CENG435 Data Communications and Networking Term Project Part 2

Misranur YAZGAN
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
misranur.yazgan@ceng.metu.edu.tr

Elif KAYA
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
elif.kaya@ceng.metu.edu.tr

***Abstract*—This paper explains and analyzes an application of the socket programming with the purpose of transmitting a large file from a source to a destination over a network which has a number of intermediate nodes namely Source, Broker, Routers and Destination. In this project, Transmission Control Protocol(TCP) is used between Source and Broker whereas between Broker and Destination, the unreliable transport layer service provided by the User Datagram Protocol(UDP) is transformed into a reliable service by constructing Reliable Data Transfer(RDT) protocol on top of it. With the aim of testing and observing how our RDT implementation behaves on situations such as packet loss, reordering of the packets and packet corruption, we conducted several experiments by introducing these factors on the links between the nodes. To serve this purpose, we exploited netem/tc Linux commands which provides the network traffic control facilities for experimental purposes.**

***Index Terms*—Network Socket Programming, Client-Server Architecture, Transport Layer Protocols, Reliable Data Transfer Protocol, Network Configuration**

I. Introduction

This paper aims to highlight the behaviors of a network which implements reliable data transfer protocol when packet loss, packet corruption and packet reordering is introduced into the links along with designing an overall network structure by implementing a UDP-based "Reliable Data Transfer" (RDT) protocol which is able to support pipelining and multi-homing.

The project concentrates on the implementation of a reliable network which is able to transmit a large file by dividing into chunks. The network connects two edges, namely Source which is the host s and Destination which is the host d, and one broker over disjoint paths on the specified topology. Network configuration is changed by modifying the routing table for the continuum of the paths. There are mainly five nodes on the specified topology which are listed as the following:

- Source
- Broker
- Router1
- Router2
- Destination

The communication between the Source node and the Broker node is established through TCP whereas the communication between the Broker, and the Destination nodes is provided through UDP based Reliable Data Transfer protocol. In routers, the router logic is implemented by manipulating the routing tables of the nodes.

II. Background Information

A. UDP Based Reliable Data Transfer Protocol

RDT protocol ensures that all the packets that are sent from the source will eventually be delivered to the destination and this delivery will preserve the order of the packets. With the help of checksum, sequence number, acknowledgements(ACKs), retransmissions and timer, the goal of the reliable data transfer protocol is guaranteed to take place.

Since UDP does not provide a reliable data transmission due to the fact that it does not employ any handshake and it directly sends the data to the given address regardless of whether the server is alive or not, implementing an RDT over a UDP guarantees the reliability and order of the messages.

B. Importance of the Broker Node

Broker is the node where the packets coming from the Source are received through TCP sockets and forwarded to the next hop addresses through the UDP based RDT. Broker has the ability to send packets through multiple links at the same time which enables us to send large files in a shorter time. In this way, by exploiting multiple links at the same time, we make sure that the network is multi-homed. In our project, we implement this ability in a way that the Broker forwards the packets to two different routers alternatingly. The Broker plays a significant role in the overall network structure since it both acts like a server by listening to the Source node and acts like a router by implementing the store-and-forward logic.

III. Design and Implementation Approach

A. Checksum Functionality

In order to know whether a message is delivered correctly without any corruption, we exploited checksum functionality.

As a beginning step, we have designated our own "input.txt" file containing just ASCII characters. With this file, we observed that the delivery was reliable and ordered. However, while conducting the experiments with the provided input file, we have faced an issue due to the implementation of our own checksum. The checksum function was designed as the following:

$sum = (sum >> 16) + (sum \ \& \ 0xffff)$
$sum+ = (sum >> 16)$
$result = (\sim sum) \ \& \ 0xffff$
$result = result >> 8 | ((result \ \& \ 0xff) << 8)$
$return \ chr(result/256) + chr(result \ \% \ 256)$

However, the function was not compatible with non-ASCII characters. Therefore, to overcome this issue we decided to change the implementation by exploiting the existing hash libraries for the experiments we conducted by using the provided large input file. With the help of md5 from the hashlib library, checksum functionality is implemented as the following:

$m = hashlib.md5(message)$
$return \ m.hexdigest()$

In addition, since this md5 function provides hashes which are 32 bytes long, we have reduced the possibility of not being aware of corrupted packets. Furthermore, when we add them to the beginning of the message as the header, we did not have to deal with padding the checksum in order to create a fixed length field in the header.

B. Acknowledgement Packets, Timer and Retransmission Mechanism

When the destination node receives a packet that is not corrupted, it sends an acknowledgement packet to the source of the RDT protocol(Broker in our case). In this way, the source can know whether or not the packet it sent is received by the destination node correctly.

The destination node performs the following operations in order to check if a packet is corrupted or not so that it can send an ACK message:

1) Parse the header of the received packet and obtain checksum field.
2) Parse the packet and obtain the original message.
3) Calculate checksum value of the received message.
4) Compare these two checksum values with each other.

So, when it compares these checksum values, it can know whether a message coming from the source is corrupted or not. If the values does not match, it can infer that the message is corrupted, so it does not send any acknowledgement. If the values match, it can infer that the message is not corrupted, so it sends an acknowledgement packet, which is preserved with the checksum functionality again, with the packet number corresponding to the received packet number to the source node.

When the source receives this ACK packet, it knows that the message it sent has been received correctly. In addition to this mechanism, the source keeps a timer for the messages it sent. When it sends a message, it starts a timer which will be expired after a certain amount of time has passed. We decided this amount of time to be 50 milliseconds which exceeds the time it takes for one RTT. If within this time the source receives an ACK from destination, it does not perform any retransmissions. However, if an ACK is not received within this time, the source retransmits the packet assuming that the packet has not been received correctly or has not been received at all due to packet loss or packet corruption.

In our implementation an acknowledgement packet is sent for each received packet instead of using cumulative ACKs.

C. Packet Numbers

In order to ensure that the information about the order of the messages is not lost, the source includes packet number to the header of a message when packetizing it. In this way, even if the packets does not arrive in order, the destination is able to merge the packets by ordering them according to their packet numbers included in their headers.

D. Message Structure

As a design choice, we have decided to divide our large message into 600 bytes chunks. Then, when we send the messages with UDT based RDT protocol, we add packet number of the related chunk in order to preserve the order of the messages. After that, we calculate and include the checksum value of this message in the header so that when the destination receives this packet, it can unpack the packet and compare the checksum value in the header with the one that it computes.

Checksum field consists of 32 bytes whereas packet number field consists of 4 bytes. In this way, our message(header + payload) becomes 636 bytes which is less than the upper limit specified for the packet size(1000 bytes).

E. Pipelining

Pipelining enables us to send packets without waiting acknowledgements rather than performing in a stop-and-wait manner. In this way, we can send large files in a shorter time period. For the pipelining part, we decided to choose our window size as 10, meaning that there can be at most 10 not acknowledged packets in the pipeline. In our implementation, the sender's window slides only when the ACK for the expected packet in the line is received. However, in the meantime if destination sends an ACK for the other packets in the window that are received, the ACKs are stored on the source side.

F. GENI Platform

We have used the virtual networks on GENI Platform and created a slice using the aggregate University of Kansas InstaGENI. We have downloaded and used the topology file in the XML format as it is given in the project requirements. The topology has created the nodes that are mentioned in the Introduction part. Then, we created and downloaded an SSH Key. After these steps, we connected to the University of Kansas InstaGENI through ssh with the following commands for all nodes:

```
ssh -i <private key location>
<username>@<hostname> -p <portnumber>
```

We have used Secure File Transfer Protocol(SFTP) to get the files from our local environments with the following command:

```
sftp -P 8085
<username>@external.ceng.metu.edu.tr
```

And we have used get and put commands when needed.

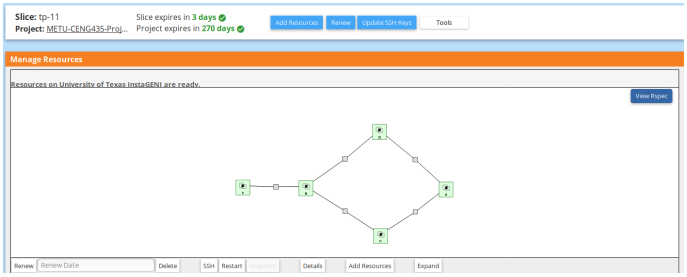


Fig. 1. GENI Platform Resources View

G. Network Configuration

For this part of the project, we were not allowed to run any scripts on the router nodes. Instead, we were expected to manipulate the routing tables of the routers by adding destination addresses. For this purpose, we have used *route* command of the Linux by exploiting the IPV4 routing mechanism implemented in the kernel.

- For Router 1:
  For the packets coming from Broker should be forwarded to Destination, the following command is given:

  ```
  ~$sudo route add -net 10.10.3.2 netmask
  255.255.255.255 gw 10.10.3.2
  ```

  For the packets coming from Destination should be forwarded to Broker, the following command is given:

  ```
  ~$sudo route add -net 10.10.2.1 netmask
  255.255.255.255 gw 10.10.2.1
  ```

- For Router 2:
  For the packets coming from Broker should be forwarded to Destination, the following command is given:

  ```
  ~$sudo route add -net 10.10.5.2 netmask
  255.255.255.255 gw 10.10.5.2
  ```

  For the packets coming from Destination should be forwarded to Broker, the following command is given:

  ```
  ~$sudo route add -net 10.10.4.1 netmask
  255.255.255.255 gw 10.10.4.1
  ```

H. Source Node

We have designed s node as a source device which employs Transmission Control Protocol (TCP) based socket application. In s.py we have implemented this node based on our design.

Definitions of the functions in s.py:

- readFile(filename):
  It opens the file specifies with the name specifies in the main.
  Then it reads chunk sized messages and appends to the chunk list.

- sendMsgToB:
  It sends message to the Broker.
  It send the chunk with the packet_number index from chunks list to the Broker until the end of the chunks list.

- connection(): It creates a TCP socket and connects it to the Broker's address. Broker's address was found from the GENI slice. We have used threads targeted sendMsgToB with the argument socket that is created in connection function.
  For the experiments we have print out the time in this function as start time of the overall process.

I. Broker Node

We have designed b node as a broker device which employs both UDP based RDT (between destination and broker) and Transmission Control Protocol (TCP)(between source and broker) based socket applications. In b.py we have implemented this node based on our design.

Definitions of the functions in b.py:

- checksum(message):
  Since the aim is implementing an RDT, and checksum is one of the essential part of an RDT we have implemented a checksum function that computes a hash value to append header. It uses hashlib's md5 and hexdigest functions because of the reasons mentioned in the Message Structure part.

- isCorrupted(message, checksum_header):
  The function checks if the checksum value in the header matches the checksum value computed by the Broker node. Returns True if they are not matched.

- sendMsgToRouters(sockbr1, sockbr2):
  The function sends the packetized messages received from s to the next hop address. It checks the previous address, if it is r1 then it sends the current message to the r2, else it sends the current message to the r1.
  The windowsize(10) , base, and timeout is implemented in this function.
  Waits reasonable amount of time (which is designed is 0.05 in our implementtaion) for ACK.
  Retransmits if ACK does not exists.

- receiveMsgFromS(client):
  It receives the messages from Source Node by using TCP. If there exists a received_message it appends the message to the send_buffer list.

- receiveMsgFromRouters(sock):
  It receives messages from routers by using UDP. In a while loop, function is marks the related index of the received_acks by checking if there exists a message and it isCorrupted function returns False(means messages are matching), we assign 1 to the received_acks[pack_num].

- packetizeMsg(packet_number, message):
  Adds sequence number(packet number) and checksum of the packet to the RDT header. It does padding for the packet_number by turning it to a string and adding zeros if needed.

- connection():
  Function creates all the related sockets for both receiving and sending messsages (between Destination and Broker, between Broker and Routers). After creating sockets, functions starts threads for sending and receiving messages by targeting related functions.

J. Destination Node

We have designed d node as a destination device which employs User Datagram Procotol (UDP) based socket application.In d.py we have implemented the node based on our design.

Definitions of the functions in d.py:

- checksum(message):
  Since the aim is implementing an RDT and checksum is one of the essential part of an RDT we have implemented a checksum function that computes a hash value to append header , like we did in Source node,. It uses hashlib's md5 and hexdigest functions because of the reasons mentioned in the Message Structure part.

- isCorrupted(message, checksum_header):
  The function checks if the checksum value in the header matches the checksum value computed by the Broker node as it does in Source node. Returns True if they are not matched.

- createACK(packet_number): Function packetize and creates an ACK message for the related message. It adds packet_number, checksum, and message. It does padding for the packet_number by turning it to a string and adding zeros if needed.

- receiveMsgFromRouters(sock): Function receives messages from r1 and r2, and buffers message to chunks[packet_number] to a list. After buffering them it appends them a text string for the comparison of input and the arriving message. Ending time for the experiments are also printed in this function.

- sendMsgToRouters(sockdr1, sockdr2): Function sends the ACK message from the Destination to the routers. It uses routers as it is like in Broker(checking the previous hop). If a message is received and it is not corrupted, it creates ACK in createACK function.

- connection(): Function creates all the related sockets for both receiving and sending messsages (between Destination and Routers). After creating sockets, functions starts threads for sending and receiving messages by targeting related functions.

IV. Experiments and Results

For conducting the required experiments for the project, our objective was to observe how the file transmission time changes according to the network traffic conditions. To serve this purpose, we manipulated the links by introducing them packet loss, packet reordering and packet corruption. We observed how our RDT protocol behaves corresponding to these situations and how the total time for the file transmission changes. In order to configure links, we have exploited several netem/tc Linux commands which provides facilities to control the network traffic such as introducing delay, packet loss and other characteristics to the packets.

A. Configuration of the Links

In our project, we needed to manipulate the traffic by introducing packet loss, packet reordering and packet corruption on the links between the Broker node and the Destination node. These links are specified as the following:

1) The link between the Broker and the Router1
2) The link between the Broker and the Router2
3) The link between Router1 and the Destination
4) The link between Router2 and the Destination

In order to learn which links we need to configure in the nodes, we checked the relevant interfaces and IP addresses of the nodes that we have reserved on the Geni Portal. After that, we typed the following command on the Broker, Router1 and Router2 nodes:

```
ifconfig
```

In this way, we learned which links to configure on the nodes for changing the network emulation delay. The corresponding names of the links shown above in our resource nodes are given as the following:

1) eth2 in Broker
2) eth3 in Broker
3) eth1 in Router1
4) eth2 in Router2

We performed 3 different experiments by introducing packet loss, packet corruption and packet reordering to the links mentioned above. For each part, we used 3 different configurations.

For the experiment about packet loss, we used the following configurations:

1) 0.5% packet loss, $3ms$ delay

2) 10% packet loss, $3ms$ delay
3) 20% packet loss, $3ms$ delay

For the experiment about packet corruption, we used the following configurations:

1) 0.2% packet corruption, $3ms$ delay
2) 10% packet corruption, $3ms$ delay
3) 20% packet corruption, $3ms$ delay

For the experiment about packet reordering, we used the following configurations:

1) 1% packet reordering, $3ms$ delay
2) 10% packet reordering, $3ms$ delay
3) 35% packet reordering, $3ms$ delay

To perform the configurations specified above, we used the following command from netem/tc library: tc qdisc change dev [INTERFACE] root netem loss 0.5

```
sudo tc qdisc <add/change> dev <interface>
root netem loss <loss percentage> corrupt
<corruption percentage> duplicate 0\%
delay 3ms reorder <reorder percentage> 50\%
```

where

- $< add/change >$: If the configurations are introduced to the link for the first time add command is used, otherwise $change$ is used.
- $< interface >$: The name of the link we learned with the command $ifconfig$, e.g. eth2, eth3.
- $< losspercentage >$: One of 0.5%, 10% or 20% for the related experiment.
- $< corruptionpercentage >$: One of 0.2%, 10% or 20% for the related experiment.
- $< reorderpercentage >$: One of 1%, 10% or 35% for the related experiment.

In order to achieve the proper results, we conducted each of the experiments many times and took the mean of the results to calculate the file transmission time.

In order to calculate the file transmission time, we kept track of the time passed from the first packet is sent from the source until the last packet is received by the destination.

B. The correlation between the File Transmission Time and Packet Loss Percentage

The following graph in Figure 2 shows the correlation between the file transmission time and the packet loss percentage.

The first, second and the third bars represent the experiments we conducted with applying 0.5%, 10% and 20% packet loss respectively. The horizontal axis represents the packet loss percentage whereas the vertical axis represents the file transmission time with 95% confidence interval in milliseconds. We can infer from the graph that when the packet loss percentage is increased, the file transmission time increases. This stems from the fact that in case of packet loss, broker retransmits the not acknowledged packets after it waits for a time span. Since the number of total transmissions increase, the total time it takes to transmit the whole file also increases.

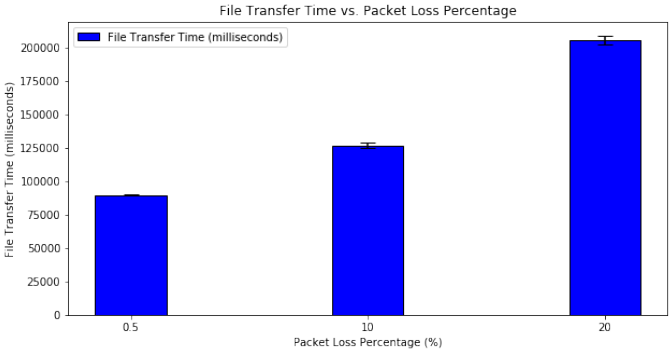
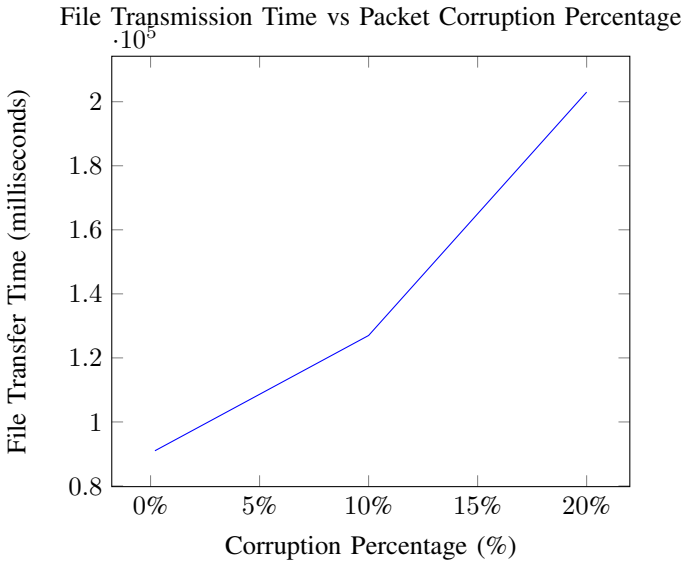


Fig. 2. The correlation between the file transmission time and the packet loss percentage

C. The correlation between the File Transmission Time and Packet Corruption Percentage

The following graph shows the correlation between the file transmission time and the packet corruption percentage.



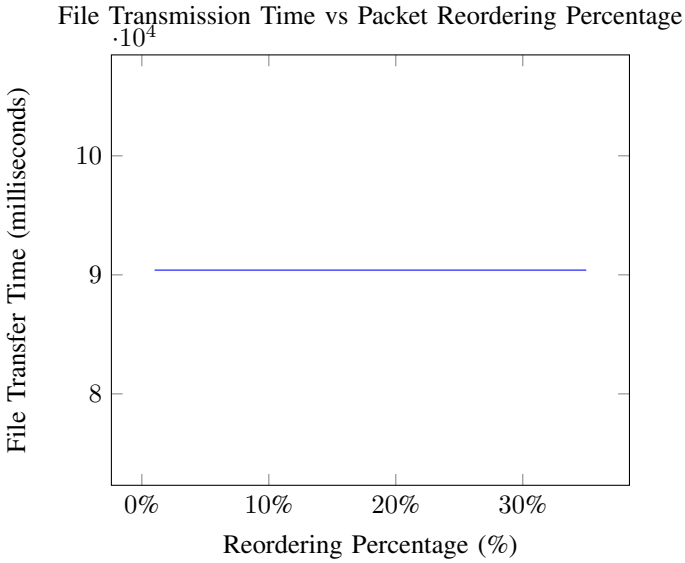
The horizontal axis represents the packet corruption percentage whereas the vertical axis represents the file transmission time in milliseconds. We can infer from the graph that when the packet corruption percentage is increased, the file transmission time increases. This is due to the fact that in case of packet corruption, destination does not send acknowledgement for that packet. So, broker retransmits the not acknowledged packets after it waits for a time period. Since the number of total transmissions increase, the total time it takes to transmit the whole file also increases.

The error ranges for the file transmission time with 95% confidence interval is given as the following:

- For 0.2% packet corruption, 63.6
- For 10% packet corruption, 412
- For 20% packet corruption, 4370

D. The correlation between the File Transmission Time and Packet Reordering Percentage

The following graph shows the correlation between the file transmission time and the packet corruption percentage.



The horizontal axis represents the packet corruption percentage whereas the vertical axis represents the file transmission time in milliseconds. We can understand from the graph that when the packet reordering percentage is increased, the file transmission time is not affected so much. This is because of the fact that in case of reordering, the packets are not transmitted again. They are only transmitted in a different order. The destination node is able to preserve the order of the packets by examining the packet numbers placed in their headers.

The error ranges for the file transmission time with 95% confidence interval is given as the following:

- For 1% packet reordering, 9.99
- For 10% packet reordering, 0
- For 35% packet reordering, 4.8

V. Conclusion

This paper introduced and analyzed an overall network design which is capable of transferring large files from a source node to a destination node by dividing them into chunks. The network design and protocol exploited the router and the broker logic and provided a multihomed and pipelined transmission of the packets.

We have performed experiments in order to test our Reliable Data Transfer protocol by introducing packet loss, delay, packet corruption and reordering of the packets to the links and observed how our RDT protocol behaves in these situations. We have inferred that when the packet loss percentage and the packet corruption percentage increases, the time it takes to transmit the file increases whereas when the packet reordering percentage increases, the file transmission time is not affected.

References