COMP3310/COMP6331/ENGN3539/ENGN6539 Week 5/6: Handling Failures

In this exercise, you will observe one of the many types of failures that a network client may encounter - an unresponsive server - and attempt to modify your client to survive this failure.

This exercise builds on the exercises from the earlier tutorials; if you haven't attempted those yet, you should do that first.

As in the previous exercises, you can use Wireshark to observe what the client and server are sending, if you'd like.

The programs

Open the client and server (in whichever language you prefer) and read through them.

The client should look mostly familiar; it is very similar to the TCP echo client (the initial version, which sends a single-line request and waits for a response from the server). There are some minor differences in how lines are sent and received from the socket, just to show you that there is more than one way to approach that task in both Java and Python.

The server is also similar. Have a look at where the response to the client is sent. Will this server always send a response to the client? What do you think will happen to the client if it doesn't?

The problem

As before, open two terminals and start a server, and then a client. Send some requests and observe the responses.

What happens after the third request? Can the client recover from this situation, or do you have to disconnect and start a new client?

While network server software can often be very well-designed, thoroughly tested, and reliable, there is always the possibility that a server fails to respond. This could be caused by a programming error, a hardware or network fault, or a failure in some other system that the server relies on.

Sometimes, when a client encounters unexpected behaviour from a server, it is safest to disconnect entirely; however, sometimes we would instead prefer to retry our request, in case the

issue was transient or temporary. Before we can do that, though, we need to detect that the failure has occured. One useful way to detect many kinds of failures is with a *timeout*.

Implementing timeouts

The standard socket API that we have been using allows us to set a timeout for reads on the socket itself. Look at the documentation for your preferred programming language, and find the Socket.setSoTimeout method (for Java) / the socket.settimeout method (Python).

Try modifying the client so that writes time out after 3 seconds. For now, make it such that the program simply ignores a timeout and continues as normal with the next request.

Implementing retries

The modified program is an improvement, as we can continue using the server after a failure, but our initial request was still lost.

Now, try modifying the program such that if a read times out, the client re-sends that request a second time.

(A real network client should usually wait longer and longer between retries - this is called *backoff* - and eventually give up entirely, as the failure may be permanent.)