**CS 3251 - Networking I**

**Programming Assignment 2**

**Spring 2018**

Neil Barooah – nbarooah3

Tai Tran – ttran95

**Project Description**

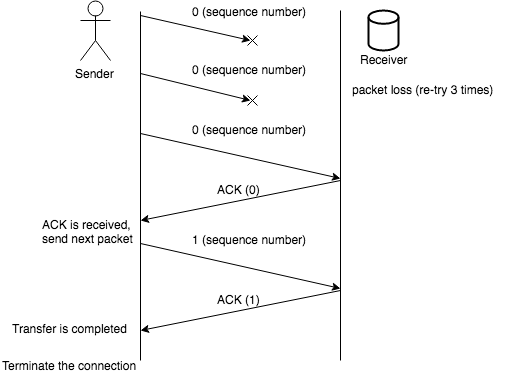
In the initialization process, the Ringo runs the peer discovery process, in which it discovers all other Ringos in the network in the form of IP address and port number pairs. Based on its knowledge of all the peers, each Ringo then pings all the other Ringos to calculate the RTT vector. During this process, the Ringo calculates the time it takes to ping its neighbors until it receives an ACK; this is then used to build the RTT vector for each Ringo. At this point, each Ringo shares its own RTT vector with all other Ringos connected to it. This helps each Ringo build an RTT matrix. The RTT matrix helps determine the optimal path when routing data from sender to destination.

Stop-and-Wait Protocol

We will be using the Stop-and-Wait Protocol to implement this protocol. The following captures the main ideas:

* Sender sends the packets to the receiver based on the RTT matrix, with the sequence number attached to each packet. In the event optimal path contains forwarders between the sender and receiver, the packet goes via the forwarders.
* The receiver receives the packet. It uses checksum to ensure that the data is not corrupted. It then sends the ACK signal back to the sender (via forwarders if applicable) to acknowledge successful receipt of the packet.
* If no ACK is received by the Sender based on a end-to-end RTT of the path, then the sender will attempt to resend it to the receiver until an ACK is received. The Keep-Alive mechanism ensures that the Sender always takes the optimal path.

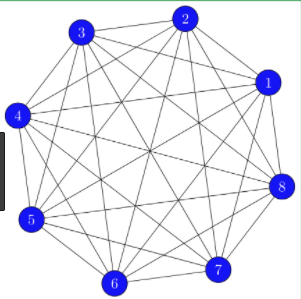
The time diagram below illustrates the simple protocol between the sender and the receiver assuming no forwarders in this case:



*Figure 1: Timing Diagram*

Peer Discovery Algorithm

The first task that each Ringo needs to perform is peer discovery, so it can know about all other Ringos in the network. This is a critical piece of information that the Ringos need to know in order to send packets to other Ringos. Essentially, one can think of this as forming a map of the structure of the network. This required as every Ringo only knows at most one other Ringo, also known as it’s Point of Contact (PoC). Assuming that the overall structure of the network is in the form of a large ring as shown in the diagram below:



*Figure 2: Sample Ringo Network*

The key assumption that we make is that each (RingoX, RingoY) where RingoY is the PoC of RingoX provided to us by the user will form a Connected Graph when combined. That is, there will be no isolated vertices so it is possible to reach all other vertices from any point. Sharing of known peers as each Ringo gets to know about a new peer from its peers will ensure that every Ringo in the network is ultimately aware of each other. As shown in Figure 2, each Ringo will have more interconnections with all other Ringos and the peer discovery will help discover all the other Ringos in the network.

The pseudocode for Peer Discovery is as follow:

def discover\_peers():

For T from 1 to 2:

For each Ringo RI  from i = 1 to i = N:

other\_nodes = Set containing the IP Address and UDP host pairs of all RK where k < i.

This is initially empty for R1.

RJ = PoC of Ri

RI pings RJand passes the set other\_nodes.

RJ adds any new elements from the received other\_nodes to its own other\_nodes set.

In the pseudocode described above, each Ringo will store the other\_nodes set. after completing 1 full loop, the last Ringo RN will know about all other N - 1 Ringos, while Ringo RN-1 will know about all N - 2 Ringos. In the second loop, RN provides R1 will the entire set of Ringos in the network. Similarly, R1 provides this to R2, and so on until the second loop is complete. At this stage, all Ringos will have known about all the other Ringos in the network. Once having discovered all peers, each Ringo will share it’s role (Receiver/Sender/Forwarder) to all its peers. This ensures that each Ringo is aware of the roles of every other Ringo which will help with routing packets correctly.

Round-Trip Time Measurements

Each Ringo needs to take Round-Trip Time (RTT) measurements in order to construct its RTT vector. In order to do this, each Ringo will ping all of its peers and measure the time it takes to receive an acknowledgement. This is known as the RTT. Due to the randomness in networks, we repeat this process 5 times and take an average in order to gain reliable RTT measurements. The following pseudocode captures this concept:

def build\_rtt\_vectors():

For each Ringo RI from i = 1 to i = N:

peers = set of all other Ringos in the network except RI.

rtt\_vector = A vector of length N - 1 that is initialized to ∞.

For each Ringo RJ in peers:

rtt\_j = 0

Repeat 5 times:

RI pings RJ at time tX.

Suppose RI received ACK from RJ at time tY:

rtt\_j ← tY - tX

rtt\_vector[j] ← rtt\_j / 5

Now, each Ringo contains an RTT vector, which is the time it takes to travel to other Ringos. The next aspect is to combine these RTT vectors in order to form the RTT matrix. Each Ringo will share this information with all of its peers such that we can combine the RTT vectors in order to form the RTT matrix at every Ringo. The following pseudocode captures this idea:

def build rtt\_matrix():

For each Ringo Ri from i = 1 to i = N.

rtt\_vector ← RTT vector of RI.

rtt\_matrix ← Initial matrix of size N x N containing ∞’s in order to build the RTT matrix.

rtt\_matrix[i][i] = 0

rtt\_matrix[i][all indices except i] = rtt\_vector

for each peer RJ of RI:

RI sends RJ its rtt\_vector (try until succeeds).

RJ adds rtt\_vector to its own rtt\_matrix

In the above pseudocode, each Ringo initializes an RTT matrix and sends it’s RTT vector to all its peers. Each peer then adds the RTT vector to its own RTT matrix. By the end of this subroutine, each Ringo would have the same RTT matrix. Therefore, every Ringo will know the RTT vector of every other Ringo. This helps with determining the optimal path.

Optimal Ring Formation

Given that each Ringo now contains an RTT matrix containing values for directly connected neighbors, our task is now to compute the optimal ring amongst all the Ringos. Here, optimal ring is defined as the path that covers all N Ringos in the network such that the cumulative RTT is the minimum amongst all possible paths of length N.

def compute\_optimal\_path():

paths ← All paths of length N such that each Ringo is contained in path

rtt\_matrix ← RTT matrix of RI.

optimal\_path ← None; optimal\_RTT ← MAX\_INT

For each path p in paths:

rtt ← 0

For Ringo rI in p:

rtt += rtt\_matrix[rI][rJ]

if rtt < optimal\_RTT:

optimal\_RTT = rtt

optimal\_path = p

This fully completes the initialization for each Ringo. Given that all Ringos have the same RTT matrix, the optimal ring for each Ringo will also be the same as it’s based on the RTT matrix measurements. Based on the knowledge of Roles of each Ringo, every Ringo will formulate an optimal path based on the optimal ring. From any given Sender, there are 2 possible paths to a Receiver in an optimal ring: one is the clockwise direction and other is the counter-clockwise direction. In order to determine the optimal path, we use a similar cumulative RTT approach to determine which of the 2 directions is fastest way to reach the Receiver.

Churn and Keep-Alive Mechanism

To implement the Churn and Keep-Alive mechanism, each Ringo in the network sends out short message to all its peer at all times on a separate thread while a port is dedicated in each Ringo to serve Keep-Alive requests. Due to time-outs in the network and given that each Ringo goes offline for at least 15 seconds, we will keep trying to reach a Ringo for 5 seconds until we receive an ACK. If no ACK is received, it is assumed that the Ringo is offline. In the event that one of the Ringos go offline, all the other Ringos check if that Ringo was contained in the optimal path being used for data transfer. If so, the optimal path is recomputed; that is, the path is reset to the opposite direction (if clockwise, it is now counter-clockwise). Another aspect of Keep-Alive is tackle the issue of a Ringo needing to regain all basic information such as who its peers are, RTT vector, RTT matrix, optimal ring etc. In order to make this process fast, each Keep-Alive message contains this information. Hence, once a Ringo wakes up, it can regain all of this information based on Keep-Alive messages it receives rather than having to recompute everything.

Lost Packets

Lost packets are accounted for by sequence numbers attached to each packet. The starting sequence number is a random number between 1 and 1000000000 decided upon by the Sender. The sequence number increases with the number of bytes of data sent in each packet. The Receiver keeps track of the sequence number of the last packet received. This helps ensure that the Receiver is not collecting duplicate data. Accordingly, the Receiver sends back an ACK sequence number, which guides the Sender as to how much data the Receiver has received. When the Receiver receives (last sequence number sent + length of data sent) as the ACK sequence number, it proceeds to send the next fragment of data.

Corrupted Packets

In order to handle corrupted packets, we will use a checksum mechanism to ensure the packets are received by the sender as intended. This will be based on the checksum values contained in header of the packet, computed by the sender. We compute 2 checksum values: one for the IP header and one for the UDP header + data sent. In case the data is corrupted, the receiver discards the packet will send a NACK to the sender. This signals to the sender that the packet was corrupted and it re-sends the same packet to the receiver. The checksum value is computed using the Fletcher Algorithm as shown in the pseudocode below:

def verify\_checksum(msg):

s = 0

if (len(msg) % 2) == 1:

msg += struct.pack('!B', 0)

for i in range(0, len(msg), 2):

w = (msg[i] << 8) + (msg[i + 1])

s += w

s = (s >> 16) + (s & 0xffff)

s = ~s & 0xffff

s = s % 65535

return s == 0

Thread Architecture

In order to concurrently manage multiple tasks, we are using multiple threads in the project. One thread is dedicated to the server for each Ringo, which responds to all normal interactions received from other Ringos. Another thread is dedicated to pinging peers for Keep-Alive while another thread is dedicated to responding to all Keep-Alive messages. Lastly, the main thread acts as the client and interacts with all other clients to send packets and does all other computations such as Peer Discovery, RTT Vector, RTT Matrix, Optimal Ring Formation, Optimal Path etc.

Once a Ringo is taken offline with the “disconnect” command, all threads mentioned above are paused for a period of time specified by the user. Upon waking up, the Ringo is reset so no information is retained and all threads are resumed with the Keep-Alive threads going up first so the Ringo may regain all its information before interacting with its peers for data transfer.

Packet Header

In this section, we describe the structure of the each packet consisting of 475 bytes.

|  |  |  |
| --- | --- | --- |
| Source Address (4 bytes) | | |
| Destination Address (4 bytes) | | |
| Zeroes (1 byte) | IP Protocol (1 byte) |  |
| Source Port (2 bytes) | | Destination Port (2 bytes) |
| Length of Entire Packet (2 bytes) | | IP Checksum (2 bytes) |

*Figure 3 (a): IP Header*

|  |  |  |
| --- | --- | --- |
| UDP Source Port (2 bytes) | | UDP Destination Port (2 bytes) |
| UDP Length (2 bytes) | | UDP Checksum (2 bytes) |
| UDP Sequence Number (4 bytes) | | |
| UDP ACK Sequence Number (4 bytes) | | |
| Control (1 byte) |  | Data Length (2 bytes) |
| Data (475 bytes) | | |

*Figure 3 (b): UDP Header*

* **Source Address**: contains IP address of Sender.
* **Destination Address**: contains IP address of Receiver.
* **Zeroes**: No specific purpose. This is to ensure an even number of bytes for packet size.
* **IP Protocol** – Protocol used for data transfer (UDP).
* **Source Port**: contains the Sender’s port number.
* **Destination Port**: contains the Receiver’s port number.
* **Length of Entire Packet**: 18 (IP) + 19 (UDP) + data\_length in bytes.
* **IP Checksum**: Checksum value of all IP header fields.
* **UDP Source Port**: contains the Sender’s port number.
* **UDP Destination Port**: contains the Receiver’s port number.
* **UDP Length**: 19 (UDP) + data\_length in bytes.
* **UDP Checksum**: Checksum value of UDP header fields and data.
* **UDP Sequence Number**: Attached to each packet to maintain reliability in terms of the order of the packets transmitted. Starting sequence number is chosen by Sender on random between 1 and 1000000000.
* **UDP ACK Sequence Number**: Attached to each packet to maintain reliability in terms of number of bytes successfully received by the Receiver. This is used by the Sender to determine next fragment of data to be sent.
* **Control**: This helps determine the type of packet (SYN, SYN\_ACK, ACK, NACK, FIN, DATA).
* **Data Length**: indicates the payload size/block.
* **Data**: data being transmitted.