Austin Miller

Tyler MacNeil

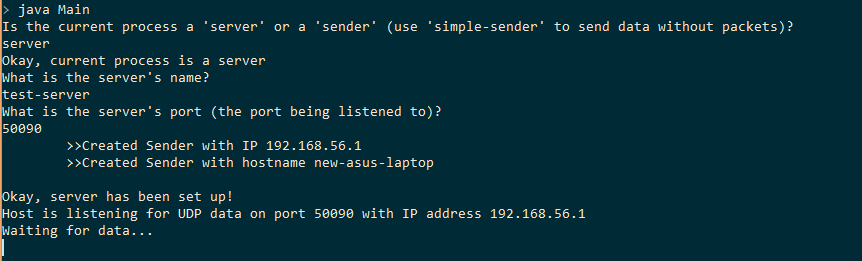
CS490: Networking

**Implementation Checkpoint: Instructions**

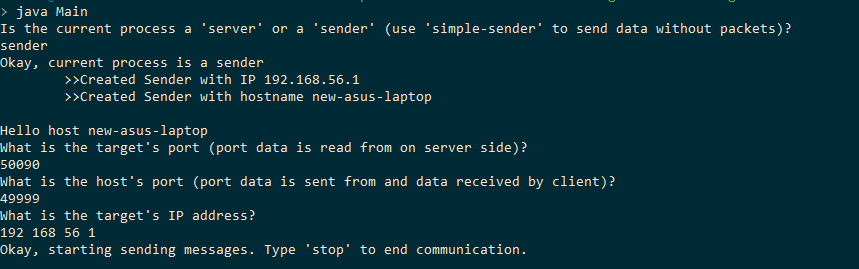
How to Run:

1. First, compile the project: javac \*.java
2. Then, you’ll want to run two instances of the project, one to be the ‘sender’ and one to be the ‘server’. So first, make the server:
   1. Run 'java Main’
   2. Input ‘server’, a server name, and the port that you’re machine will be listening to (for example, 50090). If you’re trying to send across the Internet and not between two machines on the same network, you’ll need to make sure that this port is forwarded in your modem/router settings before messages can be properly received.
   3. After inputting this, the server will be created using your IP (determined via the InetAddress class). The server is now waiting for input from any type of sender on the specified port.
3. Next, you’ll need to create a sender which will communicate with the server:
   1. Run ‘java Main’ in another terminal.
   2. Input ‘sender’, the target port (the port of the server from earlier), the host port that data is being sent out of, and the target’s IPv4 address (specifically, input the IP address numbers with spaces between each). If both sender and server are on the same computer, use ‘127 0 0 1’. If the sender and server are on different machines in the same network, then you’ll need to input the local IPv4 address of the server’s machine which is printed when the server is started. You can also run ipconfig in another terminal on the machine that is running the server to determine the local IP address of the server. If the machines are on different networks, info on which IP addresses to use is in the notes below.
4. Type anything in the sender’s command prompt, and it will be sent over to the server. Since the maximum transmission unit (MTU) is 128 bytes, any message with a size greater than that will be broken into several packets that will be sent individually. For messages that are broken-down like this, the server will recreate the full message by concatenating the app-data in each packet until the last packet is arrived. This full message is then printed out.

Examples:

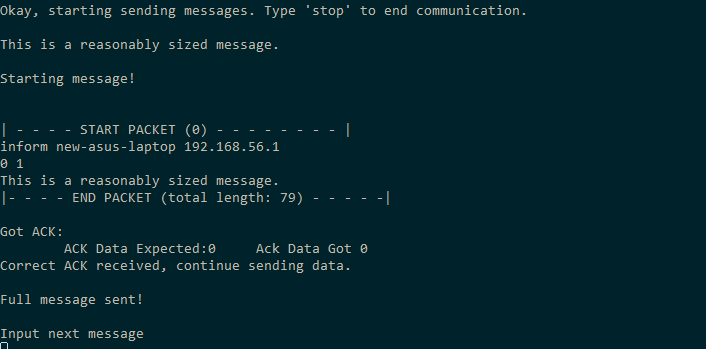
Initializing server with port 50090:

Initializing server with target port 50090 and the IP address specified above:

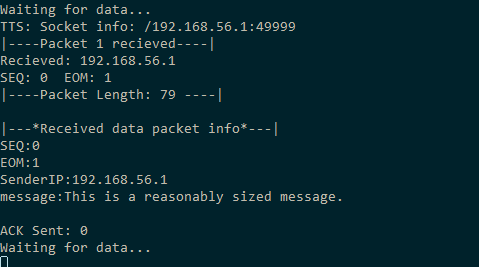


Sending short messages (messages smaller than MTU):

Sender sending message “This is a reasonably sized message”

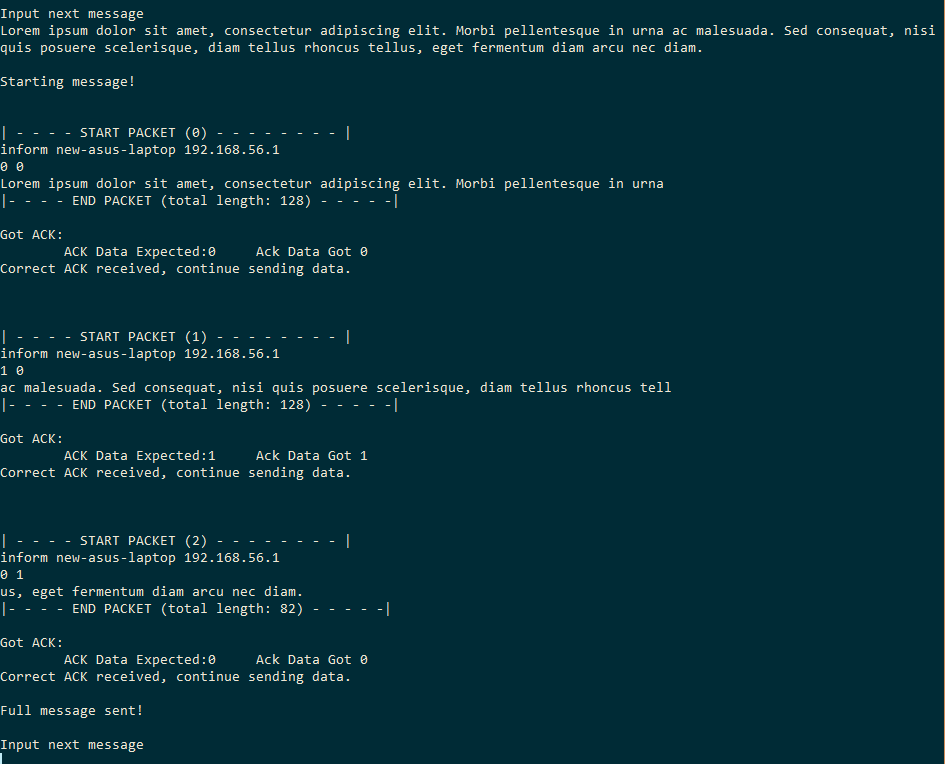


Server receiving this message:

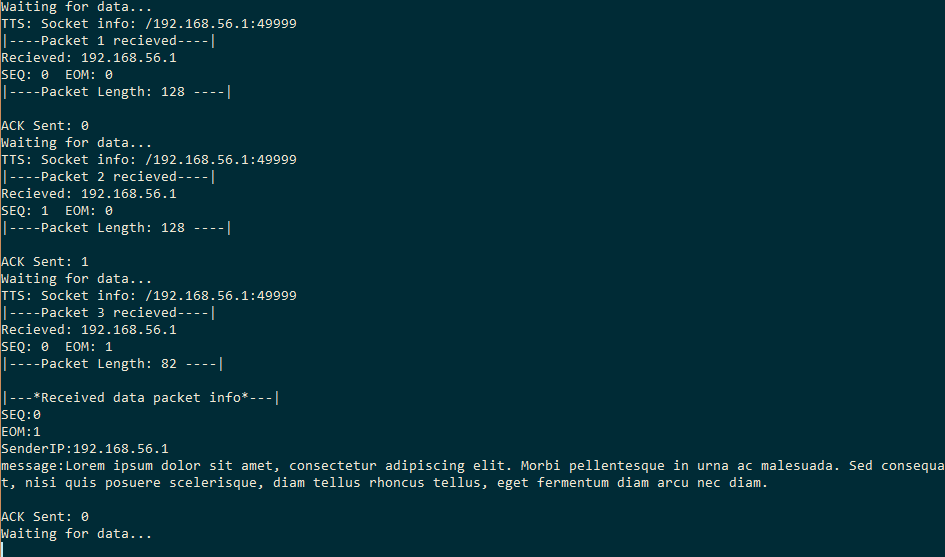


Long messages (messages larger than MTU):

Sender

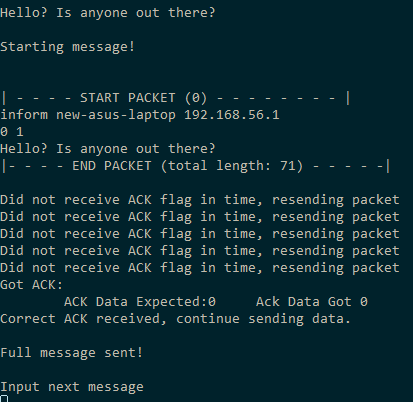


Server



Notes:

* Public IP or WAN Address
  + When using the client and server on different networks (ie. The server and sender are not on the same local network) the sender must use the Public IP address or WAN Address from where the server is from. To find this information either search “what is my IPv4 address” into google and click one of the links such as whatismyip.host or log into the modem of that network and locate the address from there.
  + It is important to note that this is only needed when the client and server are not on the same network. If they are on the same network, the IP address displayed from the server side when starting should be used.
* Port Forwarding
  + It may be necessary to forward a port to allow the connection to be made to the server. Instructions for this are difficult and vary based modem and firewalls used in the network.
  + Important is the the port only has to be forwarded on the server side.
* RDT
  + Corruption is handled by UDP automatically. Any packets that are sent and have been corrupted (whether it be from client to server or ACKS from server to client) are automatically dropped. This works well with the timers mentioned next.
  + All packets sent from client to server must be ACKed. After sending a packet, the sender starts a timer that resends the packet if an ACK has not been received in a certain amount of time. This handles the issues of packet loss and data corruption on packets sent from client to server or vice versa. This is because getting no ACK means that the sent packet was lost or corrupted (and dropped) OR that the ACK was lost or corrupted (and dropped). To see the timer in action, simple start up the sender first, send a message, then start the server:



* + All messages larger than the MTU are broken into separate packets. The server pulls the app-data from each packet and concatenates it all together until it gets the packet with the EOM flag, which specifies which packet is the last. It stops appending at this point, and has the full message.
  + Duplication of packets (caused by lost ACKs) on the server-side is handled via the SEQ flags of the packets. If the server ever receives a packet with the unexpected SEQ number, it simply sends the ACK it had already sent again without using the duplicated packets data (avoiding the duplication of data).
  + The last issue that was taken care of was the Sorcerer’s Apprentice Syndrome, where twice as many packets would needlessly be sent. This is solved in the code by having the client only accept ACKs that they expected, meaning delayed ACKs that are received after a message was resent are ignored.