Brown Computer Science Capstone Writeup: Neighbor Discovery Protocol and TCP Tahoe

Colin Savage

May 24, 2024

1 Introduction

For my CS Capstone, I worked on two extensions to the existing functionality for the IP and TCP projects of CSCI 1680: Computer Networks. The first was a proposed structure for a Neighbor Discovery Protocol algorithm on top of our existing codebase. For the initial assignment, we assumed all hosts were aware of their neighbors on startup, however this is almost never the case in the real world. The Neighbor Discovery Protocol would allow a host to connect to a router on startup and establish itself as a neighbor on the network given minimal information. The second of the two extensions was an implementation of the TCP Tahoe congestion control algorithm. In general, congestion control is used to prevent the sending of more information than the network can handle, and various algorithms have been popularized to manage this. TCP Tahoe is split into three main stages, 'slow start', 'congestion avoidance', and 'fast retransmit.' The first is a slow increase to the congestion window size (m_cwnd), which determines the number of packages sent per RTT (round-trip time). This value is increased exponentially until we reach a point where no all packets are acknowledged correctly. The second phase, congestion avoidance, is now an additive increase to m_cwnd with a decrease every time congestion is detected (a packet is lost), using the m_ssthresh variable. Lastly fast retransmit is used when we receive 3 duplicate acks (this means we have lost a packet somewhere) and the lost packet is send immediately. This is generally much faster than waiting the entire RTO time.

2 Neighbor Discovery Protocol

The way that I would implement this would be a one-time version of RIP, that for the purposes of the assignment, I'm going to call NDP (Neighbor Discovery Protocol). Since the network is static (in terms of who has what neighbor), once we are able to assign each host their neighbors we can assign each host their neighbors before anything to do with RIP or sending test messages begins, so we then start up these processes in the exact same way and have our code run as if nothing different had been done.

To go into more specifics about this, I'd define a new message type 129 (NDP_MESSAGE) with two protocol fields, 1 (NDP_INFO) and 2 (NDP_RESPONSE). The pieces of information that we need to have for this are as follows:

- Router:

```
    - Interfaces (Which VIPs they cover)
    - Neighbors (Their VIP and UDP Address + Port)
- Host
    - Just their (UDP Address + Port)
```

The router needs to have all of its host's VIPs and UDP address info to be able to forward them messages in the future, but also to be able to send the hosts their info as well as the info of the neighbors. They also need to know their interfaces so that they can only send hosts the nodes that correspond to their subnet (are their neighbors). The Hosts only need to know their UDP address and port so that they can receive this initial message from the network.

I can also foresee an implementation where the router doesn't know about the neighbor's VIP, only the UDP information, while the host already knows its own VIP. That being said, this would require multiple exchanges of info, since the router would need to get back the VIPs of all of the hosts, then send another message containing the VIPs of the nodes in the subnet for each of the nodes themselves so that they actually get their neighbors. For this reason, I think by implementation would be better to use.

In the end, either the host needs to know it's own VIP, or the router needs to know the VIPs for all of its hosts. and in any case, the all nodes need to know about their own UDP address and port and the router needs to know about the host's UDP address info at the start to be able to initiate this process.

When we start up, we make the host's neighbor list completely empty and have them listen for UDP broadcasts on their assigned ports. On the router side, we set up the neighbor list, interfaces and their respective neighbors they encompass and the basic routing table with local routes as we did before. Then we send an NDP_INFO message to each of the neighbors of the router which is not an rip_neighbor (for the sake of simplicity, we'll assume that being an rip_neighbor <=> it is a router, in an actual implementation we'd handle edge cases for this). This message would hold a struct defined as

```
struct ndp_msg_t{
    uint16_t command;
    uint16_t num_entries;
    struct inaddr ifaddr;
    struct inaddr ifmask;
    char ifname[LNX_IFNAME_MAX];
    ndp_msg_entry_t entries[];
}

struct ndp_msg_entry_t{
    struct inaddr vip_addr;
    struct inaddr udp_addr;
    uint16_t udp_port;
}
```

Note: The ndp_msg_entry_t struct is identical to the lnx_neighbor_t struct just with the list_link_t field removed since we're handling all of our list management in std::vector objects in C++.

The router will craft a separate message for each of its subnets, so that the host is not getting any

more information than it should from this process. If a router (r1) is connected to two hosts (h1,h2) in the same subnet and one additional router (r2), the message which goes to h1 and h2 will contain three entries, r1,h1, and h2. There's no need to remove an entry when we're sending to that entry, since that entry can distinguish that it's itself based on the udp_addr and udp_port. In fact there should also be no issue with just not being able to distinguish since then the host is added to its own neighbor list and can send to itself. We've already handled determining which of the hosts and router belong to which interface as a part of our normal functionality, so we determine which nodes to add to this message by iterating over each interface's neighbors list and crafting a ndp_msg_t struct with interface_neighbors.size()+1 entries in the entries list (to include the router itself). This way, the host receives a message with each of the entries in its subnet (all of its neighbors).

Once the host receives this message and determines that it is, in fact, a NDP_MSG of protocol type NDP_INFO, it iterates over this list, using std::reinterpret_cast as in our code to access each individual object, before adding these to its neighbors list and subsequently its interface's neighbors list as well (there really should be no difference between the two). There should be no additions to the rip_neighbor list, since hosts don't participate in RIP (in fact we should probably remove this list from our Host object in our normal implementation). It also adds to the routing table since each of the neighbors that it receives also constitutes a local route.

I also added a type called NDP_RESPONSE, so that a host can let the router know that it has received all of the info and handled it correctly. This doesn't seem entirely necessary, but if we implemented it we would have the host send this message as soon as it was done parsing through the NDP_INFO message, and the router would send all of the messages and listen for all of its responses before continuing to RIP and the CLI.

3 TCP Tahoe

For my capstone, I implemented the TCP Tahoe congestion control algorithm. This can be used in conjunction with our original TCP implementation by using the sc or sf commands with 'tahoe' as the inputted option.

For this implementation, I initialized m_cwnd to 1024 and m_sstresh to 4000 ($MAX\_WINDOW/2$).

For slow start, I increased the value of m_cwnd by the value of the amount we acked every time we received a non-duplicate ack. This effectively the same as doubling every time we receive our full window size back (up until we reach m_cwnd==MAX_WINDOW). The second part of this is that if we see any sort of packet loss (this is done by a duplicate ack), we halve the value of m_cwnd and set the original value to be m_ssthresh.

Congestion avoidance was probably the most difficult part to implement because of its involvement with the timing aspects of RTO calculation, however I've verified these times to be correct in our normal implementation, so I had the benefit of building on top of an already working component. One of the issues in our TCP implementation is that we require a slight delay in between write times, so I devised a more mathematically intensive, but equally effective way to achieve the additive increase. What I do is to weight the increase in m_cwnd by the ratio of the maximum segment size to the current window (this gives a slightly subadditive increase but I'm okay with this).

The fast retransmission portion of the algorithm was a bit easier to implement since for TCP Tahoe, I didn't need to worry about the fast recovery aspect of the program. I simply kept a counter of duplicate acks (I could check this by comparing the received ack to our current stored sequence

number) and once this number reached 3 I immediately sent a retransmission of the first element in the retransmission queue (not waiting the time for the retransmission to be automatically triggered). It was likely that this packet was dropped by chance, so this fast retransmission will probably be accepted and we can move on much faster than had we waited for the retransmission and also potentially avoid a zero-window probing case if we were to get this packet accepted by the receiver after the rest of the window is already filled. After retransmitting this package, I set m_ssthresh = m_cwnd/2 and m_cwnd=1024 (the initial value) and we start up the slow start process again since we deem this to be a "catastrophic event"

Packet Captures: See corresponding .pcap files for specific info.

Example 1: (Congestion Control with no Drop Rate): Since we had essentially no packet loss in this case, there was nothing interesting to report about this. The Tahoe algorithm spent almost the entire time in congestion avoidance and the m_cwnd variable had essentially no effect on the algorithm. In addition, because there was no packet loss, there was no fast retransmit to be done, so there were no performance benefits from this. We ended up getting a runtime of 2.44 seconds, which was pretty comparable to our original implementation (even a little bit faster).

Example 2: (Congestion Control with 2% Drop Rate): Now, because of the drop rate, we actually had some fluctuation between slow start and congestion avoidance throughout the running of the sf command, so our congestion control was actually being used. In addition, the fast retransmit being used in cases where a packet was dropped, and we definitely reaped the performance benefits of not having to wait the full 250ms before retransmitting a packet. Our runtime for this was 13.18 seconds, which was 2x faster than with no congestion control.

Example 3: (Multiple sf and rf operations going on at once and 2% Drop Rate): In order to test this, I started two rf commands, then went over to the other host and ran two corresponding sf commands. As a result of the brute-forcey nature of this, they didn't exactly overlap (i.e. there was definitely before the first one started). I tried doing this without the drop rate, but the runtime was too fast for me to get the second command off in time. Note that in the attached packet capture for Example 3, we include the sending messages for both of the sf commands (I couldn't figure out how to distinguish between the two in Wireshark). We definitely had a slightly increased runtime from the previous example, going up to 15-16 seconds, but this is still much faster than without the congestion control in place.