# 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 determine the ones that are directly connected to it. 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 the packet is timed out after 3 seconds, then the sender will attempt to resend it to the receiver for 3 times. If it fails, then the sender calculates the next best path from the RTT matrix and repeats the entire process again.

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

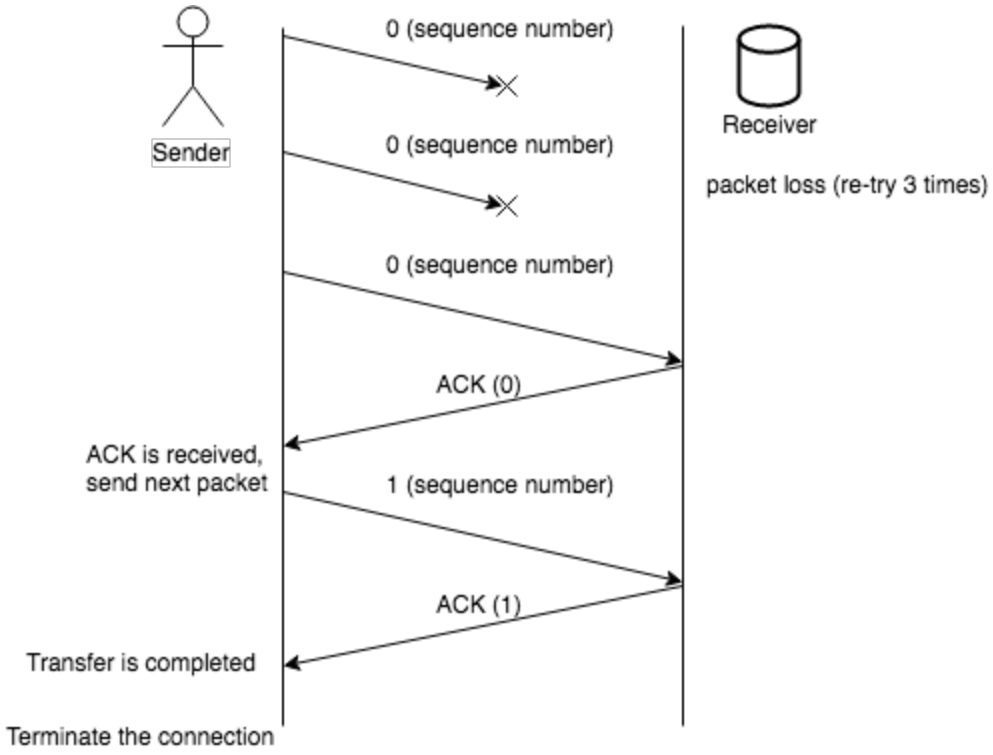


*Figure 1: Timing Diagram*

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 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:

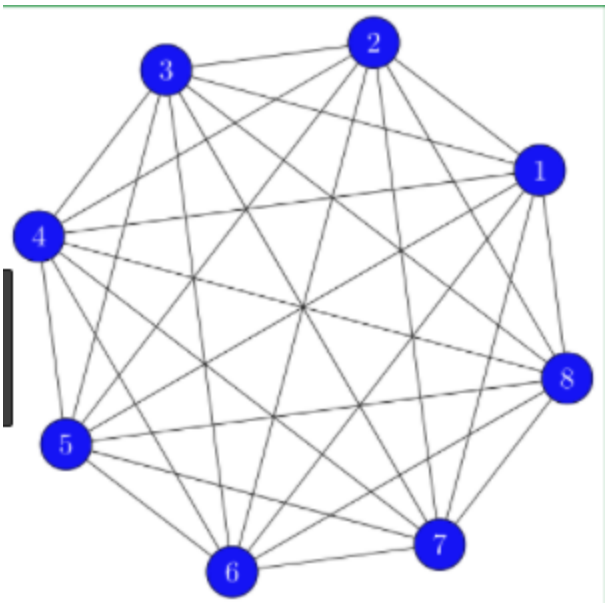


*Figure 2: Sample Ringo Network*

The key assumption that we make is that each ($Ringo_X$, $Ringo_Y$) where $Ringo_Y$ is the PoC of $Ringo_X$ provided to us by the user will be unique. In other words, given ($Ringo_X$, $Ringo_Y$) pair, we don't expect to see ($Ringo_Y$, $Ringo_X$) again. Rather, $Ringo_Y$'s PoC will be some other ringo other than $Ringo_X$. As shown in Figure 2, each Ringo will have more interconnections with 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 R_I  from i = 1 to i = N:
         other_nodes = Set containing the IP Address and UDP host pairs of all R_K where k < i.
                              This is initially empty for R_1.
         R_J = PoC of R_i
         R_I pings R_J and passes the set other_nodes.
         R_J 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 $R_N$ will know about all other N - 1 Ringos, while

Ringo $R_{N-1}$ will know about all N - 2 Ringos. In the second loop, $R_N$ provides $R_1$ will the entire set of Ringos in the network. Similarly, $R_1$ provides this to $R_2$, 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.

Round-Trip Time Measurements

The second aspect of peer discovery is the notion of knowing which Ringos are directly connected to each other. For instance, $R_1$ knows about all other Ringos $R_2$, $R_3$, …, $R_N$ but it may only be connected to $R_5$. This is an important piece of information. Therefore, the next part of the step is to know which Ringos are directly connected and the distance between them (time between $R_I$ and $R_J$ where i and j are directly connected). The following pseudocode captures this concept:

```
def build_rtt_vectors():
    For each Ringo R_I from i = 1 to i = N:
        other_nodes = set of all other Ringos in the network except R_J.
        rtt_vector = A vector of length N - 1 that is initialized to ∞.
        For each Ringo R_J in other_nodes:
            R_I pings R_J at time t_X.
            If R_I received ACK from R_J at time t_Y:
                rtt_vector[j] ← t_Y - t_X
            Else:
                Retry 3 times.
                If no ACK received from R_J:
                    R_I is not directly connected to R_J.
```

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 other Ringos such that we can combine the RTT vectors in order to form the RTT matrix. In the process described above, each Ringo essentially know who its neighbors are. Therefore, each Ringo sends its RTT vector to all of its neighbors to eventually obtain the final RTT matrix. The following pseudocode captures this idea:

```
def build rtt_matrix():
    For each Ringo R_i from i = 1 to i = N.
        rtt_vector ← RTT vector of R_I.
        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 j in rtt_vector where j != ∞ and j != i:
    R_J ← neighbor of R_I.
    R_I sends R_J its rtt_vector (retry if failed).
    R_J 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 neighbors. The neighbor then adds the RTT vector to its own RTT matrix. By the end of this subroutine, each Ringo would have an RTT matrix with non-$\infty$ values representing the RTT between neighbors. All $\infty$ values are unreachable by that Ringo directly. These values will be filled while we find the optimal ring formation.

Optimal Ring Formation

Given that each Ringo now contains an RTT matrix containing values for directly connected neighbors, we will now use Dijkstra's Algorithm to find the RTT between Ringos that are not directly connected.

```
def complete_rtt_matrix():
    For each Ringo R_I from i = 1 to i = N:
        rtt_matrix ← RTT matrix of R_I.
        For each Ringo R_J that is not a neighbor of R_I:
            t = time taken to travel from R_I to R_J using Dijkstra's Algorithm.
            rtt_matrix[i][j] = t
```

This fully completes the RTT matrix for each Ringo. Using this matrix, the Ringos can now optimally route packets based on the destination Ringo provided by the user.

Churn and Keep-Alive Mechanism

To implement the Churn and Keep-Alive mechanism, each Ringo in the network sends out a short exchange message to its peer periodically. The keep-alives will be sent after every 15 seconds. If the Ringo doesn't receive an ACK from its peer, then it knows that the connection is terminated or the Ringo is down. In the case that the packet is being sent to the receiver and the connection is down, the keep-alive mechanism will let the Ringo know, and it will try to find the second best available path to reach the receiver.

Lost Packets

In the event that a sender does not receive an ACK from the receiver within 3 seconds of sending the packet, it will attempt to send the packet again 2 more times.

Corrupted Packets

In order to handle corrupted packets, we will be using a checksum mechanism to ensure the packets are received by the sender as intended. This will be based on the checksum value contained in the byte of the header of the packet, computed by the sender based on the data in the packet. 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 packet to the receiver.

Packet Header

In this section, we describe the header structure of the each packet.

| Source Port (32 bits) |
|---|
| Destination Port (32 bits) |
| Sequence Number (32 bits) |
| Data Length (32 bits) |
| Checksums (8 bits) |
| NACK(1 bit) |
| ACK (1 bit) |
| SYN (1 bit) |
| FIN (1 bit) |
| Tasks (2 bits) |

*Figure 3: Packet Header*

- **Source Port**: contains the sender port number.
- **Destination Port**: contains the receiver port number
- **Sequence Number**: Attached to each packet to maintain reliability in terms of the order of the packets transmitted.
- **Data Length**: indicates the payload size/block.

- **Checksum**: used to verify the data integrity based on the data.
- **NACK**: used to let the sender know that the previous packet was corrupted as a result of the checksum computation done.
- **ACK**: used when the receiver has successfully received the packet.
- **SYN**: used to indicate the establishment of a connection.
- **FIN**: used to indicate that termination of a connection.
- **Task**: bits used to classify the type of task to be performed. For instance, keep-alive is a task and computing the RTT is a separate task. Based on this information, each Ringo will process the packet accordingly.