CMP303/501 Network (Systems for) Game Development

Lab 3: UDP Sockets

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

Introduction

In previous labs, we've used TCP, which provides reliable stream communication between two endpoints. In this lab, we're going to experiment with UDP, which provides unreliable datagram communication – it lets you send individual packets of data over the network.

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

Get labfiles_03.zip from Week 3 section on MyLearningSpace, and unpack it. It contains two Visual Studio projects, client and server. Open and compile both projects.

The code is similar to that from Lab 2 – except that it's been changed to use UDP rather than TCP. Each message typed into the client is sent as a UDP message to the server; the server then sends back a UDP message to the client, which prints it out.

Run the programs and check that they work. Look in particular at the IP addresses and port numbers that both programs show – note the difference in port numbers between the two ends. (You might like to refer back to Lectures from weeks 2 and 3 to make sense of this.)

Compare the Lab 2 and Lab 3 code, first for the client and then for the server. Note the differences between them. Which functions aren't used when working with UDP? Which new functions have been introduced?

Packet sizes

As provided, the client sends 40 bytes and the server expects to receive 40 bytes. Try making the client send a bigger message – what happens? What's the biggest message you can send without getting an error code back from `sendto`?

Unreliability

UDP communications are unreliable – packets may be thrown away by the network at any point. In practice, this is quite unlikely to happen on the Abertay local network (and it's basically never going to happen on localhost) – however, if it did, the client and server won't work very well. Look through the code and work out what would happen if a packet was sent but not received.

For testing, an easy way to simulate unreliable communication is to wrap your `sendto` calls with something like `if ((rand() % 100) < 60) { ... }` – this'll make 40% of packets "get lost"! Try this out and see what actually goes wrong.

Plan (don't implement yet) how you would modify the programs to handle this more gracefully. Do you need to change both the client and the server?

An object-oriented API

You're probably a bit fed up with the low-level C-style API provided by WinSock by now.

If you were using sockets in a bigger program, it would be helpful to have a `UDPSocket` class, representing a socket, that wrapped up all the detail of initialising WinSock (the first time it's used), filling in socket address structures, allocating buffers, etc.

The socket programming API provided by the SFML library is a good example of this kind of object-oriented API. Find the SFML documentation for this online and compare it to the "traditional" sockets API we've seen already.

Finally

Do one (or both, if you have time) of these two activities:

- Implement your plan from "Unreliability" above: make the sample programs handle packets getting lost, in either direction.
- Rewrite this week's sample programs using SFML's classes rather than raw sockets. (You'll need to download your own copy of the latest version of the SFML library.)