

CSIS0234B Computer and Communication Networks (Class B)

Assignment 4

Tutor: tqwang@csis.hku.hk, Deadline Jun 1, 2003, 5:00pm.

In tutorial 3, we wrote a datagram server and client, which sends and receives a file using datagrams. But at the end of the tutorial, we noted that it is not very reliable: when sending large files, it is possible that some packets are dropped (or even misordered), and as a result the two programs hangs there waiting for acknowledgement.

In this assignment, you are required to modify the programs so that it allows the program to work in various kinds of networks, while still using just UDP. What we need is to reimplement part of TCP to provide reliability to the unreliable datagram abstraction provided by UDP.

This is a group assignment allowing 1–2 students in each group.

Requirements:

- The functional requirement is the same as the tutorial: the client and the server should be able to transfer file, the server always read the file from `orig.txt`, the client always save it as `received.txt`. There is an exception: the port number should be specified on the command line (alongside with the hostname for the client), which would make it easier to work when the port is used by another student.
- The large file should be sent in small-sized datagrams. Your program should have a constant for specifying the size, which can be modified at compile-time to be between 1 byte and 60000 bytes. All packets except the last should be of this size (plus application level header) in size, and a packet with a smaller size indicate the desire to terminate a connection.
- A windowing protocol should be used to guard against packet lost, duplication and re-order. Since there is only one direction of transfer, there is no need for piggybacking. On the other hand, negative acknowledgement should be used to speed up the recovery from lost datagrams.
- The sender should continuously measure the delay incurred by packets until the acknowledgement is received. A weighted average of this delay should be used to set the timeout period (the variance can be assumed small, so that it is safe to assume normally packets round-trip time should not vary by more than 40%). Resent frames should not be considered when computing the weighted average (since you won't know whether it is due to which datagram you sent). Instead, resent frames should simply increase the estimate by 10%. You should expect the network to never delay a packet for more than 15 seconds (i.e., RTT of 30 seconds), so you should resend at least every 30 seconds if you cannot receive anything.
- Both the sender and the receiver should stop and signal an error if they wait for more than 1 minute.
- The program should use TCP style AIMD to dynamically adjust the size of sending (congestion) window, so that when the network bandwidth changes, an appropriate number of packets are sent (rather than having most of them lost). The initial threshold should be 100 packets. In contrast, there should be no limit on receiving window size, since the program runs in user-mode and do not have concerns about limited buffer size.

You may not have access to a network that you can control the load for effectively testing your program. Therefore, we provide a program in the web page of the course. The program opens

two UDP sockets, and copy datagrams arriving from one UDP port to the other. It can be configured at runtime to control the amount of delay before sending, the maximum number of packets that can send through the network per second, and the probability that a packet is lost (and therefore not copied). E.g., the following sets up the program so that datagrams sent to port 12345 is resent to port 12346 of localhost:

```
> relay 12345 localhost 12346
Relay [d=0.0000, o=0.0000, r=50000.0]: d 0.01
Available commands:
  omit x: omit a fraction x of datagrams
  delay x: delay x seconds
  rate x: limit each side to x datagrams per second
(x can be floating point numbers)
Relay [d=0.0000, o=0.0000, r=50000.0000]: d 0.01
Relay [d=0.0100, o=0.0000, r=50000.0000]: r 50
Relay [d=0.0100, o=0.0000, r=50.0000]:
```

Hints

- To measure the time between sending a frame and an acknowledgement is received, you can use *gettimeofday()* to get the time and perform subtraction.
- You need to wait for a frame in such a way that if after some time the frame is not received, then you want the system call to stop. We suggest that you use the *setitimer()* system call to setup a signal to occur after a certain amount of time. You should setup a very simple signal handler for SIGALRM using *sigaction()*, so that a variable is set if timer expires. Then you can make the *recv* call as usual, and if the timer expires, the *recv* call will return -1, with *errno* set to *EINTR* (interrupted system call), and you should use the variable set in the signal handler to check whether a timeout occurred. Your code might look like this:

```
bool timeout_occurred = false;  
  
void timeout_handler(int) {  
    timeout_occurred = true;  
}  
  
void setup_handler() {  
    struct sigaction action;  
    action.sa_handler = timeout_handler;  
    sigemptyset(&action.sa_mask);  
    action.sa_flags = 0;  
    sigaction(SIGALRM, &action, 0);  
}  
  
void setalarm(float value) {  
    struct itimerval val;  
    val.it_interval.tv_sec = val.it_interval.tv_usec = 0;  
    val.it_value.tv_sec = int(value);  
    val.it_value.tv_usec = int((value - int(value)) * 1000000);  
    setitimer(ITIMER_REAL, &val, 0);  
    timeout_occurred = false;
```

}

- You need at least 3 types of packets: ACK, NAK and DATA. A request can be done by an ACK with sequence number -1, which fits very nicely with the remaining code of the program.
- You don't need to deal with connection establishment. There should be a disconnection timeout, which can be implemented by recording the last time when any message is received by the other side.
- The following shows the structure of a simple windowing algorithm. You may base your implementation on this algorithm, and add code to deal with delay estimation and congestion control.

```
sender() {  
    curr = winstart = winend = 0           // window edges and current packet  
    winsize = 1                           // maximum window size  
    timer = 10.0                           // how long to wait before resend  
    for (;;) {  
        while (curr < winstart + winsize && curr is still within file) {  
            send_data(curr) // send frame number curr  
            if (curr is a partial packet)  
                break  
            curr += 1  
            winend = curr  
        }  
        setalarm() // to timer sec after sendtime(winstart)  
        rcv(msg)  
        if (timeout) {  
            curr = winstart  
            continue  
        }  
        seq = msg.seq  
        if (msg.type == ack) {  
            while (seq >= winstart)  
                ++winstart  
            // should also cleanup for winstart  
        } else if (msg.type == nak) {  
            send_data(seq)  
        }  
    }  
}
```

```
receiver() {
    last = -1
    timer = 10.0
    nak_sent = false
    for (;) {
        send_ack(last)
        setalarm()           // to timer sec from now
        recv(msg)
        if (timeout)
            continue
        if (msg.seq > last + 1) {
            if (!nak_sent)           { // don't send multiple nak for a packet
                send_nak(last+1)
                nak_sent = true
            }
            // save the packet in some data structure (e.g., map)
        } else {
            nak_sent = false
            write the packet to the file
            // write saved packets that are in sequence
            last = last packet output to file
            if (last frame is partial) {           // i.e., EOF
                send_ack(last)
                break
            }
        }
    }
}
```