

CSIS0234B Computer and Communication Networks (Class B)
Assignment 4
Reliability and Congestion Control over UDP
Tutor: cmtsang@csis.hku.hk , Deadline May 23, 2004, 5:00pm.

In tutorial 3, we wrote a datagram server and client, which sends and receives a file using datagrams respectively. But they are not very reliable: if any packet is dropped, the two programs hangs there waiting for each other; and if packets are misordered, or the client get incorrect data.

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 specified by a command line argument, the client write it to standard output (you can use `stderr` for debug messages). The port number should also be specified on the command line (alongside with the hostname for the client), to make it more flexible (the use of the `relay` program as specified later will require you to specify different ports). The server should be an iterative server expecting multiple connections one by one.
- Large files should be sent in small-sized datagrams. The datagram size should be automatically found using the **PMTU** (Path Maximum Transfer Unit) **Discovery protocol**. (See the man page `ip(7)` in Linux. The use of this option limits the program to execute only in Linux.) You may assume that MTU will not change during a connection.
- The **TCP style windowing protocol** should be used to guard against packet lost, duplication and re-order. Unlike TCP, packet numbers are used as sequence numbers, starting from 0. Acknowledgements are always explicitly sent without delay or piggybacking.
- TCP style **fast-retransmit** and **fast-recovery** should be used: at the third time you receive a ACKs, the acknowledged frame should be retransmitted without waiting for timeout. The receiver should buffer out-of-order packets so that if only one data packet is lost, only that packet will be resent. This should immediately set the threshold to half the congestion window, and set the congestion window to 3 plus the new value of the threshold. Further duplicated acknowledgements should increment the congestion window, and a new acknowledgement would set the congestion window to the threshold, resuming normal processing. The timer should not be reset, so that once the original timeout occurs, packets are resent with a full slow start.
- For measuring RTT, each side should **send timestamps** for measuring the delay incurred by packets. Such timestamps should be echoed by the other side. This is similar to TCP timestamp extension, but you do not need to stick to the TCP rules as specified in RFC 1323. Instead, use a strategy that estimate delays accurately.
- The estimated mean round trip time is used to set the retransmission timeout (RTO), assuming the variance is negligible. The setting of RTO should take into account the need for longer timeout for fast-retransmit to be effective.

- TCP style **AIMD** should be used 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). Loss of packets is used as an indication of congestion. The initial threshold should be 64KiB. In contrast, the **receiving buffer** should be fixed at 64KiB, i.e., the default UDP buffer size. This is because acknowledgements are done by the user program.
- When the retransmission timer fires, the server should temporarily double the RTO. If the resulting RTO becomes larger than 30 seconds, the server should assume the connection is lost. The RTT is used again once a new acknowledgement arrives.
- Make the following guarantee: if there is only one client, then both the server will eventually reset to the starting state, and the client will eventually finish, even when some packets are lost.

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]: help
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]:
```

The program does not relay ICMP packets needed for PMTU discovery, so it must be executed in the client side of the connection (the packets from the relay program to the destination will allow fragmentation).

Hints

- You need to find the current time and put it to the packet in order to successfully measure the RTT. You may find *gettimeofday()* useful.
- You need to set the IP level `IP_MTU_DISCOVER` option to `IP_PMTUDISC_DO` so that the DF bit of the IP header is set. Here's a simple way to use it: Start by setting the option and connecting to the client UDP socket. Then get the `IP_MTU` socket option to find the currently known MTU for that destination. Use that as the MTU, subtract the length of IP header, UDP header and your own application header to get the MSS (Maximum segment size), send a packet of that length, and start receiving. If the packet is still too large for some part of the network, your call to *recv()* will get an error, with *errno* set to *EMSGSIZE*. At this time you can repeat the process to get the `IP_MTU` option and send a smaller packet. Note that Linux refuse to give a `IP_MTU` smaller than 552, so you have to check whether the

IP_MTU value is really getting smaller: if it is not you have to disable IP_MTU_DISCOVER. You might also need to define the IP_MTU constant yourselves (to the integer literal 14), since not all distributions have it in the user-level headers.

- 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) { // set timeout_occurred = false first  
    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);  
}
```

- You will need special packets for connection management, e.g., to start a transfer, to end a transfer, and to reset a connection. This allows your client to terminate correctly even if the disconnection packet is lost.
- The following shows the structure of a simple windowing algorithm. You may base your implementation on this algorithm, and add code to deal with PMTU discovery, delay estimation and congestion control.

```
sender() {  
    curr = winstart = 0 // current packet and left window edge  
    winsize = 1 // maximum window size  
    timer = 1.0 // how long to wait before resend  
    timer_running = false  
    for (;;) {  
        while (int(winstart + winsize - curr) > 0) {  
            if (curr is past the EOF)  
                break  
            if (!timer_running) {  
                timer_running = true  
                setalarm(timer)  
            }  
        }  
    }  
}
```

```
    }
    send_data(curr) // send datagram #curr
    curr += 1
}
recv(msg)
if (timeout) {
    timer_running = false
    curr = winstart // causes resend to start
    winsize = 1 // slow start begins
    continue
}
if (msg.seq + 1 == winstart && repeated the 3rd time)
    send_data(msg.seq) // resend the packet
else if (msg.seq >= winstart) {
    timer_running = false
    setalarm(0)
    winstart = msg.seq
}
if (msg.seq is past the EOF)
    return // finish sending file
}
send_fin()
}

receiver() {
    ack_nr = 0
    timer = 1.0
    for (;) {
        send_ack(ack_nr)
        setalarm(timer)
        recv(msg)
        if (timeout)
            continue // send the ack again
        if (msg.type == fin) // i.e., EOF
            break
        if (int(msg.seq - (ack_nr + winsize)) > 0 ||
            int(ack_nr - msg.seq) > 0)
            continue // send ack immediately
        if (msg.seq == ack_nr) {
            write the packet to the file
            // write saved packets that are in sequence
            ack_nr = last packet output to file + 1
        }
        else // out-of-sequence datagram
            // save the packet in some data structure (e.g., map)
    }
}
}
```