CENG 435 - Term Project - Part 2

1st Hakan Sonmez
Department Engineering Computer Engineering
Middle East Technical University
Ankara, Turkey
hakan.sonmez@ceng.metu.edu.tr

2st Onur Tirtir
Department Engineering Computer Engineering
Middle East Technical University
Ankara, Turkey
onur.tirtir@ceng.metu.edu.tr

***Abstract*—The purpose of this work is to design and construct a reliable network with end systems and intermediate nodes. It is an extension of the previous implementation. The aim is to send messages on this network using Transmission Control Protocol (TCP) and a modified and improved version of User Datagram Protocol (UDP). We connect these different nodes containing different form of protocols with communication links. We then worked on conducting experiments on this network of multiple nodes measuring transfer time of a file.**

***Index Terms*—server, client, socket, router, broker, RDT, packet loss, broker, network topology, TCP, UDP, ACK, checksum, go-back-n**

I. Introduction

In this work, we aimed to measure file transfer time with respect to different *network* configurations. We performed our work on a network consisting of five nodes, each of which has a different functionality. Along these nodes, we utilized different transport layer protocols to serve an end-to-end communication between *source node* and *destination node*. The topology is same as the previous part. We also setup different network configurations on the links between these nodes to perform our experiments. We will explain our network topology and configurations briefly in following Section III.

We will describe our implementation in different aspects in Section III. We will give details about how we have established the connection between our nodes, protocols utilized from transport layer and run-time behavior performed by each node. We will also explain how we provided pipelining and multi-homing to our nodes for a better utilization of the overall network.

As we mentioned above, we set different *network* configurations in the form of reorder, packet loss and corrupt packages to measure transfer time of a file. We will describe and interpret results obtained from experiments in Section IV precisely.

After giving a brief conclusion on overall work in Section **??**, we will finalize report.

II. Design

In this section we have first given the overall topology of the network. We have summarized the basic differences between last part and this one. After that, we have given the problems of previous part's design and implementation. Then, we have explained our overall design of the project while explaining our solutions to the problems we have described.

The network topology is mostly same as the previous part. To explain briefly; There are five nodes in the network due to the given topology: Source Node (s), Broker Node (b), Two Router Nodes (r1 and r2) and Destination Node (d). These nodes are connected to each other with *communication links* as provided again in network topology. More specifically, Source node shares a link with Broker Node; Broker Node is connected to the routers and these routers are then finally connected to the Destination Node. The connection between Source node and Broker node is done via using TCP. Instead of using UDP directly we have extended and improved UDP on top of it.

In the first part of the project we have designed a mostly reliable network. Source node creates a random message with fixed length. It then sends this message to broker node. Broker, decides the path of the message. This is achieved by simply alternating the routers. The router nodes also had a script to route and forward the message in the application layer. Destination node gets the message from a router and sends a response in the form of an acknowledgement. Although, this process works fairly well, it had some problems regarding the reliability concerns.

A. Problems with the Previous Design

- The messages we were sending were being generated randomly on the fly. There was not a concept of file transfer. As a result we were not explicitly observing the reliability of the network. Even though our project in part 1 was satisfying the requirements and the specifications, it was not enough to observe and handle the overall behaviour per se. The messages were merely placeholders.
- The source node was sending the messages one by one. Process of sending the messages was: Source creates a random fixed length message. Source sends the message to the adjacent broker node using TCP. Message then travels through the network and reaches the destination. After destination receives the message, it forms an acknowledgement message, and sends it back to the network. Throughout this process, source is in a busy waiting state, waiting for the acknowledgement. This behaviour meant that there was no pipelining. As a result, sending multiple messages was slow, as there is blocking. Pipelining was absent.
- The broker did not have a routing table. The broker was alternating the messages to the routers one by one.

(broker sends to router1, sends to router2, sends to router1 etc.) This resulted in a network without multihoming.

- UDP is not a strictly reliable protocol. There is always the risk of losing packets. A lost packet needs to be handled as it can then result in a catastrophic failures if the message is crucial. In the previous task, we did not handle this case. There was not a timeout or re-sending. The source node did not even kept the messages sent because it was random.
- Similar to the previous point, UDP also does not guarantee the order of the packets. A packet can arrive before a previously sent packet. This was not a concern of ours as we were not sending the packets before getting the acknowledgement of the previous one. However, if we want to have a generalized pipelined network, this needs to be handled.
- Also similar to the previous two points, UDP does not guarantee the corruption in the packets. Corruption of the packets is a universal problem regardless of the protocol. Even though the checksum idea is used in TCP and UDP, it still does not mean that the packet sent and the received will include the same content. The bits may flip or swap between them. This corruption can happen in any of the layers. This can happen even in the most perfectly reliable checksum methods as it is possible in a random chance the packet will change and have the checksum. Our previous design did not consider any corruption case.

In the first part we have discussed and decided source node should behave as a client since it is the one generating and sending the data and does not need to be open at all times. We have also mentioned that it is only supposed to connect to broker node and this connection is done by *TCP*. It first reads the data from the supplied file. It then processes this reading and creates the packet. The constructed message size is fixed throughout the topology. Every node in the topology receives and sends the packets of this size. However, we have decided not to use that approach because of the complexity and the unexpected results. This note utilizes TCP and it is fairly reliable, so we have not used a complex ack mechanism. In order to prevent reordering and packet corruption we have included sequence numbers and checksums. After reading the message, source node calculates and adds these features to the packet as a header. This process will be explained in detail in the following sections. We have considered and experimented with doing this numbering and checksumming in the broker node since the unreliable transfer starts there but we decided to design it like this because of consistency. We have also discussed another topic about headers, where we would first send the header containing sequence number, checksum and the main packet's size. However, as predicted we would have to add separate checksums to both header and packet; so we have aborted that approach. This fixed length packet with the header containing necessary meta information is then sent to broker to be distributed to the whole network. Since the source does not wait for the acks of these messages, it does not block the whole system unlike the first part. This results in a pipelined network which we will explain in the following sections.

Same as the first part of the project broker node is basically the main component of this network. It has three connections: one TCP connection to source node and two UDP connections to both routers. Also we have used used this node as a server because the source node is a client. There are multiple connections and the specification explains that there must be one script in each node; so, we have used a multi-threaded approach similar to the first part. This allows us to have multiple connection operations made simultaneously in this node. While these messages stored, the broker sends these to the routers. We have first considered another approach, where the source would send all the data to the broker. Broker then would store all of this. After all data is received the broker would start sending it. We have then concluded that it is not a feasible scenario, as it would take a long time, and we do not want any component to block the whole system. Specifications also state that the network should be multi-homed. We achieve this by doing sending and receiving simultaneous. Our design sends data to both routers simultaneously.

In the previous part and in this one, *Router Nodes'* purpose is to make the intermediate connection between Broker node and the Destination node. Routers either get the message from the Broker node and redirect it to the Destination node or get *Acknowledgement* (ACK) from Destination node and transmit it to the Broker. Both of these connections are made with UDP. In this part, we have not designed a routing or forward mechanism in the application layer as we handled it in the network layer as explained in the Section III.

Destination Node is the node that the message finally reaches. Logically and due to the specifications, Destination node works like a server. So it has to always wait for a message. After receiving the message, it sends an Acknowledgment (ACK) message back to the link it received the message. This acknowledgement includes a sequence number and a checksum. Our packet size is same as the packet size. This ACK message is sent via UDP to the router.

In this section we have explained the overall design of the network. However, on top of that design we have added Reliable Data Transfer. We will now explain how we used those RDT concepts regarding our project. We have considered two methods for this task: Selective and Go-Back-n. We have decided to use Go-Back-n because of the ease of implementation and observation of results. Our design does not use this method though. We use a modified and combined version of Go-Back-n and RDT 3.0. The sender and receiver in Go-Back-n is broker and destination in our context.

In broker node (i.e Sender in Go-Back-n method) Go-Back-n allows us to send the messages in a pipelined way. Basically, we bulk send the messages from broker, if the destination sends the expected ack, we send more. The broker has a window of buffered packets. This window shifts while it gets the needed acknowledgement. We know the expected number of ack because of the sequence numbers that are assigned in

the source node. Sequence numbers are consecutive and the broker waits an ack with the header containing an ack number equal to the sequence number of the sent message. One of the crucial design point is the inclusion of the packet timeout. Since we used Go-back-n if there is a case of a packet loss or a reorder, all of the packets in the window get sent again. This re-sending happens if the expected acknowledgement does not come in a pre-defined timeout. If the broker does not get the ack in time, it again bulk-sends these messages and the cycle goes on.

Destination node (i.e Receiver in Go-Back-n method) does not keep a window or a timeout. It only sends an acknowledgement if the received packet is the expected one. If not however, it sends the ack of the most recently received valid packet. As a result it is in a constant waiting state. Also note that we check if the packet is corrupted in both broker and destination. This allows us to control if the packet is corrupted or not.

III. Implementation

We set up three different scripts for three different nodes (source, broker, destination) in our topology as described in Section III. All of these scripts execute in the remote servers provided. As it is explained in the Section III the data flow along the topology, we have two different partitions. First partition utilizes TCP in its underlying transport layer and second partition utilizes UDP. The TCP connections we have established are persistent. However, Since UDP is a *connectionless* protocol a signal goes out automatically without determining whether the receiver is ready, or even whether a receiver exists. After completing these implementations we have configured the the nodes with UDP connections to simulate a real life-like behaviour of reordering, packet loss and corruption.

Source node is the one creating the packets. We first read from the file. This is our payload. We have tried different payload sizes and decided to use 500 bytes as our protocol's convention. Source node then adds a header to this payload. Header has two fields: sequence number as discussed in Section . Header also contains a checksum of the sequence number and the payload. At first, we have manually calculated UDP checksum. It gave us a 2 byte value. However, as it is not a very reliable method, there was some flipped bits at the end. We have then decided to use md5 as an alternative since it is a more reliable hash. So, length of checksum is 32 bytes and length of sequence number is 6 bytes. Note that these numbers are open to change and improvement. In another configuration, changing these values will not result in unexpected behaviour. After sending this systematically constructed message, it reads from the file and sends again. This process goes on up until all of the messages are constructed and sent to broker. We do not send an ack message to source as according to specifications it is not necessary in this part of the project. Since we have to measure the file transfer time between source node and destination node, we get and save the time-stamp of the exact time right after the source sends its data.

.

Broker node does act as a server. It has three interfaces, one for connecting to *source node* and the other two are for connecting to destination - node at application layer. A *TCP* socket is utilized via first interface to accept connection from *source node* and two *UDP* sockets are utilized on other two interfaces to communicate with destination. Note that this communication to destination happens in the application layer. The route of the messages changes in the network layer. We explain it in detail below. In order to implement this concurrent data communication behaviour we have five worker threads: One for receiving from source, two for sending to destination and two for receiving from destination. The flow of the communication is done via multihoming and concurrency with Go-Back-n as our primary method. The source thread gets the data via TCP non-stop. This data gets stored in a global buffer. Then the UDP sender threads get the next eligible packet from the buffer. We have used mutex mechanisms to avoid race conditions. The threads send this packet to the next hop. Every thread has its own socket, so there is no data flow errors. While this sending process is continuing, UDP receiver threads wait for acks of previously sent packets. Hence, we have a very concise and well-defined process. We have explained our methodology of Go-Back-n in Section . As explained above, we have used a fixed length window. We have experimented with the window size during the development process. If the window would be small, then we could not utilize the pipelining well, on the other hand, if the window size would be large, then the re-sending would take more time than desired. Thus our protocol's window size is 16 packets for convention. To prevent packet loss, after sending a packet, we wait for a time. If the ack did not arrive in that time, we send all of the packets in the window. We do this in a new thread to not block other threads. To determine the timeout value, we have conducted some experiments; we have measured Round Trip Time several times to have a lower bound of sorts. We have finally decided to make timeout bounded to 0.04 seconds. This exact numbers are subject to change and can be redefined easily for any other topology or configuration. Broker also, does a corruption check while receiving from destination. This check allows us to not have any corrupt packets.

There is not a script running in router nodes. They are merely a gateway from broker to destination. Both of these nodes behave exactly like each other with the minor difference being the related interfaces. Broker sends the message to destination in the application layer. However, applying the *route* command we achieve the desired situation. This allows us to route the packets to the router nodes. We have run route commands in both broker and in destination. This added two routes to the routing table in both broker and destination.

Destination Node has two interfaces. They both use UDP connections to broker nodes in the application layer. Similar to broker node, we have also adjusted the routes to send it via routers as a gateway. Since there are two connections this script has two worker threads. Destination node behaves like a server due to the specifications. So, this node initially begins

by trying to receive data from either of the interfaces with both of its threads. Note that in the previous task if broker sends data to any of the destination interfaces while this node isn't waiting to receive, the data would be lost. However, this is prevented with go-back-n. Initially broker threads receive data and if the packets have expected sequence number and in good condition (i.e not corrupt) it produces an acknowledgement. The size of the acknowledgement in our protocol is same as packet size. If the expected packet did not arrive (either is lost or came in wrong order) destination rejects this message until it gets the right one.

IV. Experimental Results

We have conducted three different experiments as stated in specifications. These experiments are: packet loss, corruption and reordering versus file transfer time. Before doing any experiments, we have first measured the file transfer times for a basic configuration with no properties about 30 times. We have used it as a baseline and a comparison point for all of the experiments. For each of these experiments, we have performed three different link layer configurations. These configurations are exactly same as the specifications. For every bexperiment we have sent the given file 30 times and then took mean of time-stamps for these 30 readings. We also have drawn error bars for each *file transfer time/*. To draw them, we focused on the confidence interval of %95. In all of the below figures, we have four data points. 0 -representing no configuration.

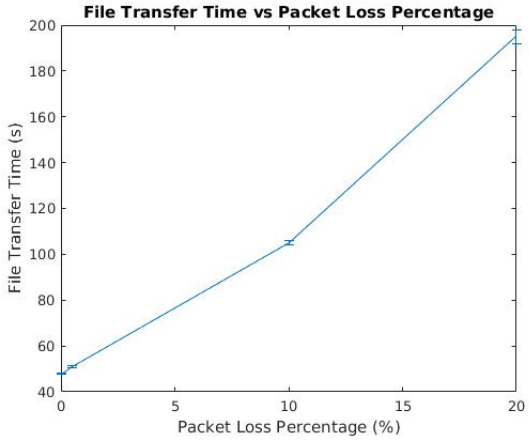

Fig. 1. Packet Loss vs File Transfer Time

In figure 1, we have measured file transfer time with %0.5, %10 and %20 packet loss respectively. As expected, the increase in the loss resulted in an increase of the FTT. In our implementation, if the expected packet does not arrive to destination node, it will not send an ack to that packet. As a result, broker's timeout condition will trigger, and all of the information will be resent. This results in a nearly linear graph.

In figure 2, we have measured file transfer time with %0.2, %10 and %20 packet corruption respectively. Similar to packet

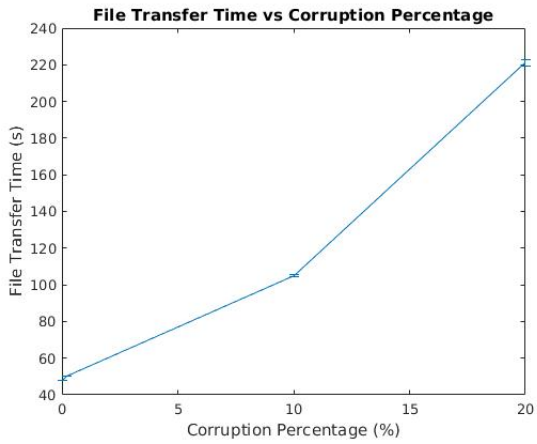

Fig. 2. Corruption vs File Transfer Time

loss, the increase in the corruption resulted in an increase of the FTT. In our implementation, if the destination calculates a checksum and it is not the same as the header of the packet, it will not acknowledge the packet. As a result, broker's timeout condition will again trigger, and all of the messages will be resent. This results in a nearly linear graph similar to packet loss.

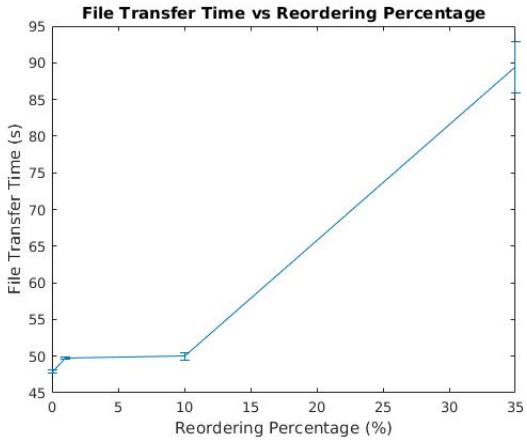

Fig. 3. Reordering vs File Transfer Time

In figure 3, we have measured file transfer time with %14, %10 and %35 packet reorder respectively. Similar to other two configurations, the increase in the percentage of reordered packets resulted in an increase of the FTT. In our implementation, if the destination gets a packet with sequence number more than the number it expects, it will ignore that packet. As a result, broker's timeout condition will again trigger, and all of the messages will be resent. This results in a mostly linear graph after a threshold. Note that in relatively low percentage reordering, the file transfer time does not generally increase. This is a result of our implementation, as the resending does not happen as often, and if happens, it takes place in a very short amount of time.

To sum up, we have generally observed that, if the network does not behave as expected the amount of time will increase gradually. This increase is usually linear.

V. Conclusion

We set up a topology with five nodes, each having a different job. The aim of this work was to observe the behaviour of transport layer protocols, their effects on the file transfer time. We utilized *tc/netem* to enrich our observations. Since we have implemented the first part very modular, we did not have a hard time implementing on top of it. As a group we have done every part together. The most time consuming part of this work for us to do the experiments, since we have done a huge amount of them.