Note
Please ask your lab demonstrator about any questions you have regarding previous labs. It will be beneficial for you to have a good foundation early on in the semester, and we are here to help!
Please familiarise yourself with the *cnet* website at http://www.csse.uwa.edu.au/cnet/. There is an introduction on the homepage that will help you with the initial tasks.
If you are running *cnet* on the university iMacs, you may need to prepare an alias for the executable, which is located at */Volumes/cslinux/bin/cnet.*

Task 1: (files required: TICKTOCK, ticktock.c)

Examine the cnet topology file TICKTOCK and its corresponding "protocol" file ticktock.c. Together they define a simple topology of two nodes connected by a single bidirectional link. The program in ticktock.c demonstrates the use of timer events in cnet which you will later use to provide timeouts in your protocols. Notice that the "protocol" file ticktock.c first includes the cnet header file, /Volumes/cslinux/adhoc/lib/cnet.h, with which you should become very familiar.

Take a copy of the files TICKTOCK and ticktock.c into a new directory and from the command-line type:

prompt> cnet TICKTOCK

The C protocol file will be automatically compiled (with gcc) and a simple graphical representation of the two node network will be displayed. Selecting either of the nodes (Perth or Melbourne) with the left mouse button will open a new window for that node on which you see the node's timer function timeouts() ticking away.

Although this is a rather trivial exercise, make sure that you understand what is happening. Play around with the settings in the GUI, speed up/slow down the messages etc., and dig around the code to determine what is happening.

__cnet_ provides additional options when it is run on the command lines; for example, you may choose to receive statistical information on the command line instead of through the _cnet_ GUI. For the next few weeks of labsheets, it will be useful to explore the different options that may be passed to *cnet,* particularly the *statistical and seed-value options.*

Task 2: (files required: STOPANDWAIT1, STOPANDWAIT2, STOPANDWAIT3, stopandwait.c)

These files provide an implementation of the stop-and-wait protocol introduced in lectures. You will not need to modify these files, but you should execute the protocol with each of the topology files.

The topology file STOPANDWAIT1 defines the basic topology consisting of just 2 nodes and 1 link - the minimum we require for a Data-Link Layer protocol. The topology file STOPANDWAIT2 introduces frame corruption and the topology file STOPANDWAIT3 introduces frame loss.

For each of the topology files, follow the path that each frame will take "through the source code" when there's no Physical Layer problems, then frame corruption, and then frame loss.

Gather some basic statistics from cnet that show how the probabilities of frame corruption and loss affect the number of frames delivered in a fixed length of time.

Task 3: (files required: plot-to-html.not-sh)

We will finally investigate the performance of our stop-and-wait protocol - how it performs under varying characteristics of (random) frame corruption and loss. We will need to run a number of simulations, varying just one attribute while holding all others constant.

Execute the provided plot-to-html.sh shellscript (with ./plot-to-html.sh) to both run your simulations and to generate an HTML file that uses Google charts to generate a plot that we can view using a standard web-browser.

The provided shellscript produces a plot basic for some "fixed" simulation attributes; can edit the first few lines of the shellscript to run different simulations, and to focus on different metrics.

Note that you may wish to print additional or different statistics. In support of this, experiment with the EV_SHUTDOWN event, and the -f and -s command line options.

Note: Please remove the 'not-' from the file extension; LMS does not like us uploading *.sh* files...

Task 4:

Task 2 presented a complete stop-and-wait protocol which used acknowledgements to report the successful arrival of data frames. If a corrupted frame arrived, the receiver would simply ignore it and the sender's timeout mechanism would cause retransmission of the data frame.

This protocol, although robust, introduces some inefficiency because the sender is not directly (or quickly) informed about a corrupt data frame arriving at the receiver. The sender actually only infers the data frame corruption after a timeout occurs. The use of an additional negative-acknowledgement frame will reduce this inefficiency.

Modify a copy of the provided stop-and-wait protocol to also use negative-acknowledgements. Have the receiver reply with a negative-acknowledgement when a corrupt frame arrives.

You may like to use gnuplot, or any other plotting software, to show that your addition of negative-acknowledgements improves the standard protocol.

[This is an interesting problem - don't panic about the apparent need for lots of tricky code. Think about the problem - it requires a bit of logic and only about **8 lines** of code.]