How was midterm?

Will review last class in detail—TA gave class
IP bits and pieces
– ICMP
– Network Address Translation

Started Routing—General approach:
– model the network as a graph
– dynamically exchange information about the
distance vector of neighbors and learn the
shortest path to destinations

Two approaches:
– Simple Routing
– General Approach

First, Project 2 Bootstrap
– Link State—exchange your neighbor information
– Distance vector—exchange your full table with

You will have to use threads to implement your program
– one thread for control (routing, hello) and one for data (forwarding packets)
– use usleep, it sleeps only one thread (example below)
– I am guessing sleep works the same way
– use usleep; it gives you finer control over time (microseconds instead of seconds)

Sleeping/Implementing Timers
You will have to use threads to implement your program
– one thread for control (routing, hello) and one for data (forwarding)

Use usleep and it sleeps only one thread (example below)

SUNY-B

#11 1 Project Overview

1. Create topology (make up sockets, one for routing and one for data)

2. Implement routing (control) thread

   Track neighbors: Send hello packets to your direct neighbors, periodically send your routing table to your neighbors; if you are destination, receive it

   If not, check routing table for next hop and forward the packet

3. Implement your forwarding (data) thread

   When you receive a data packet, check destination

   If destination, receive it

   If not, check routing table for next hop and forward the packet

   Need to include <unistd.h>

   Example

   ```c
   # include <pthread.h>
   # include <unistd.h>
   # include <stdio.h>

   void *slowthread(void *i) {
     int j=0;
     while(++j) {
       printf("Hi%d from slowthread\n",j);
       usleep(5000000);
     }
   }

   void *fastthread(void *i) {
     int j=0;
     while(++j) {
       printf("Hi%d from fastthread\n",j);
       usleep(1000000);
     }
   }

   main() {
     pthread_t *thread;
     int dummy;
     pthread_create(thread,NULL,slowthread,(void *) &dummy);
     fastthread((void *) &dummy);
     pthread_exit(NULL);
   }
   ```

   SUNY-B

   #11 2
Recall Overall structure of Program

- Two threads, one for routing/control and one for forwarding/data
- Routing thread: (1) Exchanges hello messages; (2) Exchanges routing messages; (3) Builds routing table; (4) Interacts with control client (optional)
- Data thread: (1) Receives incoming data packets; (2) Forwards it if we are not destination according to routing table; (3) If you do not implement control client, should with low probability generate packets to random destinations

Data Thread – Event Driven

1. In a loop, send out any locally generated packets
2. Wait for incoming packets (use select, with a reasonable timeout value, say 0.1 sec)
3. If you receive an incoming data packet, check if you need to forward it or receive it locally and do so

Routing Thread – Time Driven

- Also runs in a loop
- Use usleep with a suitable period (say 0.1 sec) to control loop interval
- Things here happen periodically (e.g., should send hello packets every \( n \) seconds); keep a count for every event in terms of the loop granularity (a count of 50 will give you 5 seconds if your period is 0.1 seconds)
- Every loop:
  1. Check if you need to generate control packets (hello or routing) based on their count expiring
  2. Check for control packets (hello, or routing) and update your routing table accordingly – Behavior will vary depending on whether you use distance vector or link state...
Hello Packet

- Hello protocol simply keeps hosts aware of their neighbors
  - Each host should have a data structure (array or linked list) of its immediate neighbors
  - Everytime they receive a hello packet from them, they refresh the TTL entry on the neighbor list (lets say starting value is 1000 timer pulses)
  - Everytime the routing thread wakes up reduce TTL by 1, if it reaches 0 remove the neighbor
  - If a link goes down, we stop receiving packets
- What should the packet format be? Hello Opcode and Source Id is enough

Packets and Bits

- Three types of packets:
  - Data packets do the work of carrying the data.
  - Hello packets (or heartbeat packets) keep neighbors informed that the link between them exists
  - Routing packets, exchange routing info according to your routing protocol
  - For the optional client part, you have other control packets
- Data packets are received on the data port, everything else on the routing port

Routing Packet

- Depends on your routing protocol
  - Distance Vector: should include your full routing table
    - Example: [Opcode—Source—host1—Cost1—host2—cost2—...]
  - Link State: Send only your neighbor list
    - Example: [Opcode—Source—SequenceNumber—Neighbor1— Neighbor2—...]
    - Note that you have to flood this (send it to all other neighbors) the first time you receive this sequence number
- When your receive a routing packet, you should update your routing table according to your protocol

Data Packet Format

- Should have fields for:
  - Packet Id, so that we can track it
  - Source and Destination Id
  - Previous hop (this allows us to tell what link it came on since all of the links share the same protocol Id (necessary to implement upper layer protocols used in bonus parts)
  - Time to Live (TTL)
  - Data
- Like project 1, you can use memcpy to assemble/disassemble packets in character buffers to send them over the optional client port.

Hello Packet

- Each host should have a data structure (array or linked list) of its immediate neighbors
- Everytime they receive a hello packet from them, they refresh the TTL entry on the neighbor list (lets say starting value is 1000 timer pulses)
- Everytime the routing thread wakes up reduce TTL by 1, if it reaches 0 remove the neighbor
- If a link goes down, we stop receiving packets

Routing Packet

- Depends on your routing protocol
  - Distance Vector: should include your full routing table
    - Example: [Opcode—Source—host1—Cost1—host2—cost2—...]
  - Link State: Send only your neighbor list
    - Example: [Opcode—Source—SequenceNumber—Neighbor1— Neighbor2—...]
    - Note that you have to flood this (send it to all other neighbors) the first time you receive this sequence number
- When your receive a routing packet, you should update your routing table according to your protocol
Network Address Translation

Assign my machine a private number, say 192.168.0.100 (why private?)

NAT box has a real IP number, say 128.226.123.110

1. A connection from my machine to yahoo.com, connects through NAT box, port x
2. NAT box connects to yahoo.com from its port y, it records x to y mapping
3. Any packet from my machine to yahoo, the source IP address and portnum is replaced with the NAT box IP address and portnum
4. Any packet from yahoo back to my machine comes to NAT box port y. NAT box figures out true destination from mapping; it replaces destination address and portnum with my machine address and portnum, and sends it on private network

Distance Vector Example

Distance Vector Example (2)

Last Time – Distance Vector

- Every node keeps a routing entry for every other node in the network
  - Entry includes Next hop, and cost
- Periodically exchange the full table with your neighbors
- Update paths if you find a shorter path
- Problem: information is learned second hand; updates take a long time to propagate
  - Loops and suboptimal paths possible; e.g., count to infinity problem
  - We discussed solutions last time
- The RIP protocol (unix routed daemon) uses DV
Count to Infinity (1)

Count to Infinity (2)

Distance Vector Example (3)

Loss of Route

- What happens if the link between B and D is broken?
RIP Problems

- Slow to converge – one hop every update
- Infinity too small
  - Any problems making it larger?
- Routing metric is number of hops
- Utilizes only a single path even when multiple are available

Count to Infinity

- When is the problem resolved? What happens when the costs reach 10?
- What happens if C crashes?
- Is there anyway to avoid this problem? Not really; some heuristics
  - Idea: Split horizon – don't tell the route to the node you learned it from
    - Does it work?
  - Idea: pick infinity to be a small number (e.g., 16 hops)

Distance Vector Discussion

- RIP/DV problems
  - Infinity too small
  - Routing metric is number of hops
  - Utilizes only a single path even when multiple are available

- Problem: updates take a long time to propagate
  - Split horizon – don’t send an update to a node next hop if you heard the route from it (i.e., if it is the
    - Problem: state updates that contain wrong
      - Split horizon difficult to solve. Some heuristics
        - Always send updates that contain wrong information; if you think you can’t propagate to infinity, send
          Problem: sometimes it does not converge – count to
      - Sometimes it does not converge – count to

Split Horizon not Perfect

- Idea: Path vectors – include the full path (or a portion of it), detect loops
**Link State Routing**

- **Idea**: Send updates to *all nodes in the network* with information about *only your immediate neighbors*
  - The update consists of a Link State Packet (LSP) which holds
    * Source node id
    * Cost of the link to each directly connected node
    * A sequence number (SEQNO)
    * A time to live (TTL)
  - How to send the packet to all nodes in the network?
- Each node has the full topology of the network, and can carry out shortest path computation locally to find optimal paths

**Reliable Flooding**

- Each node forwards the packet first time it sees to all outgoing links
- Store most recent LSP from each node
- Generate a new LSP periodically; increment its SEQNO
- Decrement TTL of each stored LSP; delete if it reaches 0

**Route Calculation**

\[
M = \{s\}; \\
\text{for each } n \text{ in } (N - \{s\}) \\
C(n) = l(s,n); \\
\text{while } (N \neq M) \\
\quad M = M + \{w\} \text{ such that } C(w) \text{ is minimum } \\
\quad \text{of } c(x) \text{ for all } x \text{ in } (N-M); \\
\quad \text{for each } n \text{ in } (N-M) \\
\quad \quad C(n) = \text{Min}(C(n), \ C(w) + l(w,n)); \\
\]

- Classic Dijkstra's shortest path algorithm
- \(N\) denotes the set of nodes in the graph
- \(l(i,j)\) denotes non-negative cost for edge \((i,j)\)
- \(s\) is the node carrying out the computation
- \(M\) is the set of "visited" nodes
- \(C(n)\) is the cost of the path from \(s\) to \(n\)

**Forward Search Algorithm**

1. Initialize Confirmed with entry for me; cost 0
2. Select the node just added to Confirmed; call it next
3. For each neighbor of next, calculate the cost to visit the neighbor (the cost from me to next, and from next to the neighbor)
4. If the neighbor is neither in confirmed or tentative, add (neighbor, cost, nexthop) to tentative, where nexthop is the direction to reach next
5. If the neighbor is in tentative and the cost is less than its current cost, replace the tuple with (neighbor, cost, next)
6. If tentative is empty, stop, otherwise select the node with the lowest cost and add it to confirmed, go to step 2
Determining Link Cost

- How should the cost of a link be determined?
  - Why not just unity cost/use number of hops?
- Original ARPANET metric – number of packets queued on a link
  - This way, congested links get a very high cost
  - Any reservations?
- Second ARPANET metric was developed to take into account latency and bandwidth
  - Delay = (DepartTime - ArrivalTime) + TransmissionTime + Latency
  - transmission time/latency deduced when link layer ack was received

Third Measure

- New metric (routing units)
  - 225
  - 140
  - 90
  - 75
  - 60
  - 30
  - 25%
  - 50%
  - 75%
  - 100%
  - 9.6-Kbps satellite link
  - 9.6-Kbps terrestrial link
  - 56-Kbps satellite link
  - 56-Kbps terrestrial link

Problem: Oscillation
- As queue length became high, the cost became very high and connections were routed away from the link. The idle link advertised the low cost, attracting back all traffic.

Solution: Base cost on utilization, compress the range of the cost to avoid circuitous routes.

Forward Search Algorithm

- Track the operation of link state in this network; all edges cost 1
Link State Example – OSPF

- Neighbors are discovered using hello messages
- Routing messages are authenticated
- Measure/request cost to/from each neighbor
- Transmit routing information to all other routers (reliable flooding)
- Shortest Path based on Dijkstra’s algorithm, with a modification to obtain all shortest paths to a destination (not just one)
  - Packets are routed on all shortest paths to avoid congestion
- Three types of nodes
  - Routers
  - Networks
  - Hosts
- A router is elected for each network to report its LSP

Example from OSPF RFC

Note that the costs a connection can be different in each direction

Each router constructs its shortest path tree

Example shows the tree for router 6

Neither approach scales gracefully
- Must reduce size of stored and exchanged information
  - Use Autonomous Systems
    - Each autonomous system routed independently using an interior routing protocol (Intra-domain). Examples: RIP, OSPF
    - Routing across autonomous systems using an exterior routing protocol (Inter-domain). Examples: EGP, BGP
OSPF provides internal hierarchy to deal with a large autonomous area
- Connections across different areas pass through the backbone
- Routers are classified depending on their location. They take part in routing among their peers
- Three types of routes
  - Intra-area routes (within area)
  - Inter-area route (area1 – backbone – area2)
  - Inter-AS routes

Note R11, it is connected to two areas, but not to backbone

Link State Advertisements
- Router LSA
  - Describes the states of the router interfaces; flooded within the area
- Network LSA
  - Describes the states of LANs (not point to point links)
  - Flooded in the area by designated router
- Summary-LSA
  - Routes to destinations outside area but inside AS
  - Flooded in the area by the area/backbone router
- AS external LSA
  - Describes a route to a destination outside the AS (can be default)
  - Flooded through AS by boundary routers
- Can we do better?