CSE 160
Computer Networks

Project #1 Report

September 20, 2018

Members

Chris DeSoto
Khushpreet Buttar

1 Design Decisions

Our design process for Project #1 began with a simple analysis of the skeleton code provided for this project. Specifically we studied the given data structures and determined whether or not they would be adequate for the tasks assigned to us in the project. While the Hashmap data structure proved useful for the Neighbor Discovery portion of the project, Flooding required a slightly more complex data structure. Because we needed to track packets from every node within our network's broadcast domain we required a list of some sort. However, storing all the packet sequence numbers in a single list could prove to be a less than ideal solution as it would require an iteration over the entire list with the receipt of every packet. Our solution was to design a new data structure, much like Hashmaps in traditional programming languages in like Java, in which a key could hash to a list of values rather than a single value. The bucket of our MapList consists of an integer as the key and a list of integers as it's value. This data structure cuts down on the number of memory accesses by our algorithm by searching only the particular sources previous packet sequence numbers.

With this data structure in place, Flooding proved to be simple. A variable of type uint16_t was used to hold the value for the sequence numbers. That value and a static value for the TTL of each packet were used in the makePack function. A PING is sent out and the sender's sequence number is incremented. When a packet is received and the node is not the destination, the packet is forwarded on to all neighbors with a decremented TTL. If the node is the destination, the packets sequence number is entered into the MapList and a PINGREPLY is sent back to the source of the original PING.

Neighbor Discovery was relatively simple to implement, with a single caveat. How can we differentiate a normal PING from a PING sent by our neighbor discovery component? We rejected the idea of introducing a new protocol, as the instructions specifically stated that the project could be accomplished with only PING and PINGREPLY (however we admit that that solution may have been simpler and more elegant than ours). Without this, there is a situation in which we cannot tell if a packet came from a neighbor or was simply being forwarded. The only data we have access to is the source field, which is not the last node to transmit the packet (our neighbor). Our solution was to PING all neighbors, periodically, with a packet containing a destination of 0. The neighbor nodes were then programmed to recognize this packet and respond by immediately transmitting a PINGREPLY packet, also with a destination of 0, to all of its neighbors. This behavior allowed us to track packets received from neighbors and create a List of all of the neighbors of each node.

The only thing that remained to be implemented was a way to prune old neighbors that had left the network and were no longer responding. At this point we decided that the List may not have been the best data structure to use. In order to track when each PINGREPLY was received we would need to include a time with the node id of each neighbor. The obvious solution is to use the Hashmap component provided to us with the skeleton code. When the Neighbor Discovery timer is fired, neighbors that have not replied in a specified amount of time are removed. Finally, we included a component to introduce randomness to our Neighbor Discovery timer so each node would not broadcast at the same time and cause collisions.

2 Discussion Questions

1. Event driven scheduling allows for simple scheduling of processes. Instead of a complex time-sharing scheme, events simply preempt the system's main loop and perform their task. Queuing can be used to resolve time conflicts. It is also particularly useful for embedded devices which often contain sensors and respond pragmatically when triggered by some event (a user interacting with a touchscreen, a timer triggering a sprinkler system, a sensor detecting rain, etc.). A downside of event driven programming is that the code can often lack a hierarchical structure making it difficult to develop large applications. Components within an event driven application can act independently, or be triggered by other components, or even a series of other components. Once an application becomes sufficiently large it can be nearly impossible to debug.

2. By having both flooding checks and TTL, we can prevent both endless cyclic packet flooding and packet transmission away from the destination. If we had only TTL implemented without flooding checks then packets would be forwarded unnecessarily in cycles and indirect routes until their TTLs expired. This is wasted bandwidth on the network as the packet has already been forwarded through those nodes. With flooding checks and no TTL, packets can continue to be forwarded further and further away from the destination. If the topology of the network is somewhat linear this could result in a majority of the network receiving the packet despite being nowhere close to the destination. This situation also results in wasted bandwidth.

3. The best case situation would be if all nodes are connected in a line(1-2-3-4-...-n). This would mean every packet was transmitted "forwards" and "backwards" for each link, resulting in $2 * (n - 1) - 1$, for $n > 1$, packets going out. The worst case would be if all the nodes were connected to every other node. This would mean $n-1$ packets go out to $n-1$ nodes. Then each of the nodes, except the destination, would send out $n - 1$ packets with the total number of packets sent being $(n - 1)^2$. Best Case: $O(n)$, Worst Case: $O(n^2)$

4. Using neighbor discovery we can set up a system for routing packets towards their destination along the most efficient route(s). Each node would only need to know the direction to send the packet on its next hop, through one of its neighbor nodes. This would require keeping a list (or table) of nodes and the neighbor node through which to route the packet.

5. An alternative approach to our neighbor discovery implementation would be to implement a new protocol, say PROTOCOL_NEIGHBORDISCOVERY. This would be a more explicit and elegant solution than our approach of sending a PING with a destination 0. Our implementation is fully functional since all nodes are numbered [1-n], with n being the number of nodes the broadcast domain. The recipient nodes respond by changing the source to their node id and respond with a PINGREPLY, thereby signaling they are a neighbor. TTL is limited to 1 so the packet is not re-transmitted beyond immediate neighbors. A con of this system is that it does rely on the node numbering scheme aforementioned. This scheme may not be guaranteed, nor may it be optimal as the network can hold one fewer node than it otherwise could have. A pro of the system is that it doesn't introduce the complexity

of a new protocol. It is not always realistic to simply implement a new protocol for every problem or task that arises, as much of network architecture is implemented at the hardware/firmware level making it very hard, or even impossible, to update.