CS 352: Internet Technology, Summer 2016
Programming Assignment 2: Reliable Broadcast Protocol Using UDP

1 Introduction

In this assignment we will explore the mechanics of creating a reliable communication protocol on top of an inherently unreliable communication fabric. Your task is to create protocols for reliable communication from one machine to many machines. You have to write two programs that implement fast, reliable one-way communication from a single sender (the first program) running on one host to multiple receivers (the second program) running on different hosts in an unreliable environment.

2 Sender Program

The first program is a sender. The sender runs on one host and reads the names of large data files as input. The sender reads the data files, breaks the file contents into UDP datagrams and broadcasts the packets to multiple receivers running on the same subnet. The sender must be able to hold on to previously transmitted datagrams for retransmission as needed. Files get distributed one at a time (i.e the sender works on transmitting the contents of one file before starting to transmit the next file). We deliberately make communication between sender and receiver unreliable to simulate poor-quality links, highly overloaded links or congested routers in a large-scale private network. The receiver has to randomly drop a specified percentage of packets it receives to simulate an unreliable environment.

3 Receiver Program

The receiver program fakes unreliable communication due to poor quality links, overloaded links or congested routers in a large-scale private network by deliberately dropping some percentage of the incoming datagrams, thus acting as if the datagrams never arrived. A command-line argument specifies the percentage of packets lost. To simulate unreliable communication, the receiver has to randomly drop a specified percentage of incoming packets. All receivers will communicate with the sender on the same port. The sender and receivers will use an Automatic Repeat reQuest protocol of your own devising. The receiver sends ACK or NACK packets to the sender, and the sender responds to the ACK or NACK packets by sending more packets to the receiver(s). You have to design an efficient ACK/NACK protocol between the sender and receiver that minimizes ACK/NACK packet overhead and data packet retransmission. The receiver has to reconstruct the received data files on the remote hosts. The receivers should construct only complete files when possible.

4 Acknowledgment and Retransmission

The single sender is responsible for building packets and transmitting the packets to one or more receivers. The unreliability of the communication fabric used to carry packets from sender to receiver(s) means that the sender has to hold on to packets until all the receivers acknowledge receipt of individual packets. The sender must also know who its receivers are and which receivers have received individual packets. Thus, the sender must have the ability to receive some kind of acknowledgment packet from the receiver(s) that either acknowledge which packets have been received (a positive acknowledgment or ACK), or request packets that have not been received (a negative acknowledgment or NACK). The sender retransmits packets to receivers that have not received the requested packets. The sender can only discard a packet (and any associated reception status information) only when it knows that all receivers have actually received the packet.

Packets sent from the sender may not arrive at a receiver at all or they may arrive in a different order from the order they were sent. The possibility of packet retransmission means that a receiver might even get the same packet more than once. It is therefore necessary to carry some kind of packet sequencing information as metainformation communicated from sender to receiver. The receiver is responsible for receiving the packets, extracting application data from the packets in the correct order, requesting missing packets and ultimately building the proper output files.

The metainformation about packet sequencing and the communication of file name information and specification of where one file ends another file begins are things you have to design. This design is part of the *protocol* shared between sender and receiver(s). Part of your job is to design an application-layer protocol that carries sequencing data and metainformation from sender to receiver(s) as well as the design of ACK or NACK requests sent from receiver to sender.

The details of these kind of protocols as well as the design options available to you will be discussed in lecture.

5 Packet Resequencing

In distributed applications, some processes transmit packets (unicast or broadcast) and other processes receive the data packets, but the packets may not arrive in the same order they were sent, due to packets taking different routes from sender to receiver or due to some receivers losing packets that other receivers actually receive. The sending process will therefore embed a sequence number in the transmitted packets to allow the receivers to resequence potentially out-of-sequence packets.

The receiving processes must resequence the packets. The packets can be assigned integer sequence numbers as part of the application-layer protocol. The problem is the inherently finite size of any unsigned fixed point integer means that there is a maximum integer value after which any subsequent increment will overflow to zero. Any sort of packet resequencing algorithm should take inevitable overflow into account to run continuously.

A practical solution to the overflow problem is *Red-Blue Infinite Sequencing*. Here, we assign a ”color” to a sequence number. The sequence would start with red zero. The incremented value remains red until it overflows, when the red value becomes a blue zero. The blue value is then incremented and on overflow becomes a red zero again.

The red/blue numbers could be compared indefinitely according to the following axioms:

- `small blue > large red`

- `small red > large blue`
- `large red > small red`
- `large blue > small blue`

6 Timeouts

Due to unreliable communication between sender and receiver, the sender and receiver must take responsibility for determining when a transmitted packet has not reached it intended destination(s). A *timeout* mechanism allows a program to simultaneously do what it needs to do and wait a certain amount of time before taking an action.

Timeouts would be used in the following ways for this assignment:

1. `sender fails to receive acknowledgment from receiver`
2. `receiver fails to receive desired packet`
3. `sender decides that a receiver has become unresponsive (having crashed, shut down, etc)`

7 Program Start-up

The sender and receiver programs can be invoked in any order. The sender starts broadcasting packets immediately, even if there are no receivers. A receiver waits to receive a packet and then starts transmitting ACK/NACK packets back to the sender. The sender adds to its knowledge of who the receivers are from what receivers (identified by IP address) are sending ACK/NACK packets.

The sender will take a port number as a command line argument. The receiver will take a port number and a percentage (expressed as an integer between 0 and 100) as command line arguments.

8 Program Termination

When the sender has successfully transmitted all of its files to all receivers, it could send an "all finished" message to the receivers. The receivers would then shut down gracefully.

A receiver wishing to shut down before the sender has finished, can send a "leaving" message to the sender. The sender would then remove that receiver from its list of known receivers.

9 What to turn in

A tarred gzipped file named `pa2.tgz` that contains a directory called pa2 with the following files in it:

- A `readme.pdf` file documenting your design **paying particular attention to the thread synchronization requirements of your application**.
- A file called rbprotocol-testcases.txt that contains a thorough set of test cases for your code, including inputs and expected outputs.
- All source code including both implementation (.c) and interface(.h) files.
- A makefile for producing the executable program files.

Your grade will be based on:

- Correctness (how well your code is working, including avoidance of deadlocks.
- Efficiency (avoidance of recomputation of the same results).
- Good design (how well written your design document and code are, including modularity and comments).