CMP303/501 Network (Systems for) Game Development

Lab 6: Exchanging Data

Andrei Boiko, Laith Al-Jobouri (and Gaz Robinson)
School of Design and Informatics

Introduction

In this lab, we'll experiment with exchanging messages between clients and servers in a multiplayer network game.

If you have any questions or problems with this week's practical, ask us on-campus or in the MS Teams channel for "Lab Help". If you have a general question about the module, please use the "General" channel or email us (a.boiko@abertay.ac.uk, l.al-jobouri@abertay.ac.uk).

The application

Imagine you're working on a network game with a number of moving objects, each of which has an ID number and a position on the screen. Each object is controlled by one player, but all the players in the game need to know the positions of all of the objects.

This game uses a client-server architecture. Each player runs a client program, which exchanges messages with a single server running on a different machine. The client periodically sends a message to the server giving the details of the object it controls.

When the server receives a message from a client, it redistributes it to all the other clients. When a client receives a message from the server, it updates its local copy of that object's information.

For simplicity, assume that you have a fixed number of players in your game (e.g. 2).

Data structures

We'll use UDP for this application – so please download the UDP client and server from labfiles_03.zip from Week 3 section on MyLearningSpace as a starting point. However, the messages in that example were fixed-length strings of text, and we want to send messages about game objects' positions...

A convenient way of doing this is to define your message's contents as a `struct`:

```
struct Message
{
    int objectID; // the ID number of the game object
    int x, y;     // object position
}
```

This is a plain old data type, as defined by the C++ language standard – it contains only public fields and doesn't use any of C++'s more complex OO facilities (e.g. virtual). This means it's stored in your program simply as a block of memory containing all the variables – which is convenient if we're working with functions that send and receive blocks of memory.

To send a message about a change in the game world, you would declare a variable of this type, fill in its fields as appropriate, and then call `sendto`, giving a pointer to the `Message` variable and using `sizeof` to find out how big the `Message` type is in bytes:

```
Message msg;
msg.objectID = myID;
msg.x = myBike->x;
msg.y = myBike->y;
... = sendto(sock, (const char *) &msg, sizeof(Message), 0, &addr, addrsize);
```

Similarly, you could receive a message directly into a `Message` variable:

```
Message msg;
... = recvfrom(sock, (char *) &msg, sizeof(Message), 0, &addr, addrsize);
```

Add a definition of a `Message` type to your programs. Make sure this is the same on both the client and server – in a real application, you would probably define it in a header file that could be shared between the two programs.

Client to server

We'll set this up in several stages...

Start with an empty main loop in the client. Add a one-second delay to the loop (e.g. using `sleep`).

Add some variables in the client to keep track of the player's position, and make it send a `Message` to the server every time it goes round the loop. You don't need to actually make the player controllable, but it'll be more interesting if you can make its position change over time.

Make the server's main loop wait for a message from any of the clients and print out the contents.

Test this out with multiple clients, and make sure you can see all the position updates in the server's output.

Server to client

Make the server, when it receives a message, send it back to the **same** client immediately. (This isn't very useful for the game, but it'll let us test the next bit.)

Make the client test to see whether it's received any messages each time it goes round its main loop. You can't just call `recvfrom`, because it could block – you will need to use select, as we did in Lab 4. Make the client print out the contents of any messages it receives.

You can now remove the delay, and use a one-second timeout with select instead.

Test this out with multiple clients – you should see each client getting its own message back from the server.

Server to multiple clients

Make the server keep track of which clients have joined the game (e.g. have a `std::vector` of addresses), and make it send any messages it receives to **all** the clients. Make the client keep track of the positions of all of the objects in the game (e.g. have a `std::map` of positions), updating them as it receives messages, and print all the positions out each time it goes round its main loop. If you're putting `sockaddr_in` addresses into standard library collections, you'll find it useful to have definitions of the `==` and `<` operators for them – see **CompareAddresses.cpp** in the Week 6 section on MyLearningSpace.

Test it out – you should see all the clients now exchanging positions.

Portability

There are several portability problems with the simple struct approach we've used that may prevent it working between programs running on different architectures or built with different compilers. To solve these, you would need to:

- Find out how to enable **structure packing** for your message structure, so that it's laid out the same way across different compilers.
- Use **fixed-size types** (e.g. `int16_t`) in your messages, so they're the same size across different architectures.
- Find out how to transform numbers in your messages into network byte order, so they use the same endianness across different architectures.

In practice, a better way to solve these problems is to use a **serialization** approach (e.g. SFML's `sf::Packet` mechanism).

If you've got time left over…

- Fix the portability concerns outlined above.
- To save network bandwidth, make the client only send a message if the client's information has actually changed.
- Packet loss doesn't really matter in this application – unless lots of packets are getting lost. Make the server detect when packets get lost (e.g. by including a sequence number in each message, incremented for each message sent), and complain if more than 50% of packets are missing.