

CMP303 - Networking

Philip Gooch - 1703221



Network Architecture

- Considerations
  - Needed to handle multiple clients connecting and disconnecting in real time.
  - Aimed to support as many clients as possible.
  - The budget for the game.
  - Handle connections with different quality of internet connections.
- Client / Server
  - Decided to use a client / server model.
  - Server will have the game state that is considered “correct”. All client game states will be based off of this.
  - Any collisions that would happen would be calculated on the server side.
  - Server can decide what to do with poor connections. (disconnect / prediction)
  - Designed to have a dedicated server that all clients will connect to.
  - Less chances of cheating. For example, the server can check the positions that the packets are indicating and run a check to see if they are possible and the packets have not been tampered with.
  - Could have different servers for different geographical areas to reduce latency.
- Peer to peer
  - Easier to cheat. You are in control of what packets other player’s receive.
  - More network lag. Network traffic may have to travel a long distance.
- Hybrid
  - Client acts as a server. If the host disconnects, every other player will disconnect. Could “migrate” the host but would have to freeze the game.
  - Cheaper. No dedicated server. Perhaps a more realistic choice.

Application

- Client reads the IP address of the server from a file that can be edited by the user.
- Different types of packets identified by the first byte in the packet.
- An initial conversation is had between the client and server upon connection via TCP before the game begins.
  - Client attempts to connect.
  - Server accepts connection, generates and sends an ID to the client.
  - Client sends it’s UDP port number to the server.
  - Server and client send empty packets to each other to calculate latency.
  - Server sends it’s current time plus the latency to the client to sync the times.
  - Server generates a snake struct for that client and adds it to a map.
  - Server sends “add snake” packets to the client for each snake.
  - Client generates snakes using this information.
  - Server sends a “finished sending” packet to indicate there are no more snakes.
  - Client knows to stop listening for more “add snake” packets and continues.
  - Server sends an “add snake” packet to every other client for the new client’s snake.

Application

- Client enters the game loop.
- Client tries to receive on both it’s TCP and UDP port every frame.
  - If Client receives a TCP “add snake” packet, it generates a snake from this information and adds it too it’s map.
  - If the Client receives a TCP “remove snake” packet, it removes the snake with the given ID from it’s map.
  - If the Client receives a UDP “update” packet, it stores the information in the snake class’ “lastConfirmedPositions” deque. This is sorted by the time stamp of the packet.
- Client updates it’s local snake using the direction vector from centre of the screen to the mouse. The snake steers towards this direction. The eyes of the snake use the direction vector directly so it is always looking in the direction it wants to go.

Application

- Client updates all the other snakes in a similar way but uses the confirmed positions from the server to predict where the head of the snake will be and interpolates between the last predicted position and this predicted position.
- Client sends an “update” packet to the server every 0.05 seconds. This contains the local head position, direction, a “boosting” Boolean and the ID of the Client’s snake, along with a time stamp.
- Server only keeps track of the snake’s heads positions. (For implementing collisions, it would have to generate the positions for all the snakes tails.)
- Server attempts to receive UDP packets from every client, every frame.
  - If Server receives an “update” packet, it uses the ID to update the associated snake’s information and keeps track of the time stamp.
- Server broadcasts “update” packets for every client, to every client, every 0.05 seconds. (This is expensive and could run into problems when dealing with a large amount of clients. But it handles at least 10 clients in testing.)

Transport Layer

- TCP
  - Used for Important packets.
  - Reliable. Three way handshake. Packets will get there.
    - Connection / Disconnection packets.
    - Add / remove snake packets.
    - Packet giving client it’s ID.
    - Receiving UPD port number.
- UDP
  - Used for less important packets.
    - Position update packets.
  - Sent very frequently.
  - Time sensitive.
  - Doesn’t matter too much if packets are dropped.

Prediction and Interpolation

- Prediction
  - Just using the most recent position the server sends means the other snakes are delayed.
  - This is countered by predicting where the other players should be in real time.
  - Most recent 3 packets stored in order of time stamp.
  - The positions and time stamps in the first two packets are used for linear prediction.
- Interpolation
  - I then interpolate between the last predicted position and the most recent predicted position to a point half way between them.
  - Aiming for a point half way between the two.
  - This smooths the movement of the snake when there are sudden movements from the player.

Network API

- SFML
  - Abstracts low level code like setting up network addresses.
  - Application is in SFML so makes sense to use it’s networking library.
  - SFML Packets handle big / little endianness (order or bits / bytes) across different processor architectures.
  - SFML variable types are guaranteed to be the same size on different machines.

Network Structure

- Using non-blocking input / output for both TCP and UDP sockets.
- Blocking mode for connection on client side as the program should not continue if there is no connection.
- SFML provides a sf::SocketSelector class that has an isReady() function that returns true if the socket passed as an argument will not block. This is used for listening for TCP connection and disconnection packets without stalling the program.

Testing

- Tests were carried out using clumsy, on three computers. The server running on one and two clients running on the two other computers.
- The tails of the snakes are updated locally and not by information send over the network, however they are indicative of where the head has been so it is useful as visual feedback.

Testing

- Latency – client side - UDP
  - 100ms: Slightly jumpy. Not very noticeable. The prediction algorithm has less recent information to work with.
  - 200ms: Jumpy. Visibly Over compensating on sharp movements. Slight delay in movements.
  - 500ms: Very jumpy. massively over predicting movements. When boosting, the head jumps noticeably further than it should. Big delay in movements. Borderline unplayable.
  - In all three tests, when moving predictably, the snakes are nicely synced.
- Latency – server side - UDP
  - 100ms: More exaggerated inaccuracies than when the client was lagging.
  - 200ms: Worse. More twitchy. Still playable.
  - 500ms: Unplayable.
  - Again when moving predictably, it always goes back to its correct position regardless.
  - The server has a higher work load receiving packets from every client and broadcasting to all of them so this is expected behaviour.
- Latency – TCP
  - As latency increases so does the connection time.

Testing

- Packet drop – client side - UDP
  - 10%: Not hugely noticeable. Not important data being sent. Position updates. These are negligible as soon as a newer packet is received.
  - 20%: Occasionally jumpy.
  - 50%: Often jumpy. Prediction not working as well as it will suddenly realise it needs to be massively ahead of where it is.
- Packet drop – server side – UDP
  - 10%: Not hugely noticeable. slightly jumpy.
  - 20%: Again, occasionally more jumpy. more noticeable.
  - 50%: Very noticeably jumpy. More so than on the client side. The packets can be dropped incoming and outgoing so more chance of losing them. Sometimes the head would not be on the screen. Unplayable.
- When the server is broadcasting it sends the most recent update packet from each client, regardless of if it has been updated, so it will be sending duplicates! This is a bug. This would explain the snake head skipping position as the prediction algorithm will be using two confirmed states that have the same positions and time stamps. The velocity will be zero.
- Packet drop - TCP
  - As packet loss chance increases, so does the connection time. TCP will keep sending the packets if it does not get there so will add to connection time.

What I would do differently.

- I would implement the networking functionality before making the local game as it made things more confusing than it had to be.
- I would use the in built SFML packet class to send data as it is safer, although it was good practice sending raw data.
- Try and come up with a more complex method of prediction, using the direction variables as well as position. This could help smooth the snakes even more in bad network conditions.