them against other test programs which conform to the communication protocol but behave differently. Don't hard-code any assumptions of client behaviour into your server – you must implement the protocol.

Forbidden functions and statements

You must not use any of the following C functions/statements. If you do so, **we reserve the right to remove the offending statements or function calls from your code before testing**, which will likely result in very few or zero functionality marks.

- `goto`
- `longjmp()` and equivalent functions
- `system()`
- `popen()`
- `select()`/ `pselect()` forbidden in `server`, permitted in `client`
- `poll()` / `epoll()` etc

Submission

Your submission must include all source and any other required files (in particular you must submit a `Makefile`). Do not submit compiled files (eg `.o`, compiled programs) or test chat scripts/configurations.

Your program must build on `moss.labs.eait.uq.edu.au` with:
`make`

Your programs must be compiled with gcc with at least the following switches:
`-pedantic -Wall --std=gnu99`

Running `make` should build all programs (server and client).

You are not permitted to disable warnings or use pragmas to hide them.

If errors result from the `make` command (e.g. a required executable can not be created) then you will receive 0 marks for functionality relating to that program (see below). Any code without academic merit will be removed from your submission before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality relating to any programs that can't be built).

Your programs must not invoke other programs, or use non-standard headers/libraries.

Your assignment submission must be committed to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sXXXXXXX/trunk/ass4`

where sXXXXXXX is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

The initial structure of the repository has been created for you. Your latest submission will be marked and your submission time will be considered to be the commit time of your latest submission. **If you commit after the assignment deadline then we will mark that latest version and a late penalty will apply, even if you had made a submission (commit) before the deadline.**

You must ensure that all files needed to compile and use your assignment (including a Makefile) are committed and within the `trunk/ass4` directory in your repository (and not within a subdirectory) and not just sitting in your working directory. Do not commit compiled files or binaries. **You are strongly encouraged to check out a clean copy for testing purposes, and use the `reptesta4.sh` test script when available.**

The late submission policy in the CSSE2310/CSSE7231 course profile applies. Be familiar with it.

Marks

Marks will be awarded for functionality and style and documentation.

Functionality (60 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program(s) correctly implement(s), as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories (detail new in spec revision 1.1):

1. `server` (35 marks total)
   - (a) Correctly handle invalid command lines and unopenable files (2 marks)
   - (b) Correctly create and listen on a server socket (3 marks)
   - (c) Correctly handle authentication and name negotiation (5 marks)
   - (d) Correctly handle a conversation with one client (5 marks)
   - (e) Correctly handle a conversation with two or more clients (5 marks)
   - (f) Correctly handle `LIST:` messages (3 marks)
   - (g) Correctly handle `KICK:` messages (2 marks)
   - (h) Correctly handle `LEAVE:` messages and clients disconnecting (5 marks)
   - (i) Correctly handle signals including `SIGHUP`, and statistics reporting (5 marks)

2. `client` (25 marks total)

(a) Correctly handle invalid command lines and unopenable files (2 marks)
(b) Correctly connect to a server socket (2 marks)
(c) Correctly handle `AUTH:`, `WHO:` and `NAME_TAKEN:` server messages (3 marks)
(d) Correctly handle `KICK:` server messages (3 marks)
(e) Correctly handle `LIST:` server messages (3 marks)
(f) Correctly handle `MSG:` server messages (3 marks)
(g) Correctly handle `ENTER:` and `LEAVE:` server messages (2 marks)
(h) Correctly handle (send) chat messages read from standard input (3 marks)
(i) Correctly handle standard input lines starting with '*' (2 marks)
(j) Correctly handle communication errors from the server (2 marks)

Style (10 marks)

Style marks will be calculated as follows: Let

- W be the number of distinct compilation warnings recorded when your code is built (using the correct compiler arguments)
- A be the number of style violations detected by `style.sh` (automatic style violations)
- H be the number of **additional** style violations detected by human markers. Violations will not be counted twice

Your style mark S will be

$$S = 10 - (W + A + H)$$

If $W + A + H \geq 10$ then S will be zero (0) - no negative marks will be awarded. H will not be calculated (i.e. there will be no human style marking) if $W + A \geq 10$

The number of style guide violations refers to the number of violations of version 2.0.4 of the CSSE2310/CSSE7231 C Programming Style Guide.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name).

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final - it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark - this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission, however the marker has ultimate discretion – the tool is only a guide.

SVN commit history assessment (5 marks)

Markers will review your SVN commit history for your assignment from the time of handout up to your submission time.

This element will be graded according to the following principles:

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit

Again, don't overthink this. We understand that you are just getting to know Subversion, and you won't be penalised for a few "test commit" type messages. However, the markers must get a sense from your commit logs that you are practising and developing sound software engineering practices by documenting your changes as you go. In general, tiny changes deserve small comments - larger changes deserve more detailed commentary.

Documentation (10 marks) - for CSSE7231 students only

Please refer to the grading critera available on BlackBoard under "Assessment" for a detailed breakdown of how these submissions will be marked.

CSSE7231 students must submit a PDF document containing a written overview of the architecture and design of your program.
This document should describe, at a general level, the functional decomposition of the program, the key design decisions you made and why you made them.

- Submitted via Blackboard/TurnItIn prior to the due date/time
- Maximum 2 A4 pages in 12 point font
- Diagrams are permitted up to 25% of the page area. The diagram must be discussed in the text, it is not ok to just include a figure without explanatory discussion.

Don't overthink this! The purpose is to demonstrate that you can communicate important design decisions, and write in a meaningful way about your code. To be clear, this document is not a restatement of the program specification - it is a discussion of your design and your code.

If your documentation obviously does not match your code, you will get zero for this component, and will be asked to explain why.

Total mark

Let

- F be the functionality mark for your assignment.
- S be the style mark for your assignment.
- V be the SVN commit history mark.
- D be the documentation mark for your assignment (for CSSE7231 students).

Your total mark for the assignment will be:

$$M = F + \min\{F, S\} + \min\{F, V\} + \min\{F, D\}$$

out of 75 (for CSSE2310 students) or 85 (for CSSE7231 students).
In other words, you can't get more marks for style or SVN history or documentation than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to `csse2310@helpdesk.eait.uq.edu.au`.

Version 1.1 - 14 May 2021

1. Added `MSG:` message type used for broadcasting from server to clients. `SAY:` remains used for client → server messages. This simplifies things because previously there were originally two variants of the `SAY:` command, with either one or two additional arguments.
2. Add note re: permitted use of `select()` for **client only**.
3. Minor wording changes to description of server behaviour – no change to expected behaviour.
4. Clarify that the server must clean up data structures etc. after unexpected client disconnection, just as it does after receiving a `LEAVE:` message

5. Added section that clarifies handling of spaces, colons and non-printable characters, and minimum and maximum lengths for names and messages.
6. Added clarification that names provided by clients must be at least one character long, and that zero-length names from clients shall receive a `NAME_TAKEN` response.
7. Specify client behaviour in the event of server terminating (Comms error)
8. Updated marking scheme.