# Networking

The tutorials on Networking create a very simple working model of a networking game.

It has the following limitations

* The server has no memory or internal storage of where the GameObjects are, but just broadcasts what the clients say to it.
* Nothing happens without the clients instigating it. A client move is broadcast whenever the keys are pressed, but there’s no accounting for objects that move without a client directing them.
* When the client is moving, they spam the network with a message every frame. This would be suboptimal across an internet connection.
* A new client’s object doesn’t show up until the client moves currently.

During this tutorial we’ll address all of these points one by one.

We can do this in the following testable steps.

1. Replace the struct with a GameObject class in its own file and get everything running as before.
2. Make a Read and Write function in the GameObject class to handle all reading and writing. Update the client’s code to sue these, and make sure everything still works.
3. Add a list of GameObjects to the Server’s code. You’ll need to add the .cpp (and .h) from the Client project to the Server project to do this. Update this list when a client sends an update to the Server. Put some console output in the server (eg write out all positions) to test this.

This gives the Server a list of all gameObjects.

1. When a new client is created, broadcast their presence to all other clients. You can test this by creating one client, moving them, and creating a second client. The new client should appear straight away on the first client’s screen.

This covers our final issue from the top list. We can now look at making gameObjects that move on their own, such as bullets that each player can fire. We need to lay a bit of groundwork for this.

1. On the Client, when we press space, we create a new GameObject and tell the server about it, just like we do when we start the client. Get this working and the player should be able to leave behind little pellets that sit there on all clients. Give these bullets high IDs (say start at 1000?) as dished out by the server, and draw all ids < 1000 as smaller than normal.
2. We now want to keep an update thread on the server. Every N milliseconds this will loop through all server objects (ID >= 1000) and update their position, and broadcast them to the clients. Add an initial velocity to the bullets based on the players last move or something, and you should have working bullets now!

Maybe start with a fixed velocity – all bullets fire in the same direction.

1. Put some code in to clean up old bullets when they leave the edge of the map. This gets triggered on the server, and broadcast to all clients to remove the object with that id from their list.

We’ll be seeing a lot of network traffic now that bullets update every frame. We can drop the frequency of the update thread on the server to combat this, and you’ll see the bullets now teleport along a straight line on the clients. Time to introduce velocity on the GameObjects on the clients, and use it to update in between the server updates.

1. On the client, update the server GameObject’s (anything but our object) with their velocity. Snap them to the new position when we get a message from the server. We should now see the bullets moving smoothly again.
2. We can now use this velocity update to reduce the amount of communication between clients. We’re currently sending a message to the server whenever we update a client’s position. We could change this to only update when we change the client’s velocity.

We now want to add the notion of a server position and a client position, ie implement dead reckoning. When we receive an update from the server, we snap the serverPosition instantly, but let the client position (which is displayed) to interpolate towards the server position. This way we get smooth movement, accurate enough behavior and infrequent updates.

We now need to make this system modular, and build a demo game around it.

1. Create a ClientLib static library project and move Client.cpp and GameObject.cpp into it from the main project.
2. Set up the Client Lib so it can compile (include dirs., linker inputs and library paths)
3. Make Client dependent on the CLientLib now that it no longer has GameObject and Client cpp files, and get it to compile and run.
4. Derive a class from Client and use it to handle game specific logic