CITS3002 Computer Networks
Project Report

Bruce How (22242664)

May 20, 2019

1 Scalability

The main drawback is that even well-provisioned and stable servers can only maintain a limited amount of players. Implementing an additional server should see an improvement to its scalability and handle many more clients. By multi-threading the server, the program can house more than 1 game at a time. This will allow more clients to play overall.

2 Handling Issues

Part of the tier 4 project implementation is to ensure that the server can identify cheaters. When a new client connects to the server, the client is assigned a unique client ID which is stored in memory on the server. If two or more clients were to send the same identical message simultaneously (including client ID), the server is able to crosscheck the message's client ID against its local copy. Because of this, the server is able to identify if a player has spoofed their client ID (cheating), and ultimately verify a message's validity.

In the case where a client sends 2 packets simultaneously (sequentially), the server will store the first packet and hold onto the second. The second packet would only be stored once the game has processed the first packet. The idea behind this is to ensure that packets sent too quickly or unexpectedly (e.g. whilst the round is still in process) are not lost. This, however, opens opportunities for clients to flood the server with packets. To combat this, the server has been conditioned to eliminate players if they send 3 or more unexpected packets (defined by *MAX_PACKET_OVERFLOW* in the project).

3 Network Programs

Designing a network program such as this project requires particular aspects, such as the handling of new connections, and existing client activities, to be constantly monitored. As clients are independent and freely able to send packets at any stage during the game, my program had to be structured in a way that allows it to receive these packets. As *recv* is a blocking function, my program had to use *fork()* to create child processes for each client connection; each child process is responsible for listening to *recv* to ensure that all packets are received. Blocking functions and the use of many child processes are not common occurrences with programs I have worked with as they typically only have one main parent process.

The heart of my program involves the use of *mmap* to create shared memory. This allows child processes to share data, such as a client's packet, to the parent process which handles the gameplay. Memory management in other programs I have developed are managed using *malloc* and *realloc* as the need for memory sharing across processes is not required.

4 Limitations

My current implementation involves the use of a single server to handle the entire game. As a single server can only maintain a limited number of players, my implementation is not very scalable across many players. Additionally, the use of a single server in a Client-Server architecture is not as robust as the Peer-To-Peer architecture. In the event of a server failure, the whole network would go down causing disruptions to all players.

Because my program utilises *fork()* to handle each client's connection, it can only house a limited number of players based on the number of times *fork()* can be called on the server without issues.

My implementation involves the use of a single thread to handle the gameplay. This means that each round requires at best, *O(n)* time, where n is the number of clients. The use of multi-threading could see an improvement to its performance as its time complexity is *θ(n/p)* (*O(1)* over n processes), where p is the number of concurrent threads on the processor(s) the server is running on.