CSC/ECE 573 – Internet Protocols

Project #2: Reliable Multicast
Due Date: October 31, 2011

Project Objectives
In this project, you will implement a point-to-multipoint reliable data transfer protocol using a multicast automatic repeat request (ARQ) scheme, and you will carry out a number of experiments to evaluate its performance. In the process I expect that you will develop a good understanding of ARQ schemes and reliable data transfer protocols and build a number of fundamental skills related to writing transport layer services, including:

- encapsulating application data into transport layer segments by including transport headers,
- buffering and managing data received from, or to be delivered to, multiple destinations,
- managing the window size at the sender,
- computing checksums, and
- using the UDP socket interface.

Point-to-Multipoint File Transfer Protocol (P2MP-FTP)
The FTP protocol provides a sophisticated file transfer service, but since it uses TCP to ensure reliable data transmission it only supports the transfer of files from one sender to one receiver. In many applications (e.g., software updates, stock quote updates, document sharing, etc) it is important to transfer data reliably from one sender to multiple receivers. You will implement P2MP-FTP, a protocol that provides a simple service: transferring a file from one host to multiple destinations. P2MP-FTP will use UDP to send packets from the sending host to each of the destinations, hence it has to implement a reliable data transfer service using some ARQ scheme; for this project, you will implement the a multicast ARQ scheme that will be described shortly. Using the unreliable UDP protocol allows us to implement a "transport layer" service such as reliable data transfer in user space.

Client-Server Architecture of P2MP-FTP
To keep things simple, you will implement P2MP-FTP in a client-server architecture and omit the steps of opening up and terminating a connection. The P2MP-FTP client will play the role of the *sender* that connects to a set of of P2MP-FTP servers that play the role of the *receivers* in the reliable data transfer. All data transfer is from sender to receivers; only ACK packets travel from receivers to sender.

The P2MP-FTP Client (Sender)
The P2MP-FTP client implements the sender in the reliable data transfer. When the client starts, it reads several values from the command line, including:

- a file name that contains the data to be transferred reliably to all receivers (servers);
- the maximum segment size (MSS); and
- the window size N.

The client then reads data from the file specified in the command line, and calls *rdt_send()* to transfer the data to the P2MP-FTP receivers. For this project, we will assume that *rdt_send()* provides data from the file on a byte basis. The client also implements the sending side of the reliable multicast protocol, receiving data from *rdt_send()*, buffering the data locally, and ensuring that the data is received correctly at the server.

The sender buffers the data it receives from *rdt_send()* until it has at least one MSS worth of bytes. At that time it forms a segment that includes a header and MSS bytes of data; as a result, all segments sent, except possibly for the very last one, will have exactly MSS bytes of data. If less than N segments are outstanding (i.e., have not been ACKed), it transmits the newly formed segment *separately* to each of the receivers in a UDP packet. Otherwise, it buffers the segment and waits until the window has advanced to transmit it. Note that if $N = 1$, the protocol reduces to Stop-and-Wait.

A segment is considered to have been received once the sender has received ACKs from every receiver. The sender also maintains a timeout counter for the earliest transmitted segment. When all ACKs for this earliest segment are received, the sender cancels the timeout counter, advances its window, and sets another timeout counter for the currently earliest transmitted segment. If, however, the counter expires before ACKs from all receivers have been received, then the sender re-transmits the segment, *but only to those receivers from which it has not received an ACK yet.* This process repeats until all ACKs have been received (i.e., if there are r receivers, r ACKS, one from each receiver have arrived at the sender), at which time the sender proceeds to transmit the next segment.

The header of the segment contains three fields:

- a 32-bit sequence number,
- a 16-bit checksum of the data part, computed in the same way as the UDP checksum, and
- a 16-bit field that has the value `0101010101010101`, indicating that this is a data packet.

For this project, you may have the sequence numbers start at 0.

The client implements the sending side of the multicast ARQ protocol that will be described, including setting the timeout counter, processing ACK packets (discussed shortly), and retransmitting packets as necessary.

The P2MP-FTP Server (Receiver)

Each receiver listens on the well-known port `7735`. It implements the receive side of the multicast ARQ scheme. Specifically, when it receives a data packet, it computes the checksum and checks whether it is correct, and if so, it sends an ACK segment (using UDP) to the client. It then processes the received data depending on whether the received packet is in-sequence or out-of-sequence, as explained later. In-sequence data are written into a file whose name is provided in the command line. If the checksum is incorrect, the receiver does nothing, i.e., it discards the received packet and does not send an ACK.

The ACK segment consists of three fields and no data:

- the 32-bit sequence number that is being ACKed,
- a 16-bit field that is all zeroes, and
- a 16-bit field that has the value `1010101010101010`, indicating that this is an ACK packet.

The Multicast ARQ Scheme

The multicast ARQ scheme is a simplified version of the Selective Repeat scheme. At the sender, a packet is considered as *acknowledged* if ACKs from all r receivers have been received, and is considered *outstanding*, otherwise. For each transmitted packet, the sender keeps a list of r integers that are initially set to zero.

Whenever the sender receives and ACK from receiver $i, i = 1, \ldots, r$, for this packet, it increments the i-th integer in the list. Hence, the i-th integer indicates how many times receiver i has ACKed this packet.

The sender follows these rules:

- It maintains a transmission window of size N. A new packet is transmitted only if the number of outstanding packets is less than N; otherwise, the packet waits in the buffer for each turn.
- A timeout counter is maintained for the oldest outstanding packet only. If the counter expires, the oldest outstanding packet is retransmitted, but only to receivers that have not ACKed it, and the timer is reset.
- Once all ACKs for a packet have been received, the actions that the sender takes depends on whether the packet is the oldest outstanding packet or not. If the packet that has received ACKs from all receivers is not the oldest outstanding packet (i.e., there is an older packet that is still outstanding), then no action is taken. If the packet whose status changed from outstanding to acknowledged is the oldest outstanding packet (i.e., the one that defines the left end of the transmission window), then the sender: (1) cancels the timeout counter; (2) advances the transmission window (note that the window may advance by more than one packet if packets immediately following the oldest outstanding packet were already acknowledged); (3) sets a new timeout counter for the new oldest outstanding packet; and (4) transmits any buffered packets that were waiting for transmission, until either there are no buffered packets or the window becomes full.
- If the sender receives a second duplicate (i.e., third overall) ACK for a particular packet from the same receiver, it retransmits *the following packet* to all receivers that have not ACKed it yet. (Note that, unlike TCP, a receiver ACKs a received packet, not the one it expects.)

Each receiver follows these rules:

- ACKs are *cumulative*, i.e., the most recent in-sequence packet received is ACKed. As a result, duplicate ACKs for a packet may be sent.
- When a packet is received correctly, it is ACKed, but only if it is within the receive window; out-of-window packets are discarded and no further action is taken. If this is an out-of-sequence packet, it is buffered until any previous packets are received. If this is an in-sequence packet, it may also close gaps with other out-of-sequence packets received earlier. In this case, the just received packet along with any previously received packets that now become in-sequence, are written into the local file, and the window is advanced.

Generating Errors

Despite the fact that UDP is unreliable, the Internet does not in general lose packets. Therefore, we need a systematic way of generating lost packets so as to test that the multicast ARQ scheme works correctly (and to obtain performance measurements, as will be explained shortly).

To this end, you will implement a *probabilistic loss service* at the receiver. Specifically, the receiver will read the probability value $p, 0 < p < 1$ from the command line, representing the probability that a packet is lost. Upon receiving a data packet, and before executing the multicast ARQ protocol, the receiver generates a random number q in $(0, 1)$. If $q \leq p$, then this received packet is discarded and no other action is taken; otherwise, the packet is accepted and processed according to the multicast ARQ rules.

Offline Experiments

You will carry out a number of experiments to evaluate the effect of the number r of receivers, the window size N, MSS, and packet loss probability p on the total delay for transferring a file using the P2MP-FTP. To

this end, select a file that is approximately 10MB in size, and run the client and r servers on different hosts, such that the client is separated from the servers by several router hops. For instance, run the client on your laptop/desktop connected at home and the servers on EOS machines on campus; or the client machine (e.g., your laptop) could be located in the same room as the servers, but have the client connected to a wireless LAN while the servers are connected to the wired network. Record the size of the file transferred and the round-trip time (RTT) between client and servers (e.g., as reported by `traceroute`), and include these in your report.

Task 1: Effect of Window Size N
For this first task, set the MSS to 500 bytes, the loss probability $p = 0.05$ and the number of receivers $r = 3$. Run the P2MP-FTP protocol to transfer the file you selected, and vary the value of the window size $N = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$. For each value of N, transmit the file 5 times, time the data transfer (i.e., delay), and compute the average delay over the five transmissions. Plot the average delay against N and submit the plot with your report. Explain how the value of N affects the delay and the shape of the curve.

Task 2: Effect of the Receiver Set Size r
For this first task, set $N = 16$, the MSS to 500 bytes and the loss probability $p = 0.05$. Run the P2MP-FTP protocol to transfer the file you selected, and vary the number of receivers $r = 1, 2, 3, 4, 5$. For each value of r, transmit the file 5 times, time the data transfer (i.e., delay), and compute the average delay over the five transmissions. Plot the average delay against r and submit the plot with your report. Explain how the value of r affects the delay and the shape of the curve.

Task 3: Effect of MSS
In this experiment, let $N = 16$, the number of receivers $r = 3$ and the loss probability $p = 0.05$. Run the P2MP-FTP protocol to transfer the same file, and vary the MSS from 100 bytes to 1000 bytes in increments of 100 bytes. For each value of MSS, transmit the file 5 times, and compute the average delay over the five transmissions. Plot the average delay against the MSS value, and submit the plot with your report. Discuss the shape of the curve; are the results expected?

Task 4: Effect of Loss Probability p
For this task, set $N = 16$, the MSS to 500 bytes and the number of receivers $r = 3$. Run the P2MP-FTP protocol to transfer the same file, and vary the loss probability from $p = 0.01$ to $p = 0.10$ in increments of 0.01. For each value of p transmit the file 5 times, and compute the average delay over the five transfers. Plot the average delay against p, and submit the plot with your report. Discuss and explain the results and shape of the curve.

Submission and Deliverables
You must submit your report and source code, as explained below, using the submit facility by 11:59pm on the day due. There are several weeks until the due date for you to work on this project, therefore no late submissions will be accepted.

You will carry out Tasks 1-4 offline, and submit a report (PDF or Word file) with the results and your explanation/discussion of the findings. In particular, your report should include the file transfer delay curves for Tasks 1-4, your explanation of the behavior of the curves, and conclusions you may draw regarding the scalability of the P2MP-FTP protocol.

In order for us to test your program, you will also submit the source code (*no object files*!) separately. Name the file containing the source code `proj1`, with the appropriate extension (e.g., `proj1.c` if you code in C). We would like to ensure that the TA does not spend an inordinate amount of time compiling and running

your programs. Therefore, make sure to include a `makefile` with your submission, or a file with instructions on how to compile and run your code. Therefore, if you fail to include such a file, we will **subtract 5 points** from your project grade.

The code you submit must follow these specifications.

Command Line Arguments
Each P2MP-FTP server (receiver) must be invoked as follows:

`p2mpserver port# file-name` N p

where `port#` is the port number to which the server is listening (for this project, this port number is always `7735`), `file-name` is the name of the file where the data will be written, N is the window size, and p is the packet loss probability discussed above.

The P2MP-FTP client must be invoked as follows:

`p2mpclient server-1 ... server-r server-port# file-name` N `MSS`

where `server-`i is the host name where the i-th server (receiver) runs, `server-port#` is the port number of the server (i.e., `7735`), `file-name` is the name of the file to be transferred, N is the window size, and MSS is the maximum segment size.

Output
The code you submit must print the following to the standard output:

- P2MP-FTP server: whenever a packet with sequence number X is discarded by the probabilistic loss service, the server should print the following line:

  `Packet loss, sequence number =` X

- P2MP-FTP client: whenever a timeout occurs for a packet with sequence number Y, the client should print the following line:

  `Timeout, sequence number =` Y

Grading

P2MP-FTP server and client code (compiles and runs correctly)	60 Points
Task 1	10 Points
Task 2	10 Points
Task 3	10 Points
Task 4	10 Points
	100 Points