The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310/CSSE7231 — Semester 1, 2021
Assignment 3 (v1.1 14/4/2021)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)
Weighting: 15%
Due: 3:59pm 7 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 Unix multiprocessing and inter-process communication using processes and pipes, and to further strengthen your general C programming capabilities through more advanced program design and implementation.

You are to create a set of programs, one server (`server`) and two clients (`client`, `clientbot`), which together form a simple chat messaging system. The server is responsible for running the clients as specified in a chat configuration file, and establishing pairs of pipes such that each client can communicate bidrectionally with the server. The clients and server will communicate using a simple text-based messaging protocol. `client` is a simple client that takes its commands and chat messages from a text file, while `clientbot` has slightly more advanced functionality, and listens to the chat stream and makes "canned" responses based on messages sent by other users.

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` *`configfile`*

where *`configfile`* is the name of a text file that describes the clients participating in the chat session. The format of the config file is as follows:

- Lines where the first non-whitespace character is a `#` are to be ignored as comments
- All other lines are of the form
  `program``:`*`arg`*
  where *`program`* is the path to the program to be run as the chat client, and *`arg`* is a single command line argument that will be passed to *`program`* when it is executed. These two fields are separated by a colon '`:`'.
- A colon may not appear in either the program or the argument, however other characters such as directory slashes, spaces etc are valid.
- Neither the program nor the argument may be the empty string, we will not test this case and you do not have to handle it.
- All invalid lines (such as no colon at all) should be discarded, and your program should silently move on to the next line.

The server will attempt to create a chat session with one participant per line in the config file. The following shows a sample config file.

```
# Chatter 1 is ./client, started with chat1.txt as its argument
./client:chat1.txt
./clientbot:botconf.txt
# Another comment, ignored
./client:chat2.txt
```

Listing 1: Sample server config file

This config would specify a chat session with three participants, the first running `./client`, the second `./clientbot`, and the third `./client`. Note that these may include include leading directory specifiers (e.g. `./`, `../`, or absolute paths). These should be passed directly to `exec()` when attempting to start the clients - you are not required to test for the validity or existence of these files prior to attempting to execute them. Note that in this chat session, the same program `client` is run twice, but with a different command line argument.

General notes on required server functionality:

- Your server will `fork()` and `exec()` a new process/program for each client, according to the chat config
- Your server must establish communications with each child using pipes and the `pipe()` system call. Note that communications are required in both directions, so two pipes will be required for each client.
- Your server is to support an arbitrary number of clients (up to system limits of maximum processes, file handles etc). A session with a single client is also valid, if somewhat lonely.
- If the specified chat config file is missing, or the server command line is malformed, the server shall emit an error message to `stderr` and exit with the appropriate exit code specified in Table 1.
- The second argument on each line in the server config is passed unmodified as the command line to the respective client upon startup.
- The server will need to track the expected 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 should learn the names of all clients before sending any "YT:"s
- After a server sends a client YT: they should only read from that client until it receives "DONE:" or "QUIT:" back from that client

- The server must suppress any client output to `stderr` (see the Hints section at the end of this document).
- As clients quit or are kicked, the server must track this and no longer attempt to communicate with terminated clients.
- If a poorly formatted message or eof is received from a client, treat it as if they quit
- If all clients have left the chat, the server should terminate with exit code 0.
- The server should expect no input on `stdin`
- The server should emit to `stdout`
  - a log of clients entering the chat (a `NAME:` message was accepted and verified unique)
    `(`*`name`*` has entered the chat)`
  - all messages sent by all clients (`CHAT:`
    `(`*`name`*`) `*`message contents`* (note space between `(`*`name`*`)` and the message text
  - a log of clients leaving the chat
    `(`*`name`*` has left the chat)`
- The server should emit nothing to `stderr` except as specified in Table 1.

The following shows an example `stdout` captured from a server session.

```
(client has entered the chat)
(client0 has entered the chat)
(clientbot has entered the chat)
(client) Hey everyone I'm fred
(client0) Hey where did all the dinosaurs go?
(client0) I miss them!
(client) Asteroid got'em - boom!
(client0) Can I get a coffee?
(clientbot) Yes please, make mine a double shot
(client has left the chat)
(client0) I'm still here
...
```

Listing 2: Sample server `stdout` output

Clients

This section first describes the general behaviour of client programs, before further defining the specific behaviour of each of `client` and `clientbot`.

In general, clients are programs that receive messages over `stdin`, and send their responses over `stdout`. Clients also generate output to `stderr`. This `stderr` output will be ignored by the server, however client `stderr` output will be used to test the functionality of your clients as standalone programs.

Clients must be able to respond to a specific set of messages from the server (see Table 3) for message format specifics and examples):

- `WHO:` - a query from the server asking the client to identify itself. Client respondes with `NAME:` message
- `NAME_TAKEN:` - server is indicating that name is already taken, and will ask again (with `WHO:` next time around.
- `YT:` - "Your turn" - the client is now free to send `CHAT:` messages to say things into the chat
- `MSG:` - the named client sent a message to the chat. Note that clients will receive their own messages back via this channel.
- `KICK:` - the client has been kicked from the server. It should immediately terminate with the appropriate error message.
- `LEFT:` - the named client has left the chat

Clients are permitted to send the following messages (at the appropriate time):

- `NAME:` - tell the server the name we want to use (only in response to a `WHO:` message)

- `CHAT:` - say something into the chat (only after receiving `YT:`)
- `DONE:` - inform the server that we are done talking for now, let somebody else have a turn (only after receiving `YT:`)
- `KICK:` - request to kick the specified client
- `QUIT:` - tell the server we are leaving

When the client receives the `MSG:`*`name`*`:`*`message`* command from the server it shall output the message to `stderr` as follows (note space between the sender name and their message):

`(`*`sender`*`)` *`message`*

When the client receives the `LEFT:` command from the server the client will emit to its `stderr` the following `(`*`name`* `has left the chat)`

Below is an example of the `stderr` output from a typical client interaction. Note that it's format and contents are identical to the server's `stdout`, except that no messages are printed when clients enter the chat.

```
(client) Hey everyone I'm fred
(client0) Hey where did all the dinosaurs go?
(client0) I miss them!
(client) Asteroid got'em - boom!
(client0) Can I get a coffee?
(clientbot) Yes please, make mine a double shot
(client has left the chat)
...
```

Listing 3: Sample client `stderr` output

Each client has a hard-coded name – `client` and `clientbot` as appropriate (see below). However, the server requires each name be unique, and will reply with a `NAME_TAKEN:` message if somebody is already using that name. In that case, clients are to append 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. `client`, `client0`, `client1`, ...)

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 `QUIT:` message (this is a normal exit).
- Upon reaching the end of its chat script. Do not send a `QUIT:` message, just terminate immediately (this is a normal exit).
- Upon receiving a `KICK:` message. Do not send a `QUIT:` message, just terminate immediately (this is a kicked exit).
- If it receives an invalid command from the server, or any other sort of error such as end of file on `stdin` (these are considered communications errors)

`client`

This is the simplest client. It is launched with the following command line:

`./client` *`chatscript`* where *`chatscript`* is a simple text file that contains the messages that the client will send, and other simple actions the client can take.

If the client is run with an invalid number of arguments, or the specified chat script file does not exist, `client` should exit with the appropriate error message to `stderr` and exit code as specified in Table 2.

After the initial `WHO:/NAME:` negotiation, the client can simply emit the commands that are specified in the chat script.

An example chat script is shown below:

```
CHAT:Hello everybody!
DONE:
CHAT:How are you doing today?
CHAT:I'm a pretty stupid client, but that's ok
```

```
DONE:
CHAT:I'm still talking, how about that?
DONE:
QUIT:
```

Listing 4: Sample client chat script file

Note the following about the chat script format:

- Messages to be sent to other chat clients start with `CHAT:`
- Sets of messages are terminated with `DONE:`
- The `QUIT:` command informs the server that this client is leaving the chat.
- The colon (‘`:`’) character must not appear in the body of any messages. We will not test this case and you don’t have to handle it.
- The `KICK:`*`name`* command informs the server to kick the client with that name
- No message or name may be the empty string e.g. ”`CHAT:`” We will not test this case and you don’t have to handle it.
- All invalid lines should be discarded, and your program should silently move on to the next line.
- You do not need to handle the situation that a client kicks themselves, we will not test it.

`clientbot`

The `clientbot` uses the same communication protocol and server interactions as the regular `client`, however instead of responding from a pre-written chat script, it listens for messages (stimulus) from other clients and auto-replies a response upon receiving certain messages.

`clientbot` is launched as follows:

`./clientbot` *`responsefile`*

where *`responsefile`* is a text file that contains the list of stimulus and response pairs, each seperated by a colon character. Similar to other configuration files in this assignment, lines in which the first non-whitespace character is a `#` are to be treated as comments.

Lines of the form

`<stimulus>:<response>`

specify the stimulus string to be looked for and the corresponding respond to be made.

Neither the stimulus nor the response may contain the colon character or be an empty string – we will not test for this and you do not have to handle this case.

All invalid lines should be discarded, and your program should silently move on to the next line.

For example, the following config

```
# 2014 called, they want their meme back
F in the chat:F
# allright allright allright allright
What's cooler than being cool?:ICE COLD!
# This message brought to you by Big Coffee
coffee:Yes please, make mine a double shot
```

Listing 5: Sample clientbot configuration file

would cause the `clientbot` to listen to the chat messages from other clients, match each message against the entries in the stimulus file, and if detected reply with the required response.

Stimulus messages are to be matched in a case-insensitive manner – from the previous example listing, a message containing “COFFEE” in uppercase would match. The stimulus string must appear somewhere in the incoming chat message to be matched, but it does not have to match the entire message.

If the provided *`responsefile`* is missing or invalid in any way, the `clientbot` should exit immediately, returning the appropriate error code (see Table 2).

The **`clientbot` must not reply to its own messages**, but it might respond to messages from other `clientbot` instances participating in the chat.

To operate correctly, `clientbot` must somehow remember the messages it’s seen and save them until it next receives `YT:` from the server indicating it’s time to reply. Don’t try to cheat and immediately emit `CHAT:` messages when you get matching stimulus!

Error messages and exit codes

Server

Exit	Condition	Message (to `stderr`)
0	Normal exit	
1	Incorrect number of args or unable to open configfile	`Usage: server configfile`
...		

Table 1: Server errors, messages and exit codes

clients

Exit	Condition	Message (to `stderr`)
0	Normal exit	
1	Incorrect number of args or unable to open chatscript (`client`)	`Usage: client chatscript`
1	Incorrect number of args or unable to open responsefile (`clientbot`)	`Usage: clientbot responsefile`
2	Communications error	`Communications error`
3	Client was kicked	`Kicked`
...		

Table 2: Client errors, messages and exit codes

Client/server interactions

Upon starting, the server will read the chat config to identify the clients who are participating in the chat session. The server communicates with each client in a round-robin fashion, in order of the clients' appearance in the chat config file.

The server begins by sending the `WHO:` message to each client, and listening for a `NAME:` response back.

The server checks the received name against any other named clients. If the name already exists the server should send a `NAME_TAKEN:` message to that client, then move onto the next client. Next time around it will re-send the `WHO:` message to clients that still haven't completed name negotiation. Note that this implies it is possible for a client to never complete name negotation, but other succesfully named clients can still be chatting.

```
(server)WHO:
(client)NAME:name
(server)YT:
(client)CHAT:
(may repeat)
(client):DONE:
...
(client)QUIT:
```

Listing 6: Sample communications

Message protocol

Messages between the clients and the server take the the form

`CMD:arg1:arg2:...`

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 number of arguments - see the later tables for details.

Messages are sent in text over pipes established by the server.

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:`	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	`YT:`	Receiving client is free to send messages
server → client	`KICK:`	The receiving client is being kicked from the chat – they must terminate immediately with the appropriate error code
server → all	`MSG:`*`name`*`:`*`text`*	Server is broadcasting that the client called *`name`* sent message *`text`* e.g. `MSG:fred:Hello world!`
server → all	`LEFT:`*`name`*	The named client has left the chat e.g. `LEFT:client`
client → server	`NAME:`*`name`*	Client informing the server of their name e.g. `NAME:Fred`
client → server	`CHAT:`*`msg`*	Client sends a chat message e.g. `CHAT: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	`DONE:`	Client is informing the server that it's done sending messages for now. The server will follow up with another `YT:` message when it's this client's turn again.
client → server	`QUIT:`	Client is informing the server that it's leaving the chat.

Table 3: Client / server communication protocol

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 your client programs standalone – just interact with them over `stdin/stdout` as though you are the server.
3. Be careful to flush output streams, don't make any assumptions about the default flushing behaviour of `pipe()`-created file handles.
4. Do some experiments to understand how to establish pipes in both directions between a parent and its `fork()`/ `exec()` child. Make sure you know how to do this before starting on the server.
5. Think about the various states a client can be in as it progresses through a chat session, and the messages/responses it is expecting in each state. Getting this right will greatly simplify your client and server programs.
6. When debugging client/server interactions, the `tee` command may be useful. You can wrap your client in a shell script that uses `tee` to pass through and log `stdin/stdout`, and launch the wrapper script from the server (in the chat config). This will allow you to log the interactions between the two.
7. There is a lot of common functionality between the two client programs – splitting that out into supporting `.c` and `.h` files will greatly reduce the complexity of your clients. You might also find some common functionality between the clients and the server program.

8. You may wish to consider the use of the standard library function `isspace()` when parsing configuration files.

9. The standard library `strtok()` function may be useful for extracting different parts of received messages (colon separator). Be aware that `strtok()` is not thread-safe, which is not a problem for this assignment but look into `strtok_r()` if you want to get ready for Assignment 4 (multi threading). Alternatively, `strchr()` may be a simpler alternative.

10. We will test your servers against reference implementations of `client` and `clientbot`, however we will also test them against other test agents 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.

11. For suppressing client `stderr` output when launched by the server, consider the special file/device `/dev/null`, which silently eats anything written to it. Your server is already redirecting the spawned clients' `stdin` and `stdout` to pipes, could it do something similar with `stderr`?

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()`

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 both clients).

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).strchr

Your programs must not invoke other programs (other than the server `exec()`-ing the clients as per this specification), 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/ass3`

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/ass3` 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 `reptesta3.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 (detailed subcategory marking to be advised in a future update):

1. `server` (25 marks total)
   - (a) Correctly handle invalid command lines and unopenable files (3 marks)
   - (b) Correctly identify and reject invalid chat configurations (2 marks)
   - (c) Correctly handle clients not launching or sending bad messages (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 a conversation with your own clients (5 marks)
2. `client` (25 marks total)
   - (a) Correctly handle invalid command lines and unopenable files (3 marks)
   - (b) Correctly handle ”WHO:”, ”NAME_TAKEN”, ”KICK:”, and ”MSG:” server messages (7 marks)
   - (c) Correctly respond to all server messages using chatscripts (10 marks)
   - (d) Correctly identify and reject malformed chatscript lines (2 marks)
   - (e) Correctly handle communication errors from the server (3 marks)
3. `clientbot` (10 marks total)
   - (a) Correctly handle invalid command lines and unopenable files (1 mark)
   - (b) Correctly identify and reject malformed response file lines (2 marks)
   - (c) Correctly respond to all server messages using response files (7 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 April 2021

1. Added hint re: client `stderr` suppression
2. Clarify handling and corner cases in server config file
3. Clarify behaviour of `client` regarding `KICK:` message reception, empty names/messages and ignoring invalid lines in chat scripts
4. Clarify behaviour of `clientbot` such that any invalid lines in the stimulus file should be ignored
5. Clarify that clients do not need to be able to `KICK:` themselves, and this will not be tested