CS/EE 555 Project1
Notes
September 23, 2001

These notes contain suggestions about how to approach some of the problems involved in the first project. You don't have to follow them: they are intended to provide at least a starting point for those with limited experience in concurrent programming.

The fundamental difficulty in structuring the programs is that there are several different kinds of activities that have to be managed by your programs, and those activities happen independently of each other. The four primary activities are sending a packet on behalf of the layer above (in this case, your main program), resending a packet, receiving a packet and delivering a packet to the upper layer (in this case your receiver's main program).

Early in the semester I talked about process-per-message and process-per-protocol approaches to implementation. I like to think of each process as an "engine" or "pump" that moves a message along the protocol stack. The problem faced in this assignment is that we are using an environment that doesn't provide for using more than **one** process. (Unless you use Java—the approach described here and in class Friday could be used in Java, but if using Java you should at least consider using multiple threads.)

The technique I suggest for dealing with the multiple activities is called "event-driven" programming. We use a single process that we direct to "pump" different messages in different directions at different times. You might recognize this style of programming from having developed, Windows, Mac, X or Tcl/Tk applications. The general structure of programs in these systems is:

```
…
while ( event = waitForNextEvent() ) {
     handle(event)
}
```

Your receiver program can be a particularly simple embodiment of this approach. Although in the assignment I had asked that you structure your receiver as a main loop that repeatedly calls `p1_recv(),` I think you will find it easier, and it is acceptable to me, to instead structure the receiver a main program that passes an up-call procedure, `deliver_msg` to the protocol implementation. If you take this approach, the protocol implementation would contain the above loop, using `recvfrom` to implement `waitForNextEvent` and calls to `deliver_msg` as part of the implementation of `handle.` If you want to use a structure making repeated calls to `p1_recv` you'll need to structure the receiver more like the sender as described below. Also notice that if you want to include timeout-based resends of ACKs in your receiver you'll need a more complicated structure.

The sender program is necessarily more complicated because here we **have** to deal with outgoing messages, timeout-based resends of outgoing messages, and incoming ACKs.

If you follow the implementation outline from Chapter 2 of the book, you will discover the difficulty when you reach the point in the implementation of send where it waits when the send window is full. What has to happen here? Your program must wait for there to be space in the send window. That will only happen when the ACK for the oldest message in the send window is received. So at this point we must definitely provide a way to notice the arrival of ACKs. But if that oldest message or its ACK was lost we will have to resend it, which means we must also be prepared to do time-out based resends at this point. With all that in mind, the following structure for `p1_send` seems to be required (where s is the socket on which the ACK will be received):

```
…
while ( sendWindowFull() ) {
     result = waitForAckOrTimeout(s, timeToNextEvent)
     if (result==timeoutOccurred) {
          act on timeout
     } else {
          process incoming ack
     }
}
send message – and schedule a timeout to resend it
```

`waitForAckOrTimeout` can be implemented using `select,` you just need to do the necessary bookkeeping to know the time to the next event.

You may find it useful to implement an event scheduling package such as described in Chapter 1 and for which I have provided an example header file, ev.h, in the project directory. You could also, and it might be simpler in this case, maintain a time-ordered list of resend events and process them explicitly from “act on timeout” in the above loop.

Have fun!