Lecture 9

Gidon Rosalki

2025-12-14

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_networking`

1 Interconnecting IP subnets

When considering layer 3, we want to connect LANs, or IP subnets. For our uses, they are separated by routers. There are 2 key characteristics to a network:

1. Topology: The physical interconnection structure of the network
2. Routing algorithm: This restricts the set of paths that the messages can follow (like STP)

1.1 Topology

This is how the components are connected. It has some important properties:

- Diameter: The maximum distance between any two nodes in teh network (measured in hop count / number of links)
- Nodal degree: How many links connect to a node
- Bisection bandwidth: The lowest bandwidth between half the nodes, and the other half of the nodes, across all such partitions (consider MAX CUT from algorithms)

We want to measure what makes a *good* topology. As with everything in life, this is a loaded question, since we need to define what is good. One might think that bisection bandwidth would be a good measure, but consider a network where $\frac{2}{3}$rds of the nodes are maximally connected, and the other $\frac{1}{3}$rd are maximally connected, and they only have a single link between the two. This would not be great, but the bisection bandwidth would not consider that, since it talks of halves.
Instead, we may consider an expansion of a graph, and examine every partition:

$$\min_{S \subset V,\ 0<|S|\leq \frac{n}{2}} \left\{ \frac{\text{EdgesBetween}\,(S, V \setminus S)}{|S|} \right\}$$

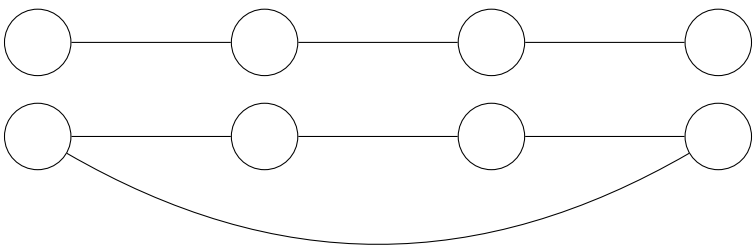
This is called the **edge expansion** of a graph. **Expanders** are graphs with a high edge expansion, i.e. close to $\frac{d}{2}$ for a d-regular graph.

There are two main tpes of topologies:

- **Explicit constructions**: Here the nodes are connected with some kind of pattern (so the graph has a structure). The nodes are identified by coordinates, and routing can usually be pre-determined by the coordinates of the nodes.
- **Non explicit constructions**: The nodes are connected arbitrarily. There is no structure to the graph, which makes it more extensible in comparison to a regular topology. These often use variations of shortest path routing.

1.1.1 Linear arrays and Rings

Consider the following two topologies, linear array, and a ring:



For the linear array topology:

- Diameter = 3 (distance between the two furthest is 3 links)
- Nodal degree = 2
- Bisection bandwidth = 1

and the ring:

- Diameter = 2
- Nodal degree = 2
- Bisection bandwidth = 2

To describe a linear array, and a ring, in an array nodes will be numbered from 0 to $n-1$, with i connencted to $i+1$ for $i < n-1$. In a ring, nodes are numbered from $0, \ldots, n-1$ as well, but $\forall i$ i is connected to $i+1 \mod n$.

1.1.2 Hypercubes

If we consider instead hypercubes (4D cubes, or greater), then there are 2^n nodes, and each node is described by its binary representation. There is a link between two nodes with the hamming distance of 1. For an n dimensional hypercube, the diameter will be n, the nodal degree will be $n-1$, and the bisection bandwidth will be 2^{n-1}.

1.1.3 Trees

In a tree, each node has at most 2 children. They are of fixed degree, a diameter of $\log(n)$, and $O(1)$ bisection bandwidth. To route between nodes a, b, one goes up to their common ancestor, and then back down to the other node.

1.2 Routing

We focus for now on routing within a single organisation, called intradomain routing. We will discuss interdomain routing later on. We are effective discussing a network of routers, where the links represent IP subnets.
Upon receiving a packet, our routing algorithm needs to, very quickly, decide what to do with it, or in other words, decide on which link it should send the packet. In order to achieve this, there is often a forwarding table, where one matches between header values, and output links. Forwarding is performing this action, and routing is the engine that decides where to send each packet. So the engine's output is the forwarding table.

We often work with static link weights. Here we want to compute the shortest paths to IP subnets based on the link weights. These are configured by the network operator (we will revisit this), and through this, we determine the next hop router (the next router in the path) to every IP subnet.

We have 2 main routing schemes:

- Link state routing: Each router floods information to the network in order to learn the topology, and then each router applies Dijkstra's algorithm to compute the shortest paths
- Distance vector routing: This is an iterative process, that uses Bellman Ford (similar to STP)

There are 2 ways to classify routing algorthms:

- Global: All routers have the complete topology, and link cost information. These are solved with "link state" algorithms
- Decentralised: Each router only knows its **physically** connected neighbours, and the link costs to these neighbours. Solving this involves an iterative process of computation, and information exchange with neighbours. It is solved with "distance vector" algorithms.

1.2.1 Link state routing algorithm

Here we will discuss Dijkstra's algorithm. In the network topology, link costs are known to all nodes. This is accomplished via *link state broadcast*. All the nodes have the same information. It computes the paths of the lowest cost from one node (the source), to all other nodes. This creates the forwarding table for that node. This is an iterative algorithm, where after k iterations, we will know the lowest cost path to k destinations.
Let us define some notation:

- $c(x, y)$ - The link cost from node x, to node y. This will be infinite if there is no link
- $D(v)$: The current value of the cost of the path from the source, to the destination
- $p(v)$: The predecessor node along the path from the source, to v
- V': The set of nodes for whom we definitely know the least cost path

The algorithm will run as follows:

1. Initialisation: Set $V' = u$, and for all nodes v, if they are adjacent to the starting node u,

$$D(v) = c(u, v)$$

and set $D(v) = \infty$ otherwise

2. Loop until all nodes in V': Find a $w \notin V' : D(w)$ is a minimum. Add w into V', and update $D(v)$ for every v that is adjacent to w, and **not** in V':

$$D(v) = \min(D(v), D(w) + c(w, v))$$

The new cost to v is either the old cost to c, or the known shortest path cost to w, plus the cost from w to v.

A naïve implementation for n nodes will involve $\frac{n(n+1)}{2}$ comparisons, and will be $O(n^2)$. More efficient implementations are possible (see data structures course), to be performed in $O(n \log(n) + |E|)$. The function may oscillate between solutions, and not complete, if the link weights are set dynamically. This can occur when the link cost is the amount of carried traffic (for example).

1.2.2 Distance vector algorithm

This makes use of Bellman Ford Equation (dynamic programming). Let us define

$$d_x(y) = \text{Lowest cost path cost from } x \text{ to } y$$

Then,

$$d_x(y) = \min_v \{c(x, v) + d_v(y)\}$$

Where the minimum is taken over all the neighbours v of x.

The basic idea is that every so often, each node will send its own distance vector estimate to its neighbours. When a node x receives this estimate from a neighbour, then it updates its own estimate using BF:

$$\forall y \in n \; d_x(y) \leftarrow \min_v \{c(x, v) + d_v(y)\}$$

So then, $d_x(y)$ converges to the actual cost.

When a link cost changes, then a node detects the local kink cost change, updates the routing information and recalculates the distance vector. If the DV changes, then it notifies its neighbours.
So, good news will travel fast (links getting faster will propagate very quickly), but bad news (link lengths getting longer) will propagate slowly.

1.2.3 Comparison of LS and DV

Complexity:

- LS with n nodes, E links, and $O(nE)$ messages sent, each node will send a broadcast message, flooded to all links
- DV: Exchange is only between neighbours

Convergence speed:

- LS: $O(n^2)$, may have oscillations
- DV: Varies, may be routing loops, and may have count to infinity problem (bad news propagates slowly)

Robustness (router malfunction):

- LS: Nodes can advertise the incorrect *link* cost, and each node computes only its own table
- DV: Nodes can advertise the incorrect *path* cost, and each node's tables are used by others (thus propagating the error through the network, blackholing traffic)

1.2.4 Real world examples

RIP - Routing Information Protocol. This is a distance vector algorithm, used by BSD-Unix in 1982. The distance metric is the number of hops.
OSPF - Open Shortest Path First. This is an open, publicly available protocol, using the LS algorithm. The OSPF advertisement carries one entry per neighbour router, and advertisements are disseminated to the entire network (via flooding). Messages are carried directly over IP, rather than over TCP, or UDP.

1.3 ARPAnet routing

ARPAnet was a wide area network that predated the internet. It originally (1969) used shortest path routing, with a dynamic setting of link weights. They were set to the instantaneous queue length, plus some constant. Each node updated its distance computation periodically.
This had some problems. There were protocol oscillations, and a high protocol overhead. Additionally, a long path would appear better than a congested path, resulting in an inefficient use of resources.

In 1979 there was a new routing protocol, which averaged the link weight over time in order to reduce fluctuations, and the frequency of updates (nicely handling some of the overhead issues). In 1987 it was revised, where traffic was shed gradually in order to prevent overreactions to congestions. Additionally, link weights were given upper bounds, in order to avoid excessively long paths.

1.4 Traffic management

Recall layering. We will cover traffic engineering, which is done in the network layer (L3), and handled within organisations, by optimising static link weights. We will soon discuss congestion control, which is done in the transport layer (L4), through protocols like TCP and UDP.

Traffic engineering is tuning the routing protocol configuration to optimise network performance (often by changing the static weights). This is dne through:

- **Measurement**: Measuring the topology (as done at the beginning of the lecture), and the traffic pattern (passively inspecting the traffic)
- **Network wide models**: Making representations of the topology, and traffic
- **Network optimisation**: Algorithms to find good configurations, and operational experience (nothing beats experience) to identify constraints

1.4.1 Theory - Flow optimisation

Recall the max flow problem from algorithms. Given an undirected, weighted graph, with a source vertex, and a target vertex, output the maximum flow from s to t. This may be done through max flow - min cut.
We need to adapt this to *multicommodity flow*, since every node can send to every node. We will remove s, t, and add in the demand matrix $D = \{d_{ij}\}$. There are a few methods to solve this problem:

- Maximum multicommodity flow: maximise the total amount of sent traffic
- Minimise congestion: Minimise the load on the most congested edge
- Fairness: Distribute the traffic as evenly as possible
- Etc.

Multicommodity optimises the routes from sources to targets, and how traffic is split between routes. We need to ask if IP routing optimises this.
To optimise the (static) link weights one begins by computing the shortest paths to other routers based on the link weights to determine the next hop to every other router. The link weights are configured by the network operator.

1.4.2 ECMP

In Equal Cost Multipath (ECMP), each router computes the shortest paths to each other router, based on the configured link weights, and splits traffic designed for said router evenly between all the nodes between them.

So, our objective given the multicommodity input is to output link weights such that ECMP flow is the optimal solution (with respect to the specific objective function). We have used the word “optimal” here, which assumes that there are a set of link weights such that the ECMP flow is optimal. We need to consider if this is in fact always the case, and if it is not, if there is always a set of link weights that approximate the optimal solution.

In short, we cannot always get the optimal solution, sometimes there is not one. Instead, we will change our objective to the ECMP flow that is “closest” to a specific objective function. Turns out that this problem is NP-hard, even for simple objective functions. As a result, network operators use (non optimal, but sufficiently good in some real life environments) heuristics.