THE UNIVERSITY OF WESTERN AUSTRALIA

***Fraser Loneragan** 22243455*
***Clayton Herbst** 22245091*

Networks Project Report

Contributions from team members include Fraser developing the majority of the PASS or FAIL game logic and assisted in the development of most of the client-side program testing environment. Clayton was involved in building the majority of the I/O functionality of the server and thus managing client-server communications. Both Fraser and Clayton participated equally in the writing of this report.

How we would scale our solution

Currently we have implemented a two-tier client-server model within our solution but in order to scale up the number of players we could use a three-tier model instead. To do this, clients would need to be capable of connecting to an available server, this server should be solely responsible for handling client TCP connections and managing TCP communications, and the 3rd tier/server should manage all players associated data such as their number of lives, dice roll values etc in a database. This will allow for more than the max size of the listen system calls backlogs to connect to the middle-tier server allowing us to scale up our solution further.

Another way to scale up our solution would be to incorporate threading. This can speed up performance through parallelism (GeeksforGeeks n.d). This way our solution would be able to handle more clients and reduce the overall performance slowdowns experienced. We could also scale vertically by adding more ram and a better processor to our computer which hosts the server. A higher clock speed would mean that a single process can process socket I/O faster and hence allow for more efficient TCP communication. Furthermore, we could also scale horizontally. This would mean using multiple computers with their own processors and ram on board, such that a different server could run on each of the computers. The player base will be split amongst these servers and hence allows for an increased number of players in the game play.

Managing identical messages arriving simultaneously on the same socket

One way to deal with identical messages arriving simultaneously on the same socket is to read the first message that comes through the socket, and then ensuring the message buffer is cleared afterwards. That way only one message is read and the other, identical messages are flushed from the receiving buffer. However, if this fails, for example if the sender has threaded it to run simultaneously and it didn’t get queued at the Internet Protocol layer, we could handle the dilemma in the application layer by checking the length of the received input and send an error back to the client asking for resubmission.

Key differences between designing network programs and other programs we have developed?

One major difference is the fact that all ‘data transfer’ is performed through sockets and TCP connections. Clients are only aware of their status when the server informs them. Previously clients would been informed through shared memory within a single program or shared group id. Network programs also need to have a clear, established/agreed method of sharing this data when communicating over its connections. In this battle royal program, a number of specific packets were defined to communicate game status information such that both the client and server can reliably interpret the incoming byte stream. Another difference is the proportion of error checking to actual logic code. In this project we need to deal with many situations which arise from clients e.g. not enough people joining, cheating, incorrect messages. As a result, a large portion of our code is checking for these situations and handling them appropriately, rather than just straight logic like for example which players loses a life and which don’t.

Limitations of our current implementation

Currently we use a queue to receive, process and notify clients. Efficiency is increased as more players are in the game. When there is a low number of players, the simple comparison of “*is the player alive”* would be quicker than our current implementation which is managing a queue abstract data type structure and enqueuing/dequeuing players. However, when there is a large number of players, efficiency is gained because we do not need to iterate through all players, only active players (players who are still alive). The implementation of a queue can also be considered to be more closely aligned to the grammar associated with the battle royal game in the sense that clients are in a ‘lobby’ and wait to receive the outcome of their move; like a queue.

Currently the server socket has a connection buffer set to 128 players as defined in the Linux man page for the listen system call. This thus limits us to potentially 128 players. However, if we tried to run this number of players on our MacBook we will be bottlenecked by the MacBook’s hardware.

Another limitation is the low complexity of the game itself. There are only 4 legitimate moves or messages that a client can choose from and send, but in a “real” battle royal there are many different inputs which can be sent (keyboard and mouse controls) all of which are sent many times a second. In this sense asynchronous capabilities would need to be implemented and clearer packet header information defined through the msghdr structure (defined in sys/types.h). So, the server must be able to handle the I/O very quickly (and relay information back to the client) as the next set of inputs is set to come in the next fraction of a second.

References

GeeksforGeeks, n.d, *Multithreading in C*. Available from: https://www.geeksforgeeks.org/multithreading-c-2/. [22 May 2019].

GeeksforGeeks, n.d, *DBMS Architecture 2-Level, 3-Level*. Available from: https://www.geeksforgeeks.org/dbms-architecture-2-level-3-level/. [20 May 2019].

Hong, J 2019, *Client Server Design*, lecture notes distributed in Computer Networks CITS3002. Available from: http://www.lms.uwa.edu.au. [15 May 2019].