CMP303/501 Network (Systems for) Game Development

# Lab 2: Variable-Length Messages

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

## Introduction

In this lab, we're going to experiment with variable length messages in a network protocol.

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 Lab_2_VariableLength.zip from the Week 2 page on MyLearningSpace and unpack it. It contains two projects, client and server. Compile both projects.

The code is very similar to what you would have had at the end of last week's lab, with all the FIXMEs fixed (you might like to check whether I fixed them in the same way you did!).

The two major changes are that the initial "hello" message exchange has been removed, and the server has been refactored so that communication with the client happens inside a function.

Check that you understand the code that uses `memcpy` with `line.c_str()` – if not, ask.

## Partial reads

In the documentation for the recv function on MSDN, note that it says something like:

> For connection-oriented sockets (type `SOCK_STREAM` for example), calling `recv` will return as much data as is currently available – up to the size of the buffer specified.

This means that you might call `recv` and only get part of the message, if it's not all arrived at the destination yet. We ignored this last week because it's very unlikely to be a problem on a fast local network, but we should deal with the problem now – we'll come back to it later when we look at non-blocking sockets. You can force `recv` to always read exactly as many bytes as you asked by passing it the `MSG_WAITALL` flag in its last argument – fix this for all the `recv` calls in the code.

## Variable-length messages

As provided, the client and server work with fixed-length messages – they always send 40 bytes and expect to receive 40 bytes. This is wasteful if the user's typed a short line, and potentially loses data if they've typed a very long line. It would be better for the length of message to vary depending on what the user typed.

However, when receiving a message, we have to say how long the message we expect to receive is. We can deal with this problem in two different ways:

- Send the message followed by a delimiter. A delimiter is a character that can't occur in the message (e.g. a `#` character). The receiver then reads small chunks of data (e.g. just a single byte at a time) until it sees the delimiter. Once it's read the delimiter, it knows it's got the whole message. With this approach, you'd send “Hello” as something like: `Hello#`
    - Think of, or implement, a solution for the case that the user includes the delimiter as part of their message. With the above example, how would the client/server handle a message that reads “Feeling #great”?
- Send the length of the message first (e.g. as a fixed-length string containing a decimal integer, or as a char/int), then send the message itself. The receiver first reads the length, decodes it, then does a read to get the rest of the message now it knows exactly how long it is. With this approach, you'd send “Hello” as something like: `00005Hello`

Both of these approaches are useful in different situations – e.g. the HTTP protocol used by web servers uses the first approach; the DNS protocol that's used to look up names of machines uses the second. Modify the code to do each of these in turn – think about why one method might be better than the other and discuss this with your peers or with myself in the lab.