CMP303/501 Network (Systems for) Game Development

Lab 4: I/O Multiplexing with "`select`"

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

Introduction

In this lab, we'll see how to deal with multiple network connections in a single program through the use of the `select` function. This is one of several possible approaches – we'll discuss some of the others in future lectures and labs.

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 labfiles04.zip from the "Week 4" section on MyLearningSpace and unpack it. It contains two Visual Studio projects, client and server. Open and compile both projects.

This is a TCP client-server system; it works the same as Lab 2's echo server. The client program is identical to Lab 2's client; you shouldn't need to make any changes to it during this lab. The server program, however, has been modified to handle multiple connections at once. Try running the server along with several copies of the client.

I/O multiplexing

In Lab 2, the server could only handle one connection at a time because the `send` and `recv` functions are blocking: they don't return until they've sent or received some data.

This lab's server program has been written in a non-blocking style of programming (as in Lecture 4): it never calls `recv` unless it knows that there is some data to read already. Similarly, it never calls `accept` unless it knows there's a connection ready to be accepted. This means that it can handle multiple connections at once – and that it can do other work in its main loop when it knows there isn't any data to receive.

To find out whether operations on a socket will block, it uses the select function. Look up select on MSDN, find the call to it in the code, and work out what it's doing.

You'll also need to understand:

- the `Connection` class – which represents an open connection to a client;
- how `readBuffer_` and `readCount_` are used in `Connection`;
- how the `conns` list is used to keep track of the connected clients.

If there's anything that's not clear (and there probably will be), discuss it with us in the lab.

Limiting connections

At present, any number of clients can connect to the server.

Modify the server so that it'll only allow a limited number of clients to connect. If there are too many clients already connected, it should close the socket immediately, rather than adding the connection to the list.

Non-blocking send

At present, the server only checks whether read will block – it doesn't check whether send will block. This means that a client that's not receiving data for some reason can prevent any other clients from talking to the server.

select can be used to check for "writability" as well as "readability". Modify the Connection class and the main loop so that they can check for writability too, in the same way as they currently do for readability – i.e. add `wantWrite` and `doWrite` methods. You may choose to add a separate write buffer, or reuse the existing read buffer, but either way you need to keep track of how much data remains to be sent.

Then modify `Connection` so that it keeps track of whether it's currently reading or writing, and switches between the two appropriately – i.e. make it a state machine. This is a common strategy in non-blocking programming.

If you've got time left over…

- Make the server detect idle clients who haven't sent anything in a while and disconnect them.
- Make the server handle variable-length messages, using one of the two strategies from Lab 2.
- Modify Lab 3's UDP client or server to use select. While the server doesn't need to handle multiple sockets, select's timeout feature is useful on its own for UDP – e.g. for detecting when a message has been lost in the client.
- When limiting connections, it'd be nicer to send a "sorry, we're busy" message, rather than just closing the connection. Modify the program to do this (making sure it can't block).