CENG435 Term Project Part-2 Report

Simge Nur Çankaya
Computer Engineering
Middle East Technical University
Ankara, Turkey
Student ID : 2099554
Group ID : 36
simgenurcankaya@gmail.com

Muhammed Süha Demirel
Computer Engineering
Middle East Technical University
Ankara, Turkey
Student ID : 2098911
Group ID : 36
msuhademirel@gmail.com

***Abstract*—In this project, we are expected to use the topology of part-1 and that topology consist one source, one destination and three router nodes. The topology will be discussed in the following chapters more detailed. Moreover, a UDP-based ”Reliable Data Transfer” (RDT) protocol of our own that supports pipelining and multi-homing. We are also expected to modify your routing implementations based on the requirements of the file transmission. Our protocol should be implemented by considering our own approach and design.**

***Index Terms*—UDP, RDT, Multihoming, Pipelined Protocols, Routing delay, ACK&NAK, Checksum**

I. Specifications

- Implementing a UDP-based ”Reliable Data Transfer” socket application. This application should contain ”Multihomig” and ”Pipelining Protocols” mechanisms.
- The links between the nodes should be met with reliable fashion by the nodes.
- ”s” and ”d” nodes should contain the implementation of for ”sender” and ”receiver” as an application layer implementation.
- The file will be sent from the node ”s” (source) to the node ”d” (destination) and its size will be exactly 5 Megabytes.

II. Introduction

Similar to the first part of the project, we have taken the slice from the GENI platform for the second part. Moreover, the hosts (s, r1, r2, r3, d) and the topology is the same with the part one but there will be no data flow between r1, r2 and r3 in this part of the project.

Implementing a Reliable Data Transfer protocol ,which will based on User Datagram Protocol, is our aim. This protocol should support pipelining protocols and multihoming mechanisms and the protocol should be implemented all links between the nodes.

III. Design & Implementation

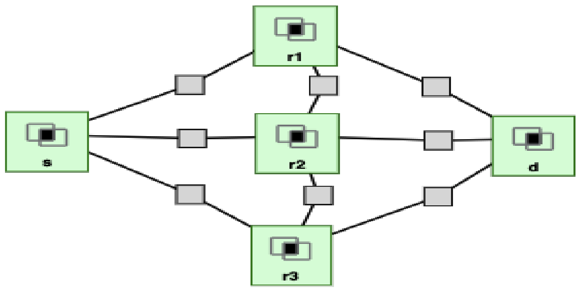

Fig. 1. The Structure of the Topology

The ”Colorado.Instageni” is the host of the slice and VMs. The topology of the project and node configurations are uploaded into the platform and necessary nodes/links via SSH connection. The methodology of establishing connection and making file transfer via SSH is explained in the report of the first project with the codes. We choose the Python2 as the programming language and we established the communication in the group with GitHub.

The topology is givin in the .xml format and imported to the platform in .xml format. Also, we are given configuration scripts, which are shown below, to implement into the topology. These scripts add determined delay which is equal to 3 miliseconds and three different packet loss percentage (%5,%15,%38) to links to the node that they run on.

The `configuration scripts`:

- tc qdisc change dev [INTERFACE] root netem loss 5% delay 3ms
- tc qdisc change dev [INTERFACE] root netem loss 15% delay 3ms
- tc qdisc change dev [INTERFACE] root netem loss 38% delay 3ms

Before the explanation of all of the developments and implementations, strategies, approaches and their reasons for our own protocol; we described briefly some key terms and mechanisms.

A. *Reliable Data Transfer*

The data will be transferred should :

- have no bit errors
- have no packet loss
- be delivered in the same sequence as they were sent to below layer.

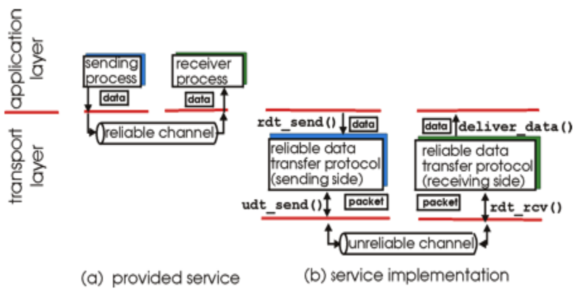

Fig. 2. The RDT Principles

B. *Checksum*

The transmitted segment may corrupt and contain some errors such as bit flipping.Detection of the errors in the segment is the aim of checksum.

The fields of the segment, contents in other words, is treated as 2-bytes (16-bits) integer sequence by the sender. The checksum is one's complement sum of the contents of the segment. User Datagram Port has a field for checksum and user puts the value of checksum into that field.

The receiver side of the communication calculates the checksum of the received segment too. Then the computed checksum by the receiver and the value in the "checksum" field of the segment are compered to each other to see whether they are same or not.

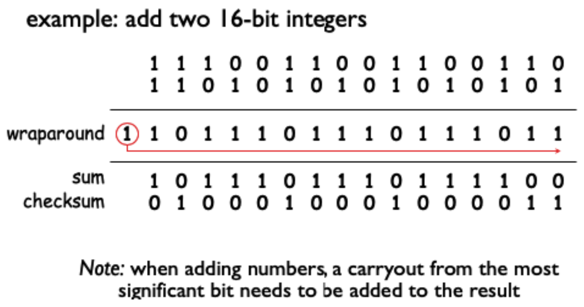
example: add two 16-bit integers
1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Fig. 3. The Checksum Example

C. *ACK & NAK*

To recover from errors in packets, receiver explicitly tells sender that packet received:

- is OK with acknowledgements(ACKs)
- had errors with negative acknowledgements(NAKs)

D. *Timeout*

Receiver is not able to understand if the last ACK/NACK packet is received without any problem at the sender and that is a problem. In order to solve this problem, sender waits "reasonable" amount of time for ACK and then:

- If ACK is not received in this time, it retransmits.
- If packet (or ACK) just delayed (not lost): Retransmission will cause duplicate; however, there is sequence number to solve this problem. Receiver must specify sequence number of packet being ACKed.
- Requires countdown timer.

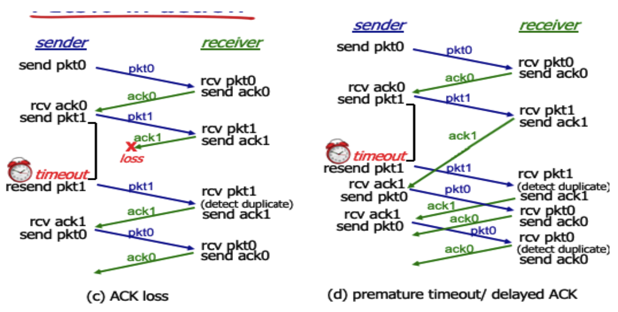

Fig. 4. The Timeout

E. *Pipelined Protocols*

Sender allows multiple yet-to-be-acknowledged packets. Two generic forms of pipelined protocols:

- Go-Back-N: Sender can have up to N unACKed packets in pipeline. Receiver only sends cumulative ACKs; doesn't send ACK packet if there's a gap. Sender has timer for oldest unACKed packet, when timer expires, retransmits all unACKed packets.

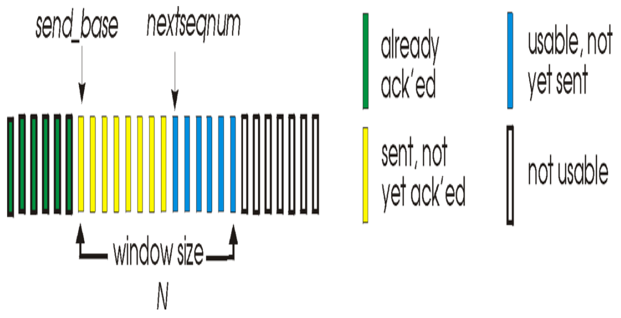

Fig. 5. The Go-Back-N

- Selective Repeat: Sender can have up to N unACKed packets in pipeline. Receiver sends individual ACK for each packet. Sender maintains timer for each unACKed packet. When timer expires, retransmit only that unACKed packet.

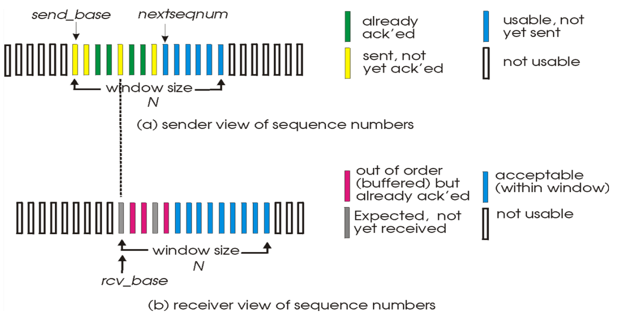
send_base
nextseqnum
already ack'ed
usable, not yet sent
sent, not yet ack'ed
not usable
window size N
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
acceptable (within window)
Expected, not yet received
not usable
window size N
rcv_base
(b) receiver view of sequence numbers

Fig. 6. The Selective Repeat

F. Multihoming

Multihoming is a mechanism used to configure one computer with more than one network interface and multiple IP addresses. It provides enhanced and reliable Internet connectivity without compromising efficient performance. The multihoming computer is known as the host and is directly or indirectly connected to more than one network.

G. Our Design Strategies

In the first part of the project, our main concern was calculating the transfer time, so our packages we used for the experiments were small. Our Dijkstra result was s-r3-d. So we started working with that 3 nodes for Experiment 1. We started with using the same send-receive functions that we have implemented previously. However, in the second part of the project, it is expected from us to send a file that has 5 Megabytes size and we can divide this file into packets which may have maximum 1000 bytes size.

We first implemented the package creator function and formed 5000 packets and then sent them over to destination node (d) without the loss. After that, to see if the data arrived correctly at the destination node (d), we got a checksum calculator code from the ”http://codewiki.wikispaces.com/ip_checksum.py”. That checksum calculator calculated the checksum of a given segment and returns two bytes as a checksum. We added this checksum calculator to the both source (s) and destination (d) nodes' source codes. Checksum function helped us to detect any bit error has occurred during the transmission of the segment from source (s) to destination (d). We calculated the checksum for the segment before sending it and we added this checksum value as an header to the segment to be sent. Tne node d after receiving the data, it divided data into two parts: The first part was the checksum value and the second part was the segment data. It compared the checksum value received with the checksum value it calculated. If the checksums were equal to each other, it saved the data. If the checksums were equal, it sent ”AKC0” which means data arrived correctly; else it sent ”ACK1” that means data was corrupted. Source (s) waits for an ”ACK” before sending new segment or the same segment again.

At this point, we have not received any data loss, so we implemented the first configuration that creates 5% data loss and 3 miliseconds delay on the interfaces between the nodes s-r3 and r3-d. After this, we got our first bit error in the data and additionally we have experienced that the ”ACK” may sometimes get lost during the transfer. Since we had not implemented a way to control the data arriving has arrived before our source (s) was also saving retransmitted data. In order to solve this problem, we implemented sequence number into the header. So, in the final form of our design implementation, our header consists of 2 bytes checksum value, 1 byte sequence number, altering between 0 and 1, and 950 bytes for data.

By implementing the sequence number, we solved the problem of saving the same data multiple times due to losing ”ACK” in the process of data transfer.

The node r3 does nothing other than receiving from the one node and sending the same data to the other note. We decided not to implement a checksum control into r3 in order to make our transfer process faster.

Because of the node s' design, it was waiting for an ”ACK0” or ”ACK1” packet to send the next or the same packet to the node d. If the ”ACK” get lost during the transfer process, node s was not able to send anymore data. To solve this problem, we introduced timeout to our socket. This solved the this problem as if the ”ACK1” was received and re-transmitted the packet one more time.

After that, we implemented two threads to work at the same time to sent different data to node r3 in node s; two threads in node r3 to work at the same time to sent the data received from node s to node d; two threads in node d to work at the same time to receive and save different data to node d.

This concludes our implementation for experiment 1. It can now do reliable data transfer with no package loss even when the loss ratio of the links is up to 38%.

In the design of the Experiment 2:

Multi-homing is one of the most important mechanism to provide reliable data transfer. When a link is down, multi-homing provides a way to send it over the other link. In order to understand whether a link is down or not, we implemented 3-way-handshaking mechanism into the nodes.

Our three way handshake mechanism first checks the links when we first run the code. It uses another port from the data sending ports in order to not the mix the ”SYN” with the data segments. The nodes r1 and r2 waits for a ”SYN” in a separate thread and when they receive an ”SYN” they send the ”SYNACK” package as a response ACK as an response to s' SYN and its own SYN, upon receiving the ”SYNACK”, the s node sends a ”SYN” meaning that it received the r1 or r2's SYN's correctly and this concludes the three-way handshake process. The connection is established after sending the last ACK. If an error occures during any part of this process, the link is interpretted as ”down” and it is not used until it is checked again with three way handshake process.

In experiment 2, our code starts with dividing the data into segments which have size of 950 bytes and adds them their separate sequence numbers to their header field and saves the data as a list in node s. And we created a window with size of 4 as type of list. The window is filled with the data which is taken from the data list in an order from index 0 to index len(datalist). By the 3-way-handshaking mechanism, we checked if the ports/links are open which means it is able to transfer data. We implemented 2 different port for each node r1 and r2 as we did in the experiment 1 for the node r3. So, we have 4 ports in total. Then, we sent the data in window over the open links and the sent data was deleted from the window. If node s receives the ”ACK0” packet, that means data is sent successfully; however, if it receives ”ACK1” packet, the data should be re-append into the window in order to be sent again. That means, we implemented the selective repeat mechanism, not go-back-n. We did not chose go-back-n because we thought that it is takes more time than selective repeat in a failure scenario. This process describes how we implemented pipelined protocol and multi-homing. Moreover, in the design of experiment 2, if the node s does not receive any ”ACK” packet for a time, it checks whether the link is down or not with 3-way-handshaking. And according to response of the 3-way-handshaking it changes the situation of the link ”open” to ”close” or remains ”open”. For a situation of a closed link is reopened, the node s checks the closed link with 3-way-handshaking every 20 data transfer.

IV. Experiment Specifications

- The file transmission is performed for two experiments: The source node will send the specified file to the destination node by using two different paths by conducting two different experiments:
  - The first path will be the shortest one based on the Dijkstra Algorithm that we found in the first part, and the second one will use the other available links ”at the same time”. These two different transmissions should be handled by using the same script. You can give the number of experiments as a parameter to your scripts and run them for two experiments.
  - For Experiment 1, our hosts (nodes) should route each of the packet based on the shortest path. This means that our file transmission will be over the shortest-path from the source node (s) to the destination node (d). The naming convention for the file can be for this transmission like input1 on the node ”s” and output1 on the node ”d”.
  - In Experiment 2, the source node exploits the remaining available links of (two) for a copy of this input file. The naming convention for the file can be like input2 on the node ”s” and output2 on the node ”d” for this transmission.
    - ∗ Exploit disjoint links between the source and destination. We will use the advantage of two links while transferring the file, exploiting these multiple paths will allow us to transfer the file faster. This part will provide us a multi-homed protocol that is expected to implement.
    - ∗ Assume that the shortest path is s-r3-d for experiment 1, and, then the second file transmission (for experiment 2) routing information should be like the following: (Use these paths at the same time) for the first path: s-r1-d and the second path s-r2-d, and for response messages also follow the inverse of these paths. The links between the r1 and r2; r2 and r3 will not be used.
    - ∗ The network may be faced with one type of failure which is link failure. One of the links between the source node ”s” and two other nodes (r1 and r2) can be down. Our reliable and multi-homed implementation should be aware of the links and update the routing and modifying the reliable transmission over the available link. This part will be applied only for the second experiment (Experiment 2).
- For two experiments:
  - The files will be protected with a checksum in the assessment. In other words, the input file at the source side, and the transmitted file at the destination side should be exactly the same.
  - We will implement and develop our own RDT protocol on top UDP. Therefore, we will use our own reliable protocol to send a large file from s to d. Then, the communications will be based on our RDT protocol.
  - Develop our reliable transport protocol that supports multi-homing by using UDP sockets.
  - Develop a packet-based protocol.
  - Design our packet structure that will go into the UDP payload. The maximum packet sizes of our RDT protocol (header + payload) can be at most 1000 bytes.
  - Only one script will run in each host.

V. Experimental Results

We are expected to plot the figure for two experiments by using our own reliable protocols for each experiment provides the relation of packet loss percentage and file transfer time with 95% confidence interval.

The configurations are listed below:

- Configuration 1: Packet Loss is equal to %5 .
- Configuration 2: Packet Loss is equal to %15.
- Configuration 3: Packet Loss is equal to %38.

.

To add emulation delay and packet loss for the experiments to the specified links, we used following code:

```
  tc qdisc change dev [INTERFACE] root
netem loss [LOSS] delay [DELAY].
```

We made the experiment-1 for 10 times with configuration 1 and its average result is 322.05 seconds; experiment-1 for 10 times with configuration 2 and its average result is 1057.29 seconds; experiment-1 for 1 times with configuration 3 and its average result is 4540.66 seconds. Hence, the third configuration takes so much time and the graph gives the general idea about the relationship between file transfer time and packet loss percentage, we have done 1 time the experiment1 with the configuration 3.

For experiment 1 :

- File transfer time for configuration 1 is 322.05 ± 42.41 seconds:
- File transfer time for configuration 2 is 1057.29 ± 197.13seconds:
- File transfer time for configuration 3 is 4540.66 seconds:

.

To plot the graphic for experiment 1, we have used octave-online, code is below:

```
>> x = [5, 15, 38]
>> y = [322.05, 1057.3, 4540.7]
>> err = [42.51, 197.13, 0]
>> figure
>> errorbar(x,y,err)
>> grid on
>> xlabel('Loss Percentage (%)')
>> ylabel('File Trabsfer Time (s)')
>> legend('Transfer Time')
```

Deriving from the data we got from the experiment1 and the graph, we can say that file transfer time increases while the packet loss percentage is increasing. The main reason of this situation can be explained like this: To recover loss packets,

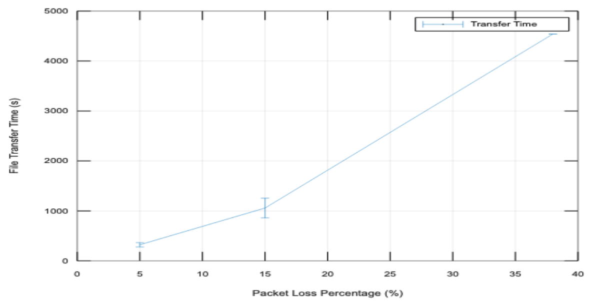

Fig. 7. File Transfer Time - Loss Percentage Graph For The Experiment 1

node s re-sends the loss data and while the loss percentage is increasing, re-send percentage is also increasing and this process consumes time.

We made the experiment-2 for 10 times with configuration 1 and its average result is 304.13 seconds; experiment-2 for 10 times with configuration 2 and its average result is 903.18 seconds; experiment-2 for 1 times with configuration 3 and its average result is 4017.93 seconds. Hence, the third configuration takes so much time and the graph gives the general idea about the relationship between file transfer time and packet loss percentage, we have done 1 time the experiment1 with the configuration 3.

For experiment 2 :

- File transfer time for configuration 1 is 304.13 ± 35.63 seconds:
- File transfer time for configuration 2 is 903.18 ± 106.66seconds:
- File transfer time for configuration 3 is 4017.93 seconds:

.

To plot the graphic for experiment 2, we have used octave-online, code is below:

```
>> x = [5, 15, 38]
>> y = [304.13, 903.18, 4017.93]
>> err = [35.63, 106.66, 0]
>> figure
>> errorbar(x,y,err)
>> grid on
>> xlabel('Loss Percentage (%)')
>> ylabel('File Trabsfer Time (s)')
>> legend('Transfer Time')
```

Deriving from the data we got from the both experiments and the graph, we can say that file transfer time increases while the packet loss percentage is increasing. Moreover, in the experiment we have better results because we have multi-homing mechanism and its re-send ratio is 2 times of the experiment 1's. Also, with the design of the experiment 2, we are able to tolerate down-link situation.

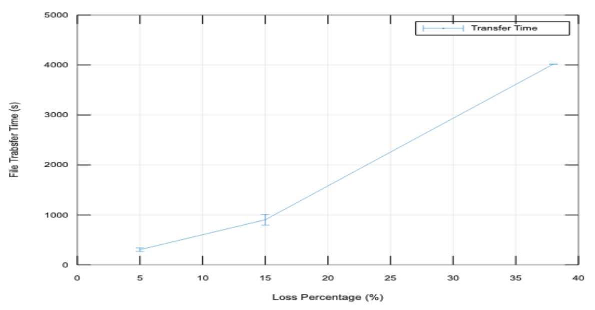

Fig. 8. File Transfer Time - Loss Percentage Graph For The Experiment 2

VI. Conclusion

In this part of the project, we have observed two experiments. The source node will send the specified file to the destination node by using two different paths by conducting these two different experiments. The first path was our fastest path (s-r3-d) according to the Dijkstra's Algorithm that we used in the first part of the project. The second path is the remaining two links between the source and destination (s-r1-d and s-r2-d).

The first experiment showed us the relation between file transfer time and packet loss percentage. The relation is when the packet loss percentage is increased, the file transfer time is increased too. Hence, RDT requires no packet loss; when a loss detected by receiving ”ACK1” or no ”ACK”, the node s re-sends the packet and every extra action takes time obviously.

In the second experiment, we observed same relation between packet loss percentage and file transfer time. But in this experiment, we also implemented a multi-homed protocol and we transferred over the two links and used these two links in the same time and this situation allowed us to transfer the file faster than transferring over one link. That design made our script faster then the design of the experiment 1 because experiment 2 has better tolarate rate than experiment 1. Moreover, in the case of a failure in one of these two links, the route of the transmission is updated and the transmission is continued on the available link and that provides us RDT.

References

threading - Thread-based parallelism¶. (n.d.). Retrieved from https://docs.python.org/3/library/threading.html.

socket - Low-level networking interface¶. (n.d.). Retrieved from https://docs.python.org/3/library/socket.html.

time - Time access and conversions¶. (n.d.). Retrieved from https://docs.python.org/3/library/time.html#time.time.

Page. (n.d.). Retrieved from https://wiki.python.org/moin/UdpCommunication.

Updated November 02, 2017 / P. J. 24. (n.d.). List of Well-Known TCP Port Numbers. Retrieved from https://www.webopedia.com/quick_ref/portnumbers.asp.

What is Multihoming? - Definition from Techopedia. (n.d.). Retrieved from https://www.techopedia.com/definition/24984/multihoming.

(n.d.). Retrieved from http://codewiki.wikispaces.com/ip_checksum.py.