Network Programming

Assignment 2

Last updated October 8.

Due Tuesday, October 30th.

You are to work on this assignment in groups of two.

The aim of this assignment is to have you do *UDP* socket client / server programming with a focus on two broad aspects :

- Setting up the exchange between the client and server in a secure way despite the lack of a formal connection (as in *TCP*) between the two, so that 'outsider' *UDP* datagrams (broadcast, multicast, unicast - fortuitously or maliciously) cannot intrude on the communication.

- Introducing application-layer protocol data-transmission reliability, flow control and congestion control in the client and server using *TCP*-like *ARQ* sliding window mechanisms.

The second item above is much more of a challenge to implement than the first, though neither is particularly trivial. But they are not tightly interdependent; each can be worked on separately at first and then integrated together at a later stage.

Apart from the material in Chapters 8, 14 & 22 (especially Sections 22.5 - 22.7), and the experience you gained from the preceding assignment, you will also need to refer to the following :

- *ioctl* function (Chapter 17).

- *get_ifi_info* function (Section 17.6, Chapter 17). This function will be used by the server code to discover its node's network interfaces so that it can *bind* all its interface *IP* addresses (see Section 22.6).

- 'Race' conditions (Section 20.5, Chapter 20)

You also need a thorough understanding of how the *TCP* protocol implements reliable data transfer, flow control and congestion control. Chapters 17- 24 of *TCP/IP Illustrated, Volume 1* by W. Richard Stevens gives a good overview of *TCP*. Though somewhat dated for some things (it was published in 1994), it remains, overall, a good basic reference.

Overview

This assignment asks you to implement a primitive file transfer protocol for *Unix* platforms, based on *UDP*, and with *TCP*-like reliability added to the transfer operation using timeouts and sliding-window mechanisms, and implementing flow and congestion control. The server is a concurrent server which can handle multiple clients simultaneously. A client gives the server the name of a file. The server forks off a child which reads directly from the file and transfers the contents over to the client using *UDP* datagrams. The client prints out the file contents as they come in, in order, with nothing missing and with no duplication of content, directly on to *stdout* (via the receiver sliding window, of course, but with no other intermediate buffering). The file to be transferred can be of arbitrary length, but its contents are always straightforward *ascii* text.

As an aside let me mention that assuming the file contents *ascii* is not as restrictive as it sounds. We can always pretend, for example, that binary files are *base64 encoded* (“*ASCII armor*”). A real file transfer protocol would, of course, have to worry about transferring files between heterogeneous platforms with different file structure conventions and semantics. The sender would first have to transform the file into a platform-independent, protocol-defined, format (using, say, *ASN.1*, or some such standard), and the receiver would have to transform the received file into its platform’s native file format. This kind of thing can be fairly time consuming, and is certainly very tedious, to implement, with little educational value - it is not part of this assignment.

Arguments for the server

You should provide the server with an input file *server.in* from which it reads the following information, **in the order shown, one item per line** :

- Well-known port number for server.
- Maximum sending sliding-window size (in datagram units).

You will not be handing in your *server.in* file. We shall create our own when we come to test your code. So it is important that you stick strictly to the file name and content conventions specified above. The same applies to the *client.in* input file below.

Arguments for the client

The client is to be provided with an input file *client.in* from which it reads the following information, **in the order shown, one item per line** :

- *IP* address of server (**not** the *hostname*).
- Well-known port number of server.
- *filename* to be transferred.
- Receiving sliding-window size (in datagram units).
- Random generator *seed* value.
- Probability *p* of datagram loss. This should be a real number in the range [0.0 , 1.0] (value 0.0 means no loss occurs; value 1.0 means all datagrams all lost).
- The mean *μ*, **in milliseconds**, for an exponential distribution controlling the rate at which the client reads received datagram payloads from its receive buffer.

Operation

1. Server starts up and reads its arguments from file *server.in*.

   As we shall see, when a client communicates with the server, the server will want to know what *IP* address that client is using to identify the server (*i.e.* , the destination *IP* address in the incoming datagram). Normally, this can be done relatively straightforwardly using the *IP_RECVDESTADDR* socket option, and picking up the information using the ancillary data ('control information') capability of the *recvmsg* function. Unfortunately, Solaris 2.10 does not support the *IP_RECVDESTADDR* option (nor, incidentally, does it support the *msg_flags* option in *msghdr* - see *p*.390). This considerably complicates things.

   In the absence of *IP_RECVDESTADDR*, what the server has to do as part of its initialization phase is to *bind* each *IP* address it has (and, simultaneously, its well-known port number, which it has read in from *server.in*) to a separate *UDP* socket. The code in Section 22.6, which uses the *get_ifi_info* function, shows you how to do that. However, there are important differences between that code and the version you want to implement.

The code of Section 22.6 binds the *IP* addresses *and forks off a child* for each address that is bound to. We do not want to do that. Instead you should have an array of socket descriptors. For each *IP* address, create a new socket and *bind* the address (and well-known port number) to the socket *without* forking off child processes. Creating child processes comes later, when clients arrive.

The code of Section 22.6 also attempts to *bind* broadcast addresses. We do not want to do this. It *bind*s a wildcard *IP* address, which we certainly do not want to do either. We should *bind* strictly only unicast addresses (including the loopback address).

The *get_ifi_info* function (which the code in Section 22.6 uses) has to be modified so that it also gets the network masks for the *IP* addresses of the node, and adds these to the information stored in the linked list of *ifi_info* structures (see Figure 17.5, *p*.471) it produces. As you go binding each *IP* address to a distinct socket, it will be useful for later processing to build your own array of structures, where a structure element records the following information for each socket :

- *sockfd*
- *IP* address bound to the socket
- network mask for the *IP* address
- subnet address (obtained by doing a bit-wise *and* between the *IP* address and its network mask)

Report, in a *ReadMe* file which you hand in with your code, on the modifications you had to introduce to ensure that only unicast addresses are bound, and on your implementation of the array of structures described above.

You should print out on *stdout,* with an appropriate message and appropriately formatted in dotted decimal notation, the *IP* address, network mask, and subnet address for each socket in your array of structures (you do not need to print the *sockfd*).

The server now uses *select* to monitor the sockets it has created for incoming datagrams. When it returns from *select*, it must use *recvfrom* or *recvmsg* to read the incoming datagram (see 6. below).

2. When a client starts, it first reads its arguments from the file *client.in*.

3. The client checks if the server host is ‘local’ to its (extended) Ethernet. If so, *all* its communication to the server is to occur as *MSG_DONTROUTE* (or *SO_DONTROUTE* socket option). It determines if the server host is ‘local’ as follows.

   The first thing the client should do is to use the modified *get_ifi_info* function to obtain all of its *IP* addresses and associated network masks.

   Print out on *stdout*, in dotted decimal notation and with an appropriate message, the *IP* addresses and network masks obtained.

   In the following, *IPserver* designates the *IP* address the client will use to identify the server, and *IPclient* designates the *IP* address the client will choose to identify itself.

   The client checks whether the server is on the same host. If so, it should use the loopback address 127.0.0.1 for the server (*i.e.* , *IPserver* = 127.0.0.1). *IPclient* should also be set to the loopback address.

   Otherwise it proceeds as follows:

   IPserver is set to the *IP* address for the server in the *client.in* file. Given *IPserver* and the (unicast) *IP* addresses and network masks for the client returned by *get_ifi_info* in the linked list of *ifi_info* structures, you should be able to figure out if the server node is ‘local’ or not. This will be discussed in class; but let me just remind you here that you should use ‘*longest prefix matching*’ where applicable.

   If there are multiple client addresses, and the server host is ‘local’, the client chooses an *IP* address for itself, *IPclient*, which matches up as ‘local’ according to your examination above. If the server host is not ‘local’, then *IPclient* can be chosen arbitrarily.

Print out on *stdout* the results of your examination, as to whether the server host is 'local' or not, as well as the *IPclient* and *IPserver* addresses selected.

Note that this manner of determining whether the server is local or not is somewhat clumsy and 'over-engineered', and, as such, should be viewed more in the nature of a pedagogical exercise. Ideally, we would like to look up the server *IP* address(es) in the routing table (see Section 18.3). This requires that a routing socket be created, for which we need superuser privilege. Alternatively, we might want to dump out the routing table, using the *sysctl* function for example (see Section 18.4), and examine it directly. Unfortunately, *Solaris 2.10* does not support *sysctl*.

More to the point, using *MSG_DONTROUTE* where possible would seem to gain us efficiency, in as much as the kernel does not need to consult the routing table for every datagram sent. But, in fact, that is not so. Recall that one effect of *connect* with *UDP* sockets is that routing information is obtained by the kernel at the time the *connect* is issued. That information is cached and used for subsequent sends from the connected socket (see *p*.255).

4. The client now creates a *UDP* socket and calls *bind* on *IPclient*, with 0 as the port number. This will cause the kernel to bind an ephemeral port to the socket.

   After the *bind*, use the *getsockname* function (Section 4.10) to obtain *IPclient* and the ephemeral port number that has been assigned to the socket, and print that information out on *stdout*, with an appropriate message and appropriately formatted.

   The client *connect*s its socket to *IPserver* and the well-known port number of the server.

   After the *connect*, use the *getpeername* function (Section 4.10) to obtain *IPserver* and the well-known port number of the server, and print that information out on *stdout*, with an appropriate message and appropriately formatted.

   The client sends a datagram to the server giving the *filename* for the transfer. This send needs to be backed up by a timeout in case the datagram is lost.

5. Note that the incoming datagram from the client will be delivered to the server at the socket to which the destination

IP address that the datagram is carrying has been bound. Thus, the server can obtain that address (it is, of course, *IPserver*) and thereby achieve what *IP_RECVDESTADDR* would have given us had it been available.

Furthermore, the server process can obtain the *IP* address (this will, of course, be *IPclient*) and ephemeral port number of the client through the *recvfrom* or *recvmsg* functions.

The server forks off a child process to handle the client. The server parent process goes back to the *select* to listen for new clients. Hereafter, and unless otherwise stated, whenever we refer to the ‘server’, we mean the server child process handling the client’s file transfer, not the server parent process.

6. Typically, the first thing the server child would be expected to do is to close all sockets it ‘inherits’ from its parent. However, this is not the case with us. The server child does indeed close the sockets it inherited, *but not the socket on which the client request arrived*. It leaves that socket open for now. Call this socket the ‘listening’ socket.

   The server (child) then checks if the client host is local to its (extended) Ethernet. If so, *all* its communication to the client is to occur as *MSG_DONTROUTE* (or *SO_DONTROUTE* socket option).

   > If *IPserver* (obtained in 5. above) is the loopback address, then we are done. Otherwise, the server has to proceed with the following step.
   >
   > Use the array of structures you built in 1. above, together with the addresses *IPserver* and *IPclient* to determine if the client is ‘local’.
   >
   > Print out on *stdout* the results of your examination, as to whether the client host is ‘local’ or not.

7. The server (child) creates a *UDP* socket to handle file transfer to the client. Call this socket the ‘connection’ socket. It *bind*s the socket to *IPserver*, with port number 0 so that its kernel assigns an ephemeral port.

   After the *bind*, use the *getsockname* function (Section 4.10) to obtain *IPserver* and the ephemeral port number that has been assigned to the socket, and print that information out on *stdout*, with an appropriate message and appropriately formatted.

The server then *connect*s this 'connection' socket to the client's *IPclient* and ephemeral port number.

The server now sends the client a datagram, in which it passes it the ephemeral port number of its 'connection' socket as the data payload of the datagram. This datagram is sent using the 'listening' socket inherited from its parent, otherwise the client (whose socket is connected to the server's 'listening' socket at the latter's well-known port number) will reject it. This datagram must be backed up by the *ARQ* mechanism, and retransmitted in the event of loss.

Note that if this datagram is indeed lost, the client might well time out and retransmit its original request message (the one carrying the file name). In this event, you must somehow ensure that the parent server does not mistake this retransmitted request for a new client coming in, and spawn off yet another child to handle it. How do you do that? It is potentially more involved than it might seem. I will be discussing this in class, as well as 'race' conditions that could potentially arise, depending on how you code the mechanisms I present.

When the client receives the datagram carrying the ephemeral port number of the server's 'connection' socket, it re*connect*s its socket to the server's 'connection' socket, using *IPserver* and the ephemeral port number received in the datagram (see *p*.254). It now uses this reconnected socket to send the server an acknowledgment. Note that this implies that, in the event of the server timing out, it should retransmit two copies of its 'ephemeral port number' message, one on its 'listening' socket and the other on its 'connection' socket (why?).

When the server receives the acknowledgment, it closes the 'listening' socket it inherited from its parent. The server can now commence the file transfer through its 'connection' socket.

The net effect of all these *bind*s and *connect*s at server and client is that no 'outsider' *UDP* datagram (broadcast, multicast, unicast - fortuitously or maliciously) can now intrude on the communication between server and client.

8. Starting with the first datagram sent out, the client behaves

as follows.

Whenever a datagram arrives, or an *ACK* is about to be sent out (or, indeed, the initial datagram to the server giving the *filename* for the transfer), the client uses some random number generator function *random*() (initialized by the *client.in* argument value *seed*) to decide with probability *p* (another *client.in* argument value) if the datagram or *ACK* should be discarded by way of simulating transmission loss across the network. (I will briefly discuss in class how you do this.)

Adding reliability to *UDP*

The mechanisms you are to implement are based on *TCP Reno*. These include :

- Reliable data transmission using *ARQ* sliding-windows, with *Fast Retransmit*.
- Flow control via receiver window advertisements.
- Congestion control that implements :
    - *SlowStart*
    - *Congestion Avoidance* (‘*Additive-Increase/Multiplicative Decrease*’ – *AIMD*)
    - *Fast Recovery* (but without the *window-inflation* aspect of *Fast Recovery*)

Only some, and by no means all, of the details for these are covered below. The rest will be presented in class, especially those concerning flow control and *TCP Reno*’s congestion control mechanisms in general : *Slow Start*, *Congestion Avoidance*, *Fast Retransmit* and *Fast Recovery*.

1. Implement a timeout mechanism on the sender (server) side. This is available to you from Stevens, Section 22.5 . Note, however, that you will need to modify the basic driving mechanism of Figure 22.7 appropriately since the situation at the sender side is not a repetitive cycle of send-receive, but rather a straightforward progression of send-send-send-send- . . . .

   Also, modify the *RTT* and *RTO* mechanisms of Section 22.5 as specified below. I will be discussing the details of these modifications and the reasons for them in class.

    - Modify function *rtt_stop* (Fig. 22.13) so that it uses integer arithmetic rather than floating point. This will entail your also having to modify some of the variable and function parameter declarations throughout Section 22.5 from *float* to *int*, as appropriate.

- In the *unprrt.h* header file (Fig. 22.10) set :
  RTT_RXTMIN to 1000 msec. (1 sec. instead of the current value 3 sec.)
  RTT_RXTMAX to 3000 msec. (3 sec. instead of the current value 60 sec.)
  RTT_MAXNREXMT to 12 (instead of the current value 3)

- In function *rtt_timeout* (Fig. 22.14), after doubling the *RTO* in line 86, pass its value through the function *rtt_minmax* of Fig. 22.11 (somewhat along the lines of what is done in line 77 of *rtt_stop*, Fig. 22.13).

- Finally, note that with the modification to integer calculation of the smoothed *RTT* and its variation, and given the small *RTT* values you will experience on the *cs* / *sbpub* network, these calculations should probably now be done on a millisecond or even microsecond scale (rather than in seconds, as is the case with Stevens' code). Otherwise, small measured *RTT*s could show up as 0 on a scale of seconds, yielding a negative result when we subtract the smoothed *RTT* from the measured *RTT* (line 72 of *rtt_stop*, Fig. 22.13).

Report the details of your modifications to the code of Section 22.5 in the *ReadMe* file which you hand in with your code.

2. We need to have a sender sliding window mechanism for the retransmission of lost datagrams; and a receiver sliding window in order to ensure correct sequencing of received file contents, and some measure of flow control. You should implement something based on *TCP Reno*'s mechanisms, with cumulative acknowledgments, receiver window advertisements, and a congestion control mechanism I will explain in detail in class.

   For a reference on *TCP*'s mechanisms generally, see W. Richard Stevens, *TCP/IP Illustrated, Volume 1* , especially Sections 20.2 - 20.4 of Chapter 20 , and Sections 21.1 - 21.8 of Chapter 21 .

   Bear in mind that our sequence numbers should count datagrams, not bytes as in *TCP*. Remember that the sender and receiver window sizes have to be set according to the argument values in *client.in* and *server.in*, respectively. Whenever the sender window becomes full and so ‘locks’, the server should print out a message to that effect on *stdout*. Similarly, whenever the receiver window ‘locks’, the client should print out a message on *stdout*.

Be aware of the potential for deadlock when the receiver window ‘locks’. This situation is handled by having the receiver process send a duplicate *ACK* which acts as a *window update* when its window opens again (see Figure 20.3 and the discussion about it in *TCP/IP Illustrated*). However, this is not enough, because *ACK*s are not backed up by a timeout mechanism in the event they are lost. So we will also need to implement a *persist timer* driving *window probes* in the sender process (see Sections 22.1 & 22.2 in Chapter 22 of *TCP/IP Illustrated*). Note that you do not have to worry about the *Silly Window Syndrome* discussed in Section 22.3 of *TCP/IP Illustrated* since the receiver process consumes ‘full sized’ 512-byte messages from the receiver buffer (see 3. below).

Report on the details of the *ARQ* mechanism you implemented in the *ReadMe* file you hand in. Indeed, you should report on **all** the *TCP* mechanisms you implemented in the *ReadMe* file, both the ones discussed here, and the ones I will be discussing in class.

3. Make your datagram payload a fixed 512 bytes, inclusive of the file transfer protocol header (which must, at the very least, carry: the sequence number of the datagram; *ACK*s; and advertised window notifications).

4. The client reads the file contents in its receive buffer and prints them out on *stdout* **using a separate thread**. This thread sits in a repetitive loop till all the file contents have been printed out, doing the following.

   It samples from an exponential distribution with mean μ milliseconds (read from the *client.in* file), sleeps for that number of milliseconds; wakes up to read and print all in-order file contents available in the receive buffer at that point; samples again from the exponential distribution; sleeps; and so on.

   The formula $-1 \times \mu \times ln(random())$, where *ln* is the natural logarithm, yields variates from an exponential distribution with mean μ, based on the uniformly-distributed variates over $(0 , 1)$ returned by *random*().

Note that you will need to implement some sort of mutual exclusion/semaphore mechanism on the client side so that the thread that sleeps and wakes up to consume from the receive buffer is not updating the state variables of the buffer at the same time as the main thread reading from the socket and depositing into the buffer is doing the same. Furthermore, we need to ensure that the main thread does not effectively monopolize the semaphore (and thus lock out for prolonged periods of time) the sleeping thread when the latter wakes up. See the textbook, Section 26.7, ‘*Mutexes: Mutual Exclusion*’, *pp*.697-701. You might also find Section 26.8, ‘*Condition Variables*’, *pp*.701-705, useful.

5. You will need to devise some way by which the sender can notify the receiver when it has sent the last datagram of the file transfer, without the receiver mistaking that *EOF* marker as part of the file contents. (Also, note that the last data segment could be a “short” segment of less than 512 bytes – your client needs to be able to handle this correctly somehow.) When the sender receives an *ACK* for the last datagram of the transfer, the (child) server terminates. The parent server has to take care of cleaning up zombie children.

   Note that if we want a clean closing, the client process cannot simply terminate when the receiver *ACK*s the last datagram. This *ACK* could be lost, which would leave the (child) server process ‘hanging’, timing out, and retransmitting the last datagram. *TCP* attempts to deal with this problem by means of the *TIME_WAIT* state. You should have your receiver process behave similarly, sticking around in something akin to a *TIME_WAIT* state in case in case it needs to retransmit the *ACK*.

   In the *ReadMe* file you hand in, report on how you dealt with the issues raised here: sender notifying receiver of the last datagram, clean closing, and so on.

Output

Some of the output required from your program has been described in the section *Operation* above. I expect you to provide further output – clear, well-structured, well-laid-out, concise but sufficient and helpful – in the client and server windows by means of which we can trace the correct evolution of your *TCP*’s behaviour in all its intricacies : information (*e.g.*, sequence number) on datagrams and *acks* sent and dropped, window advertisements, datagram retransmissions (and why :

dup acks or *RTO*); entering/exiting *Slow Start* and *Congestion Avoidance*, *ssthresh* and *cwnd* values; sender and receiver windows locking/unlocking; *etc.*, *etc.* . . . .

> The onus is on you to convince us that the *TCP* mechanisms you implemented are working correctly. Too many students do not put sufficient thought, creative imagination, time or effort into this. It is not the TA's nor my responsibility to sit staring at an essentially blank screen, trying to summon up our paranormal psychology skills to figure out if your *TCP* implementation is really working correctly in all its very intricate aspects, simply because the transferred file seems to be printing o.k. in the client window. Nor is it our responsibility to strain our eyes and our patience wading through a mountain of obscure, ill-structured, hyper-messy, debugging-style output because, for example, your effort-conserving concept of what is 'suitable' is to dump your debugging output on us, relevant, irrelevant, and everything in between.

Hand-in

The criterion for a successful assignment is that it execute correctly on the Solaris 10 *compserv* (not the Linux *compute*) nodes in the *cs.sunysb.edu* domain; with clear, well-structured output that convinces us that the mechanisms you implemented are working correctly.

We shall be testing your code by running clients and server between the following machines, to which you also have access in order to test your code before handing in. Note that, besides the *compserv* nodes on network 130.245.1.0/24 (and the *compserv{1,2,3,4}bn* IP-aliased "nodes" discussed in class), you also have available to you the *sbpub* machines on network 130.245.6.0/24 .

You should submit your code using the electronic hand-in procedure provided. Your submission must absolutely include a *Makefile* which :

- compiles your code using, where necessary, the Stevens' environment in the course account, *~cse533/Stevens /unpv13e_solaris2.10*; and
- gives the standard names *client* and *server* for the client & server executable produced.

Each group hands in just one copy of the Assignment, under either partner's login name. If you mis-coordinate with your partner and each hands in a copy under his/her name, make sure one of you resubmits just a *ReadMe* file, and nothing else, saying something to the effect of "Please ignore this submission". Please do not make us grade the same thing twice over because you are unable to get this simple coordination with your partner straight.

Do not forget to hand in the *ReadMe* file mentioned in item 1. of the section *Operation*, and items 1, 2 & 5 of the section *Adding reliability to UDP*. The first thing the *ReadMe* file should contain is an identification of the members in the group.
