1 Lecture 02-06-2020

Refresher Content

- Each router is equipped with subnet interfaces; using subnets allows routers to move traffic between routers easily.
  - Routers are equipped with link cards, which allow the router to decode incoming traffic and queue packets to be sent out.
  - These link cards consist of three parts: line termination, adaptation to the data link layer, and a queue.
- Routers have a routing algorithm (hence the name router") which allows them to correctly send packets to the right destinations.
- The routing decision is done in two portions: the data plane and the control plane
  - The data plane decides to which router/network/LAN the packet should go next- a simple decision done based on a forwarding table stored in the router. (Think about an individual step of a roadtrip- from one gas station to another.)
    - This forwarding table is basically just a map which maps one subnet (destination) to an interface of the router.
    - a special piece of hardware known as a TCAM does this math really quickly, so the table is optimized for TCAM usage- basically, the most specific subnets (most known bits) are analyzed first.
  - The control plane does some more complicated math, figuring out, in general, the steps required to get the packet to its final destination. (Think about planning out the entire roadtrip from home- which highways to take, etc.)

Router Buses

- How can we get from any arbitrary input interface $input_n$ to any arbitrary output interface $output_n$?
- It turns out that we can solve this problem rather easily- just connect each input to a bus, and connect that bus to each output.
  - This solution is brutally simple and effective, but won't scale up to situations where we need to process multiple input and output packets at a time
- So, how can we make a workable, scalable solution?
- One option is to route the packets using software, storing input and output locations in memory

  - This solution ends up being very slow, because going into memory tends to take hundreds of cycles longer than smaller, simpler ALU operations.

- Another solution is to route the packets using a special configuration of hardware known as the crossbar switch.
  - This is basically a grid of buses, where each intersection is linked to a tri-state buffer: by putting a Hi signal on the Enabled setting on the buffer, one can redirect each of the inputs into the grid towards each of the outputs of the grid.
  - Inefficiency props up when two inputs would cross with each other. For instance, if Input 1 maps to Output 4, but Input 4 maps to Output 1, the two signals would not be able to travel independently on the crossbar.
  - At best, this configuration allows for each input to map to each output, all at once. At worst, this configuration is just as bad as a shared bus.

- Additionally, some systems exist which do similar hardware gymnastics, like the Banyan Tree: with this configuration, an input can cross through a series of switchable gates to reach its output destination. This solution theoretically allows for the same best-case and worst-case scenarios as the crossbar while allowing for inputs to cross with each other, but it's more complicated to set up and likely doesn't have the same consistency for reaching an ideal state.

Queueing and Dropped Packets

- How should a router handle a situation where it needs to transport two packets across the same interface at once?

- The working solution for any router is to choose one packet to send, and "block" the other packet from being sent.

- What, then, does the router do with the "blocked" packet? There are various solutions to this problem.
  - It's always possible to simply drop all packets which aren't immediately sent. This is typically a problem, but might not necessarily be out of the picture in some situations.
  - Traditionally, routers (and their link cards) use a Queue data structure to store packets and set them up to be sent out as the interface frees up.
  - Using a Queue requires a scheduling algorithm- most queues need some semblance of priority so that important packets get through in a timely manner.
    * FIFO: The simplest Queue scheduling algorithm- the packet that's been in the queue the longest is the packet that's sent out the soonest.

    * Priority scheduling: Split your queue into two separate queue data structures (one fast and one slow), and only pull from the " slow" queue if the "fast" queue is empty.
        · There are variants of this which allow for more "weighted" switching between queues- for instance, every fourth packet must always come from the "slow" queue to prevent starving a queue.
    * Round Robin: Still two (or *N*) queues, but none of them is the "fast" one. Switch strictly between queues, pulling one from each, circularly. This allows packets which pick uncommon queues to get the "priority" treatment.
        · It's possible to get a "weighted" version of this queue as well, taking a certain number of packets from each queue in order. This means that one router chip might always choose to pull from one queue ten times as much as it pulls from the other, artificially making a priority queue.

Internet Network Layer

- Recall that routers need to figure out on a high level which path a packet should take to get to its destination
- This is a particularly difficult problem because they don't start with much more knowledge than the cost of traveling to their neighbors.
- This, for example, makes it infeasible to apply Dijkstra's Algorithm to the graph of all routers in a network. Instead, we can come up with a procedure which allows routers to figure out nearly ideal paths without a complete set of information.
- When applied across the whole set of routers, this is called the Distance-Vector. On each individual router, we use the Bellman-Ford algorithm.
    - In short, Bellman-Ford begins with the knowledge that the cost of the shortest path from the starting point to the ending point is equal to the cost of the path from the starting point to another point plus the cost of that other point's travel to the ending point.
    - In other words, if we can figure out how close each of our neighbors is to our destination, we can figure out to which neighbor we should send our packet.
- Each router starts with knowledge of its Distance Vector: the costs of travel from this router to each of its neighbors (and itself).
- Each router creates a table mapping starting routers and ending routers.
    1. This table starts out like an Identity Matrix, except each of the zeros in the Identity Matrix are replaced with Infinity, and each of the ones are replaced with zeros.

2. This makes sense because it should not take any time for a packet to be sent from one router to itself (it's already there), and without any knowledge of any other router's info, no numerical value other than Infinity would be useful as a placeholder in the table.
3. The router then populates the row for itself with its distance vector.

- Then, each router follows this process:

1. Each router shares its Distance Vector with its neighbors.
2. Each router, upon receiving a Distance Vector from another router, adds that vector to its table (updating the row if necessary).
3. Each router performs the Bellman-Ford algorithm to ensure that its distance vector is updated to show the smallest costs of travel from itself to its neighbors.
4. If performing the Bellman-Ford algorithm changed the router's Distance Vector, then it shares its updated Distance Vector with each of its neighbors.

- With this process in place, a router can know as soon as it receives a packet where it should send the packet to have it arrive at its destination quickly. Basically, a router can simply use all the info it's gained on its table to figure out where the packet should go next.

- Note: This process assumes that all routers are sending authentic Distance Vectors, and that the cost of traveling from one router to another is relatively constant. If either of these two things aren't true, then the shortest path calculated by this algorithm will either not be correct for long or will never be correct.