CITS3002 Computer Networks Report

Kieren Underwood - 21315543, Daniel Maitland - 21986102

1. **How would you scale up your solution to handle many more clients?**

The design of our game makes it easy to handle an arbitrary number of clients. The server is forced to choose, as a command line argument, the number of MAX_PLAYERS. In "game.h" we #define the MAX_LIVES value.

These two variables can be changed by anyone implementing the game in order to handle any number of lives and anywhere from 2-900 players. We also allow a single player version of the game to be played, where the player will play rounds until they lose all their lives and are eliminated. The upper bound of 900 comes from the need for a 3-digit ID, which we have decided to implement as an int from 100-999. These are mapped onto specific client_fd's.

The code's logic allows us to scale up to any number of clients (within the previously specified bounds). It runs, essentially, as follows:

- On start-up, we allocate a certain amount of shared memory, based on MAX_PLAYERS and the size needed for our *playerInfo* struct. This memory can be accessed by parent and child processes.
- We set up a lobby, and wait for the specified number of players--MAX_PLAYERS--to join. If we do not receive an INIT message from MAX_PLAYERS number of clients, we do not begin a game.
- We enter a playGame() function, and then fork() for each player. Our parent process waits for all of these child processes to enter into a playRound() function, which alters our *playerInfo* struct depending on whether they guessed correctly or incorrectly. Once all the child processes are finished, we update *numLives* for each player, and check whether a winner has been found.

The big restriction in our implementation is that one cannot choose to have a range of players: ie. one cannot choose to have a "4-8 player game." You can have 4 players in a game, or 8 players in another, but a range is not possible. You must decide a specific number of players first, and then make sure they successfully initiate a game.

2) **How could you deal with identical messages arriving simultaneously on the same socket (or on different sockets)?**

We will deal with each situation separately.

Situation 1: Two identical messages arrive simultaneously from different sockets.
In this case, one or both of the clients are sending the wrong ID number(s). This behaviour is defined by our implementation as *cheating*, and the cheating client(s) will be eliminated from the game. Having the messages arrive simultaneously is not a problem, as the process has already forked() by this time, allowing messages to be dealt with separately.

Situation 2: Two identical messages arrive simultaneously from the same socket.
Currently, after trying to test this situation with a python client, our server will not receive two messages simultaneously. It reads one, and then moves on to process the move. If it were to buffer the two messages inside one packet, then it would be processed as a syntax error (packetLength > 14) and the player on that socket would lose a life.

3) **With reference to your project, what are some of the key differences between designing network programs, and other programs you have developed?**

[Both of us have only programmed for little over a year. The programs we have built so far have relied on a set of inputs which are assumed to be of a certain format, with little need for anything but the most basic input checking--if the input is wrong, it is the users fault and they must re-input something more intelligible.]

- Error checking

The most fundamental difference in developing this client-server program was the level of necessary error checking. In contrast to previous projects, almost every stage of the transfer of the data is checked. (This may have been bad coding practise on my behalf in the past.) We check for errors when, among other places, we are:

- Setting up a server socket
- Listening for clients
- Accepting clients
- Reading client messages
- Sending client messages
- Checking client inputs
- Forking()
- Etc.

When checking client inputs in our *parse_message()* function, we stop to check for errors in the client's input four times, each time sending a default ERR flag back to our process if any syntax error is found in the packet.

The reason for this comparatively excessive checking comes from the next major difference--non-deterministic input.

- Non-deterministic input

So far, the programs I have built have dealt with what we could describe as "deterministic" inputs--files with text, integers, strings to sort and store, etc. The input has been fixed: once we start the program, we are dealing with the computer and data that will behave deterministically (in the strict sense).

When we deal with clients (putting my other philosophical objections aside), we are dealing with essentially non-deterministic input. Perhaps "*player101*" has been up all night doing assignments, and when he sits down to play our game, the numerous coffees he's drank make his hands jittery: he types "101,MOB,EVEN" (an invalid move) instead of "101,MOV,EVEN." A computer wouldn't do that, but a human would. Everything that is input, we must check. We also have to check whether the input even arrives--perhaps the coffees our *player101* had finally wear off, and he falls asleep at the desk.

The basic philosophy, then, is to set up our program to reject everything except perfection. We are dealing with non-deterministic input, so we must accept only what we determine. If it falls outside our bounds, we have to catch these errors, and send them off to other functions which will penalize the input.

- Timeouts

Other programs I have dealt with have had no need for timeout checks. The majority of the running time of the client-server program dealing with humans will be taken up by the decision making of the human. So we have to make sure this doesn't get out of hand.

- Forking()

I had only dealt with the system call fork() briefly in CITS2002: Systems Programming. But the nature of dealing with multiple clients at once forced us to use fork() extensively. How do you both *wait* for an INIT message from a client, while *listening* for other clients wanting to connect? You do it with two instances of your program--it is not obvious to me how this can possibly be done with just one instance.

Computers seems to be doing things simultaneously, but are actually carefully switching between concurrent processes. In other programs I have built, the input passed through a series of linear functions, and did not require other processes to be running at the same time.

- Shared memory

When forking(), we run into the problem that while the parent and child process can communicate with each other in a limited sense, they are generally isolated copies. They run with separate copies of memory. This means if a child copy of our program wants to decrement the number of lives *player101* has--this has no effect on the parent copy. After an incorrect guess *player101::child* has n-1 lives, while *player101::parent* still has n lives.

This is a problem I have never encountered with other programs I have built. Memory is generally accessible between all functions, if you only give it the appropriate scope, or pass it directly to your function. This problem was solved using shared memory--specifically *mmap()*. We allocate memory that is designated as shared between parent and child processes before our game has begun. When these now universal (as opposed to global) variables are changed by the parent or by a child, everyone gets to see and access the results.

4) What are the limitations of your current implementation (e.g. scale, performance, complexity)?

Assumptions:

- See README file for gameplay assumptions.

Scale:

- The program only runs a single game, and then closes all clients and servers. We are not Facebook or Twitter, with endless scrolls to suck in our players with dopamine hits. You play one game, and you are forced to quit. This is the implementation we had in mind when we started, and we decided not to change it when the possibility of recurrent games became available.
- As previously mentioned, the number of players is limited from 2-900.
- The timeout length is the same for sending a message *and* entering the lobby. This could mean that: 1) the lobby is not open for long enough if MAX_PLAYERS is large, and many players must join, or 2) the lobby will be open for long enough, but the timeout length for a player sending a message will be excessively long.

Performance/Inefficiencies:

- We have not tested the game with a very large number of players, eg. ~900. It is possible that the number of forks() called could cause a problems with the number of active processes.

Complexity:

- When a corrupt message from a client is passed to the server, no attempt at correcting it is made. Any mistake is treated as fatal, and the player loses a life. An upgraded version of the game could send back an error message to the client, asking to re-send the packet with the correct syntax.
- If a client drops out after sending an INIT, but before the game starts, the game continues as if that player had dropped out on the first round.