CMSC417 Computer Networks

Michael Li

Contents

1 Foundations

1.1 Requirements

Requirement 1: Scalable Connectivity: Network should be scalable and should be able to connect to connect to a lot of devices. At the lowest level, **nodes** (computers) are connected by **links** (e.g. wire).

- **Point-to-point**: link is only shared between 1 pair of nodes
- **Multiple-access**: link is shared between several nodes

Indirect connectivity: can be achieved in multiple ways:

- **Switched network**: uses nodes that forward data received from one link and out the other link.
  - **Circuit switched**: establishes a stream of bits
  - **Packet switched**: nodes send blocks of data in the form of **packets** and use **store-and-forward** where the node has to receive entire packet, storing it in memory, before forwarding the entire packet. Here the **switches** implement the network and the **hosts** use the network
- **Internetwork (internet)**: set of independent networks connected by **routers** that are nodes that act like switches between networks.

In order to provide **host-to-host** connectivity, each node is assigned an **address** that allows other nodes to communicate with the node. Switches and routers use this address to **route** messages to their destination

Requirement 2: Efficiency
Multiplexing: system resources shared among multiple users. Achieved by using

- **Synchronous Time Division Multiplexing (STDM)**: divide time into equally sized chunks and round-robin each flow
- **Frequency Division Multiplexing (FDM)**: transmit each flow at different frequencies
- STDM and FDM both have issues in that links may be idle during a given time interval or at a particular frequency. Also, both do not support adding new flows easily.
- **Statistical Multiplexing**: links are shared on a time-basis but is on demand, rather than a predetermined time slot. Also limits the size of packets sent over to ensure data is not lost.

With multiplexing, switches may need to buffer packets if the input rate $>$ output rate. This can lead to **congestion** where the buffer is full so it needs to drop packets

Requirement 3: Support for Common Services: networks provide **logical channels** that application processes use to communicate with each other. These channels provide a set of services and are typically imlpemented on end hosts to keep links simple

Requirement 4: Reliability: need to consider 3 types of errors

- **bit error**: bit is flipped in the physical link
- **packet loss**: packet is dropped or uncorrectable packet is discarded
- **physical crash**: e.g. network shutsdown

Requirement 5: Managbility: system needs to be scalable (should be able to support more devices)

1.2 Network Architecture

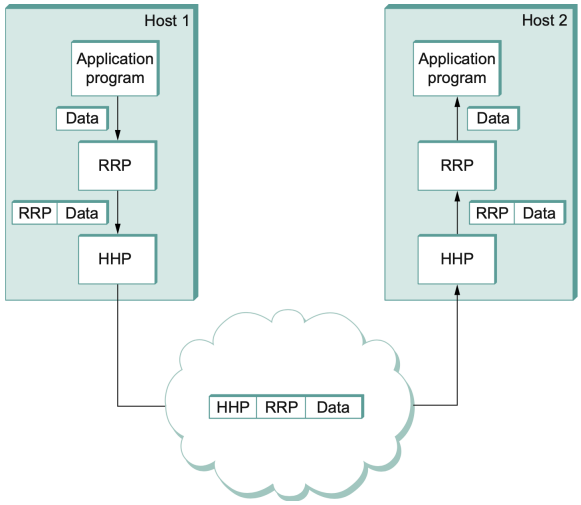
Network Architecture is a blueprint that guides the design and implementation of networks. Uses **layering** and **abstraction** to capture the important aspects of a system. Services at higher **protocols** (layers) are built using the services of the lower protocols. These protocols define 2 different interfaces:

- **service interface**: defines operations local objects can perform on the protocol (e.g. sending and receiving messages)
- **peer interface**: defines the form and meaning of messages exchanged between protocol peers

Peer-to-peer interaction is indirect; each protocol communicates with its peer by passing messages to lower level protocols that deliver the messages to its peer. This ends up creating a **protocol stack**. On the peer side, protocol layers are popped until we reach the original target.

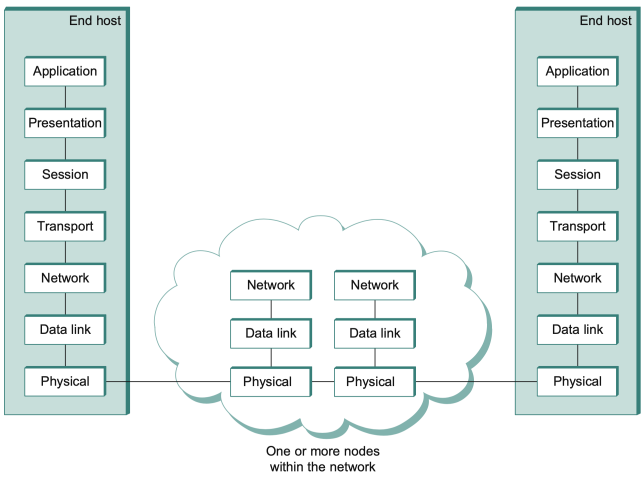
Encapsulation is performed at each protocol level; **headers** are attached to the **payload** (rest of the message) and instruct the peer on how to handle the message (i.e. how the payload is passed to the next higher level in the peer).

A **demux key** is attached to the header, allowing protocols to demultiplex messages to the correct application on the destination host



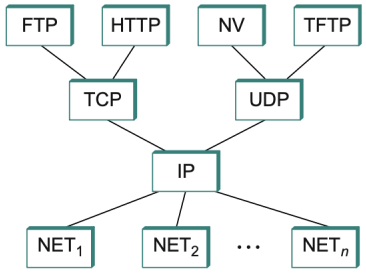
Open Systems Interconnection (OSI): partitions network functionality into 7 layers

- **Physical layer**: handles transmission of raw bits over communication link
- **Data link layer**: aggregates bits into **frames**
- **Network layer**: routes packets to nodes within packet-switched network. Unit of data exchanged is called **packet**
- **Transport layer**: implements process-to-process channels that exchange messages. Unit of data here is called **messages**
- **Session layer**: ties potentially different transportation streams
- **Presentation layer**: concerned with the format of data being exchanged
- **Application layer**: handles high level protocols



Internet Architecture: 4 levels:

- **Network protocols**: lowest leevel implemented by a combination of hardware and software
- **Internet protocol (IP)**: supports multiple networking technologies into 1 internetwork
- **Transmission control protocol (TCP)** and **User Datagram Protocol (UDP)**: TCP provides reliable byte-stream channel and UDP provides unreliable datagram delivery channel
- **Application protocols**: enable interoperation of applications
- One main difference between Internet Architecture and OSI is that layering isn't prevelant for the former; apps can skip layers if desired



1.3 Implementing Network Software

Socket: the point where application attaches to the network. The interface defines how to create sockets, how to attach them to network, how to send and receive messages through sockets, and how to close sockets. To create a socket in TCP:

- invoke socket() to create socket
- for the server side, perform a **passive open** (prepared to accept connections but doesn't actually establish them) by invoking bind(), listen(), and accept()
- for the client side, perform an **active open** (says who it wants to communicate with) by invoking connect(). Once connected, can invoke send() and recv() to send and recieve messages from specified socket to a buffer

```
// domain: protocol family
// type: semantics of communication
// protocol: procol used
// returns a handle (identifier for socket)
int socket(int domain, int type, int protocol);

// binds created socket to a specific address
int bind(int socket, struct sockaddr *address, int addr len)
// listen defines how many connections can be pending on a socket
int listen(int socket, int backlog)
// accept carries out passive open
int accept(int socket, struct sockaddr *address, int *addr len)

// connect doesn't return until TCP has established a connection
int connect(int socket, struct sockaddr *address, int addr len)
// once connection is established, applications can send and recieve messages
int send(int socket, char *message, int msg len, int flags)
int recv(int socket, char *buffer, int buf len, int flags)
```

1.4 Network Performance

Performance measured by **bandwidth** (number of bits transmitted over time) and **latency** (how long it takes for message to travel)
Bankwidth can be broken down into a sum of

- speed of light propogation (distance/S)
- time to transmite a unit of data (size/bandwidth)
- queuing delays

Delay $\times$ **bandwidth product**: maximum number of bits that can travel in a pipe at once. Tells how many bits must be set before first bit arrives at receiver

2 Routing

2.1 Vocabulary

Forwarding (Data Plane): receiving a packet, looking up its destination on a table, and sending the packet in the appropriate direction.
Routing (Control Plane): process that builds **forwarding tables**.
Forwarding table: contains information regarding mappings from network prefix to an outgoing interface to aid in forwarding packet.
Routing table: contains mappings of network prefixes to next hops.
Routing domain: internetwork where all routers are under the same administrative control.

2.2 Network as a Graph

Entire network can be represented as a graph (nodes, network links as edges with cost). The goal of **routing** is to find the lowest-cost path between any 2 nodes. Algorithms that calculate the shortest path usually do so in a dynamic, distributed manner, but this can lead to some confusion when different paths thinking that they have the shortest path available. There are 2 main routing protocols: **Distance vector** and **Link state**.

2.2.1 Distance Vector

Each node constructs an array (vector) that contains the cost to all other nodes and distributes this vector to its immediate neighbor and the vector is updated based on which nodes can be reached by the neighbor node. Links that are down are assigned an infinite cost.

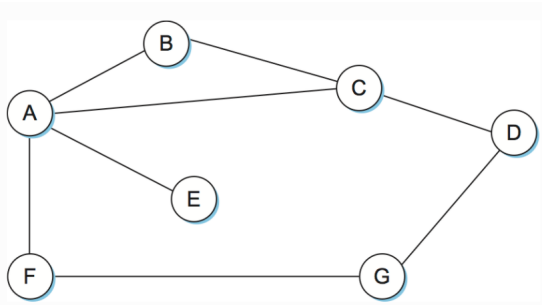


Figure 85. Distance-vector routing: an example network.

Table 16. Initial Distances Stored at Each Node (Global View).

	A	B	C	D	E	F	G
A	0	1	1	∞	1	1	∞
B	1	0	1	∞	∞	∞	∞
C	1	1	0	1	∞	∞	∞
D	∞	∞	1	0	∞	∞	1
E	1	∞	∞	∞	0	∞	∞
F	1	∞	∞	∞	∞	0	1
G	∞	∞	∞	1	∞	1	0

Routers exchange information (destination, cost) with immediate neighbors and the table is built from a sequence of exchanges.

End result is that each node has the same routing table so routing information is consistent (**convergent**). Note that no one node contains all information in this table, each node only knows about the content of its own routing table. There are 2 ways in which updates occur:

- **Periodic update**: each node automatically sends an update message, even if nothing has changed, to let other nodes know that it is still running and to allow other nodes check if the current route becomes unviable.
- **Triggered udpate**: occurs when a node notices a link failure or receives an update from one of its neightbors that causes 1 of the routes in the routing table to change. This then triggers an update to the node's other neighbors.

Count to infinity problem: example - from the graph above, if the link A to E goes down, A notifies its neighbors that its distance to E is infinity, but B and C claim that they have a distance of 2 away from E. This then leads to the conclusion that A is 3 away from E, and then B and C is 4 away, leading to a distance of infinity. Potential solutions include:

- Set a maximum (e.g. 16) to represent infinity. Once we reach that many hops, we claim that the distance is infinity. This solution is restrictive in that it isn't scalable.
- **Split horizon**: when a node sends updates, it does not send routes it learned from a specific neighbor to that same neighbor. However, this solution isn't scalable for loops involving more than 2 nodes.

Routing Informatoin Protocol (RIP): canonical example of routing protocol built on distance vector algorithm

2.2.2 Distance Vector Notations

$x.C(y)$ link cost from node x to node y. If they are not neighbors, value is ∞
$x.D(z \to y)$ distance known to x to travel from z, a neighbor of x, to y
$x.N$ set of neighbors to x
$\mathbb{N}$ set of all nodes
$x \to y : M$ node x sends a message M to y
At the start, routing table consists of values:

- if $y \in x.N \implies$ x.D(y $\to y) = x.C(y)$
- if $y \notin x.N \implies \forall v \in x.N, x.D(v \to y) = \infty$

Entries consist of:

- destination
- cost
- next hop

At the end, routing table consists of values:

- $\underbrace{x.D(z \to y)}_{z \in x.N} = x.C(z) + \underbrace{\min(z.D(v \to y))}_{v \in z.N}$

2.2.3 Link State

Since each node knows how to reach its neighbors, we can make it so that this information is known to each node to create a complete map of the network at each node. Relies on

- **Reliable flooding**: making sure that all nodes get the same copy of the link-state information from all the other nodes. Nodes send their link-state information (**link-state packet (LSP)**) to all of their neighbors who then forward to their neighbors until all nodes have been reached. LSPs contain
  - ID of node that created the LSP
  - list of connected neighbors of that node with the link cost to each one
  - sequence number
  - time to live for this packet

  First 2 items enable route calculation. Last 2 items make flooding more reliable (ensuring that each node has the most recent copy of the information since there might be multiple LSPs from different nodes traversing the network)

  How flooding works: consider a node X that received a copy of an LSP that originated from node Y. If X already has a stored copy an LSP from Y. If not, X stores the LSP. Otherwise X compares the new and old LSP sequence numbers. If the new LSP has a larger sequence (more recent) then it replaces the old LSP and X sends a copy of this LSP to all of its neighbors, except the neighbor the that sent the new LSP, who do the same check.

  LSPs are generated either on a **periodic** basis or **topology change** (a failed link is detected), similar to that of Distance Vectors.

  Optimizations for LSPs:

  - set long periodic timers for updates (reduces overhead for creating LSPs unless absolutely necessary)
  - Each time a node generates a new LSP, it increments the sequence by 1. If the node crashes, it starts at 0 when it comes back up. This ensures that newer information replaces older information
  - LSPs carry a time to live (ensures that old information is eventually removed). A node always decrements the time to live of a newly received LSP before flooding its neighbors.

- **Route Calculation**: once a node has a copy of the LSP from every other node, it can compute a graph of the entire network and can determine the best route, using Dijkstra's.

Link-state routing stabilizes quickly and responds rapidly to topology changes. However, each node ends up being very large with all of the LSPs it must hold.

2.2.4 Comparison Distance Vector and Link-State Routing

Both are distributed routing algorithms but in distance vector, each node only communicates with its direct neighbor and tells the neighbor everything it knows. In link-state, each node communicates with all other nodes but only tells them what it knows for sure.

2.3 Metrics

Measuring cost of link was done in a variety of ways

- Number of packets queued for waiting at each link (larger queue is assigned a larger cost weight). Not a good measurement since this focuses on queue size rather than distance and time.
- Consider both bandwidth and latency. Each packet is given a timestamp ArrivalTime and DepartTime and Delay is calculated as (DepartTime - ArrivalTime) + TransmissionTime + Latency, at the other node. Not a good measurement since links will become idle when they have a high load.
- Compress dynamic range of metric to smooth the variation of the metric over time. Smoothing was achieved by limiting how much the metric could change from 1 measurement cycle to the next and by calculating the average amount of utilization for a link.