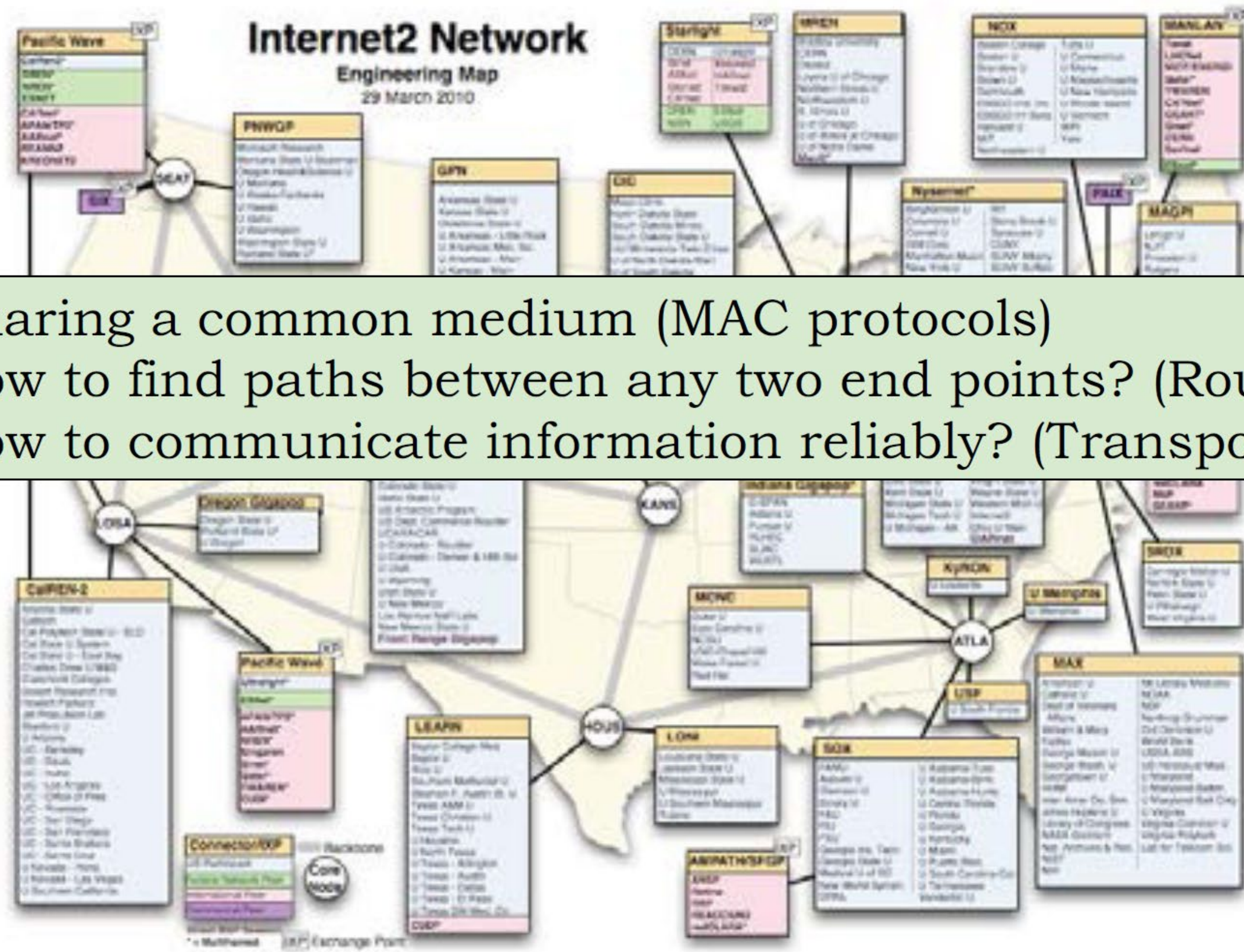
FUNDAMENTALS OF INFORMATION SCIENCE:
PART 3: NETWORKS

Shandong University
2025 Spring

Lecture 3.1: Communication Networks

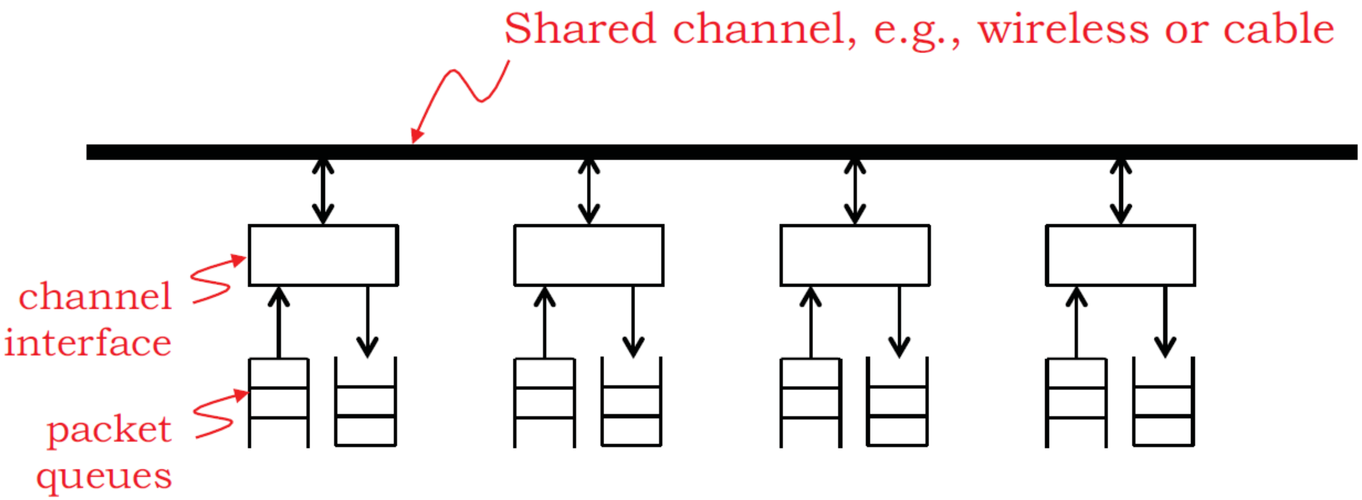
Circuit vs. Packet Switching

Circuit switching	Packet Switching
Guaranteed rate	No guarantees (best effort)
Link capacity wasted if data is bursty	More efficient
Before sending data establishes a path	Send data immediately
All data in a single flow follow one path	Different packets might follow different paths
No reordering; constant delay; no dropped packets	Packets may be reordered, delayed, or dropped



Sharing a common medium (MAC protocols)
How to find paths between any two end points? (Routing)
How to communicate information reliably? (Transport)

Shared Communication Channels



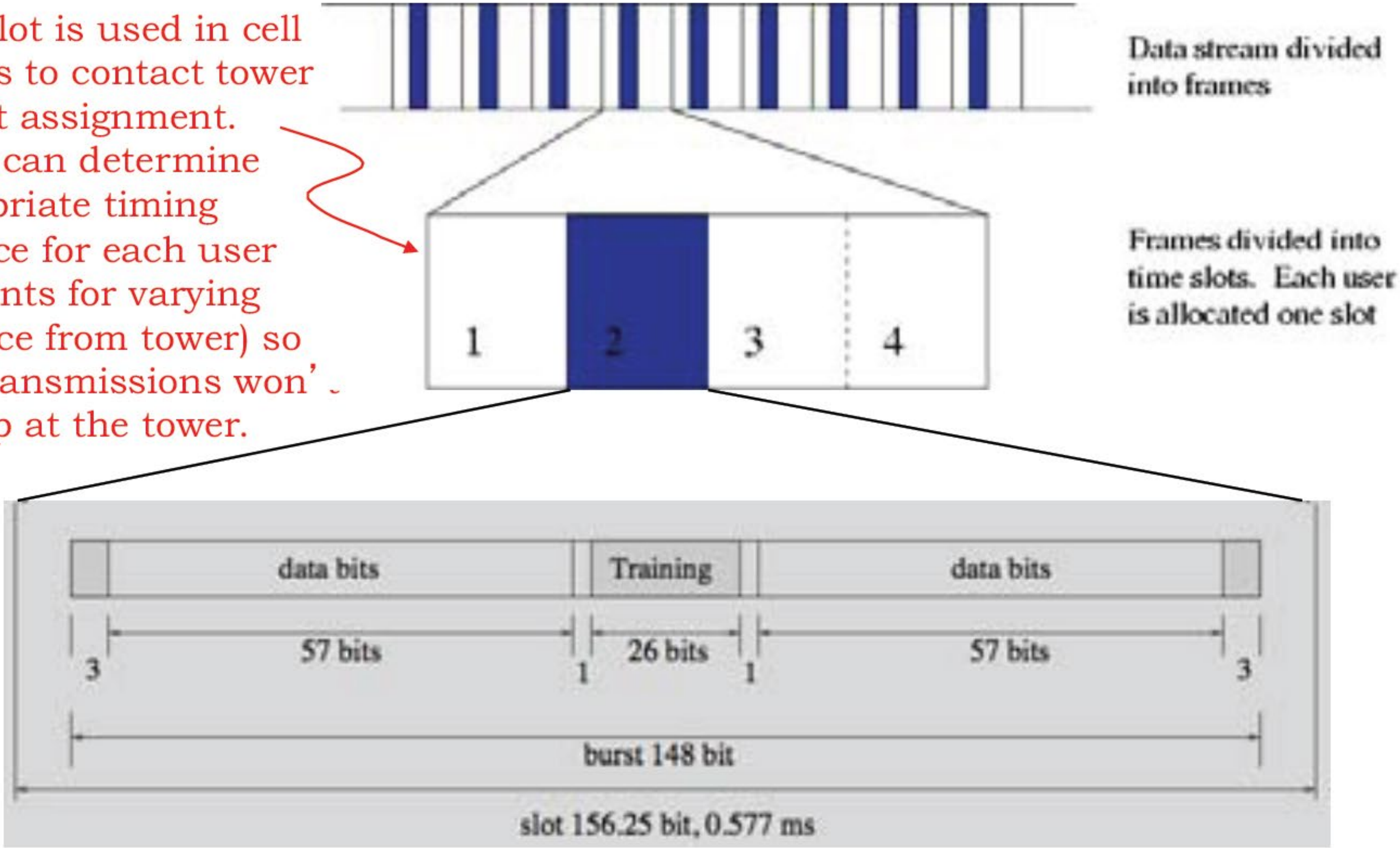
- Basic idea in its simplest form: avoid collisions between transmitters – collision occurs if transmissions are concurrent

- Wanted: a communications protocol ("rules of engagement") that ensures "good performance"

- Nodes may all hear each other perfectly, or not at all, or partially

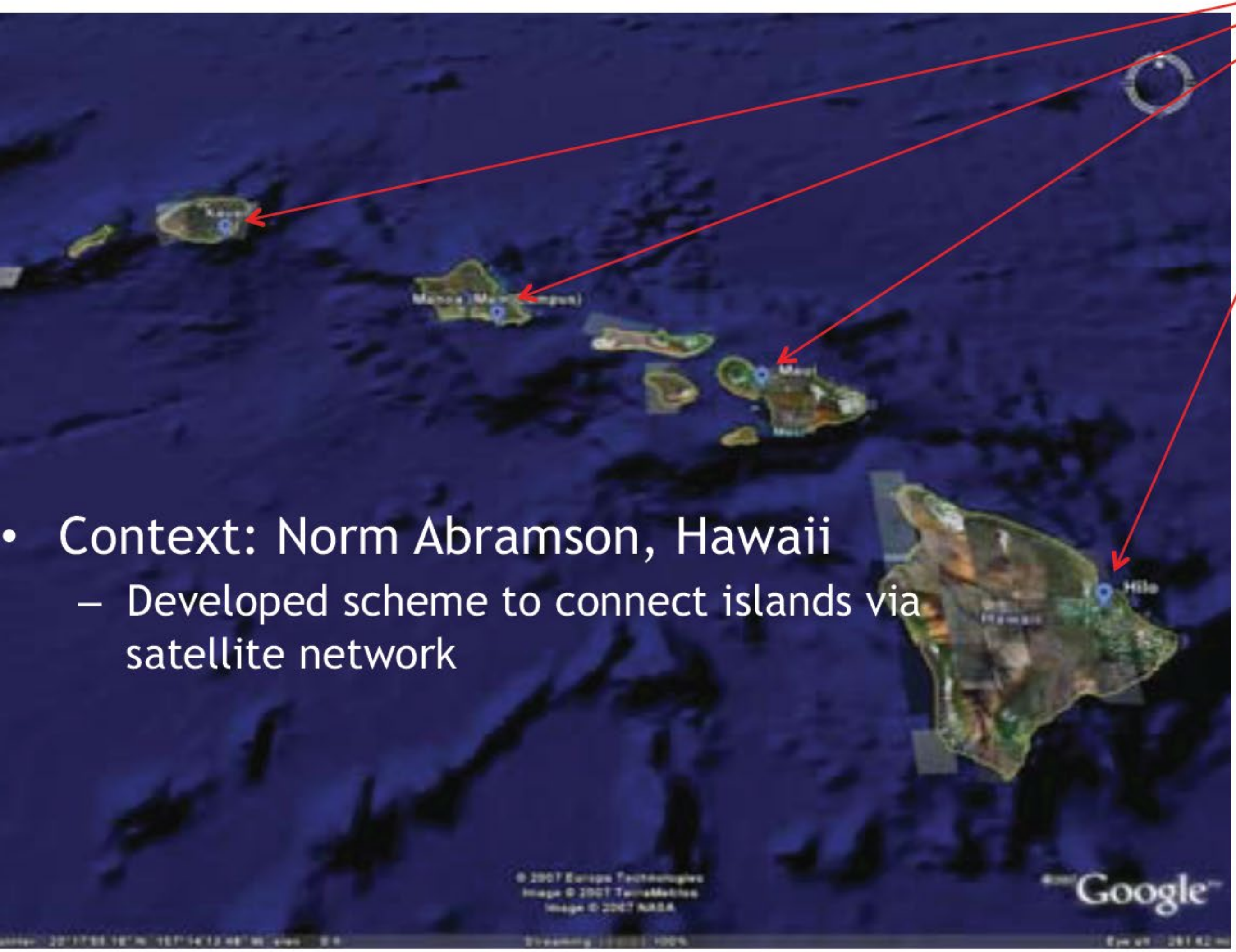
TDMA for GSM Phones

First slot is used in cell phones to contact tower for slot assignment. Tower can determine appropriate timing advance for each user (accounts for varying distance from tower) so that transmissions won't overlap at the tower.



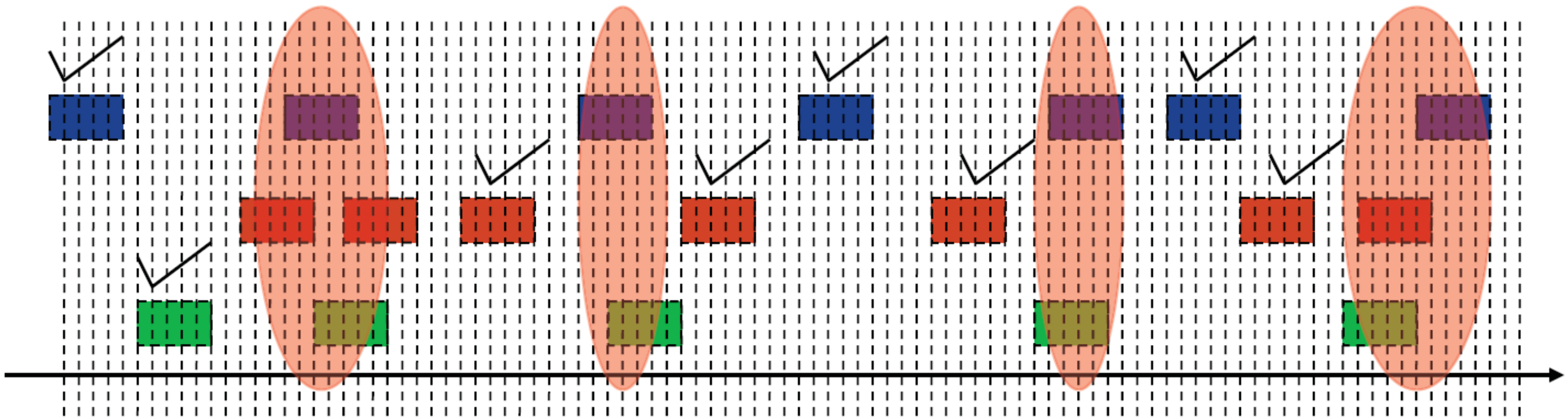
The Aloha Protocol

Satellite clip art is in the public domain.



- Context: Norm Abramson, Hawaii
  - Developed scheme to connect islands via satellite network

Aloha in Pictures: Collisions

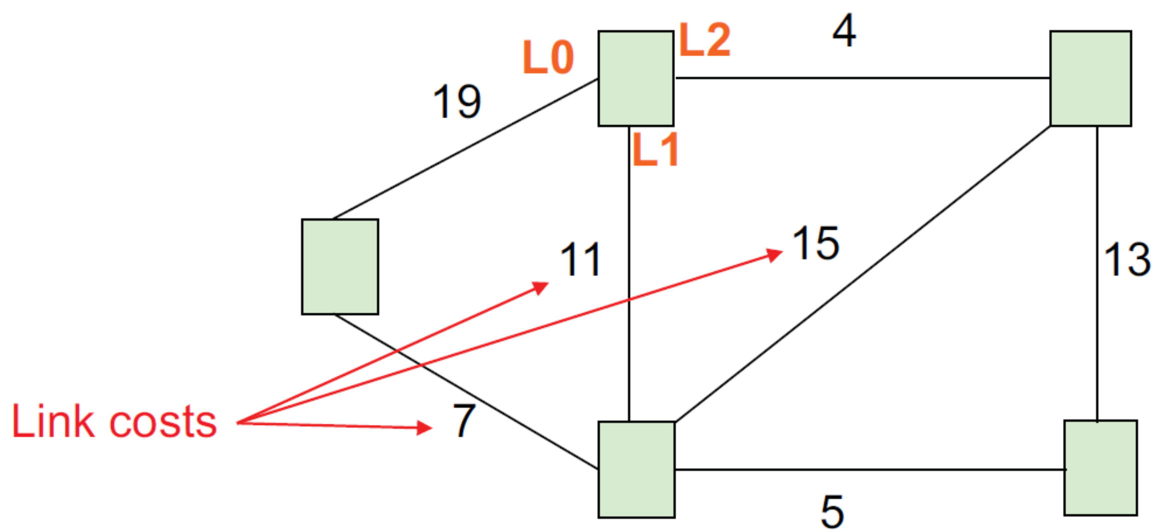


Collisions! Packet is 5 slots long in this picture

- A *collision* occurs when multiple transmissions overlap in time (even partially)
- Throughput = *Uncollided* packets per second
- Utilization = Throughput / Channel Rate

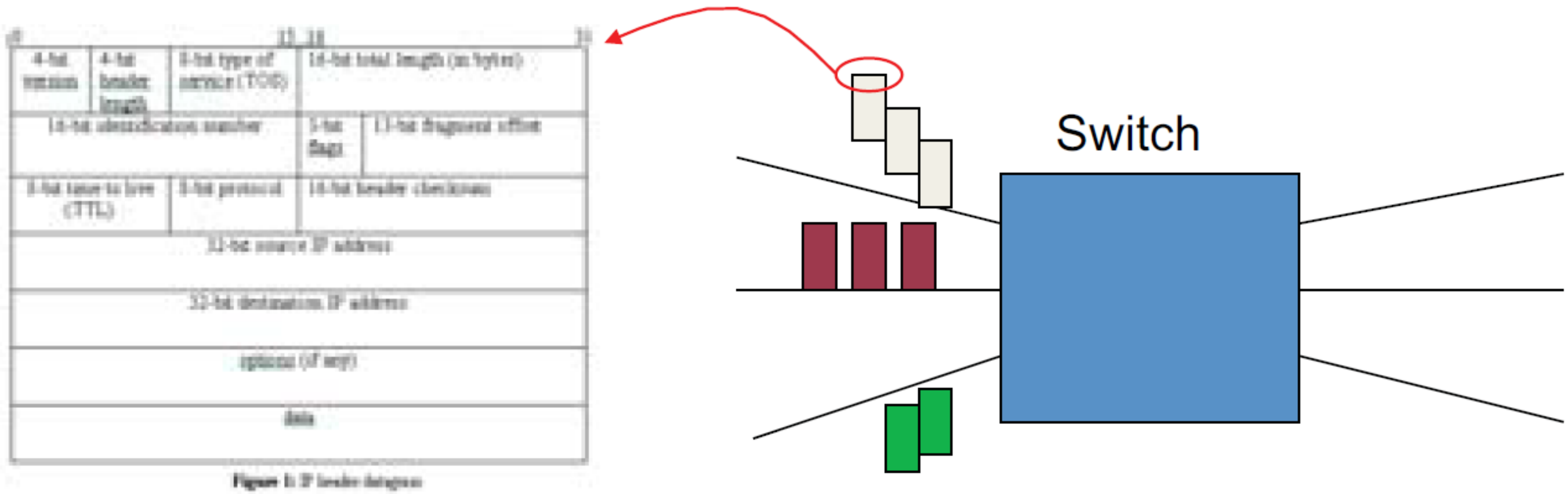
Lecture 3.3: Forwarding, Routing

The problem: Distributed Methods for Finding Paths In Networks



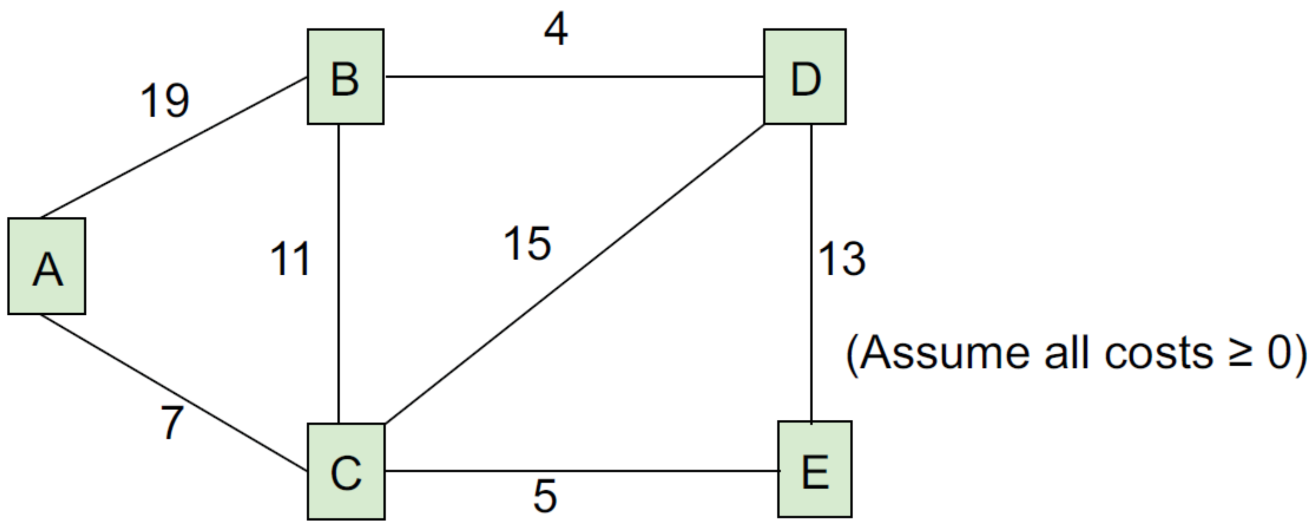
- Addressing (how to name nodes?)
  - Unique identifier for global addressing
  - Link name for neighbors
- Forwarding (how does a switch process a packet?)
- Routing (building and updating data structures to ensure that forwarding works)
- Functions of the **network layer**

Forwarding



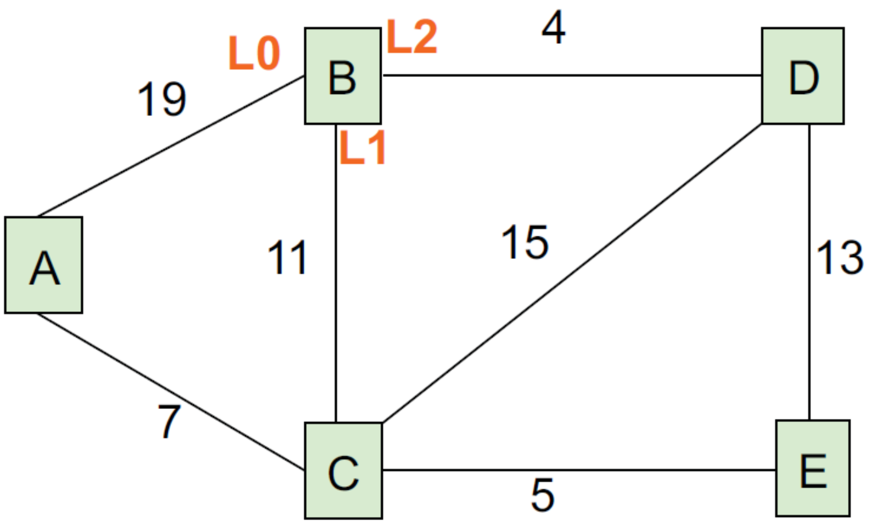
- Core function is conceptually simple
  - `lookup(dst_addr)` in routing table returns *route* (i.e., *outgoing link*) for packet
  - `enqueue(packet, link_queue)`
  - `send(packet)` along outgoing link
- And do some bookkeeping before enqueue
  - Decrement hop limit (TTL); if 0, discard packet
  - Recalculate checksum (in IP, header checksum)

Shortest Path Routing



- Each node wants to find the path with *minimum total cost* to other nodes
  - We use the term "shortest path" even though we're interested in min cost (and not min #hops)
- Several possible distributed approaches
  - Vector protocols, esp. *distance vector* (DV)
  - *Link-state* protocols (LS)

Routing Table Structure



Routing table @ node B

Destination	*Link (next-hop)*	*Cost*
A ROUTE	L1	18
B	'Self'	0
C	L1	11
D	L2	4
E ROUTE	L1	16

Distributed Routing: A Common Plan

- Determining live neighbors
  - Common to both DV and LS protocols
  - HELLO protocol (periodic)
    - Send HELLO packet to each neighbor to let them know who's at the end of their outgoing links
    - Use received HELLO packets to build a list of neighbors containing an information tuple for each link: (timestamp, neighbor addr, link)
    - Repeat periodically. Don't hear anything for a while → link is down, so remove from neighbor list.
- Advertisement step (periodic)
  - Send some information to all neighbors
  - Used to determine connectivity & costs to reachable nodes
- Integration step
  - Compute routing table using info from advertisements
  - Dealing with stale data

Distance-Vector Routing

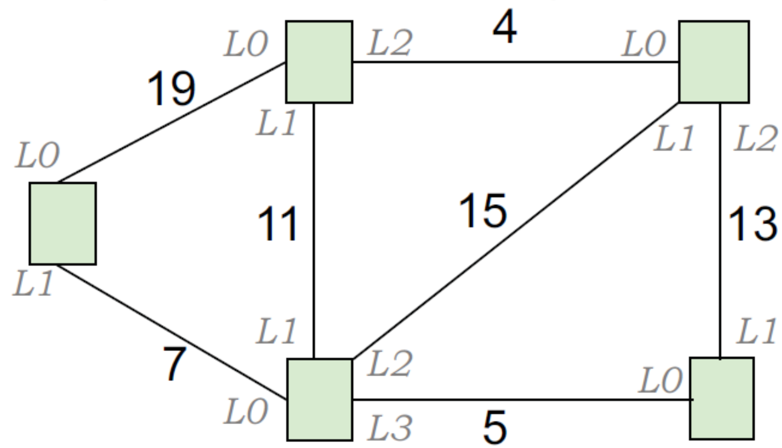
- DV advertisement
  - Send info from routing table entries: (dest, cost)
  - Initially just (self,0)
- DV integration step [Bellman-Ford]
  - For each (dest,cost) entry in neighbor's advertisement
    - Account for cost to reach neighbor: (dest,my_cost)
    - my_cost = cost_in_advertisement + link_cost
  - Are we currently sending packets for dest to this neighbor?
    - See if link matches what we have in routing table
    - If so, update cost in routing table to be my_cost
  - Otherwise, is my_cost smaller than existing route?
    - If so, neighbor is offering a better deal! Use it…
    - update routing table so that packets for dest are sent to this neighbor

DV Example: Round 2

```
{'A': (L0,19),
 'B': (None,0),
 'C': (L1,11),
 'D': (L2,4)
}
```

```
{'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}
```

```
{'A': (None,0),
 'B': (L0,19),
 'C': (L1,7)
}
```




```
{'A': (L0,7),
 'B': (L1,11),
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}
```

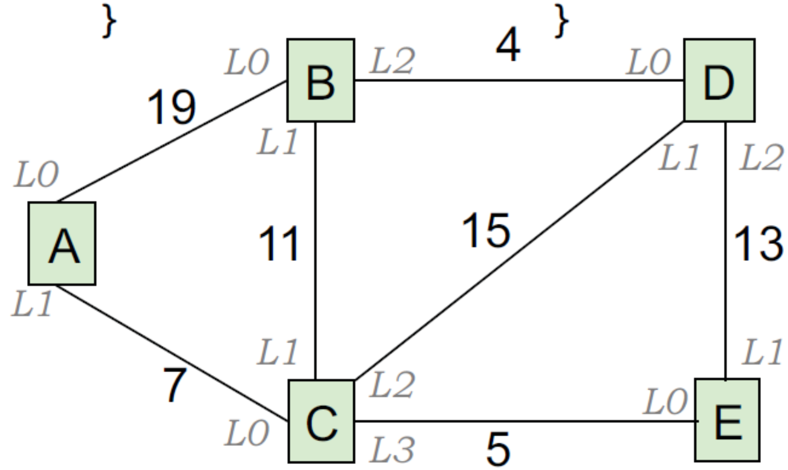
```
{'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}
```

Node A: update routes to B_C, D_C, E_C
Node B: update routes to A_C, E_C
Node C: no updates
Node D: update routes to A_C
Node E: update routes to A_C, B_C

DV Example: Round 3

```
{'A': (L1,18),
 'B': (None,0),
 'C': (L1,11),
 'D': (L2,4),
 'E': (L1,16)
}
```

```
{'A': (L1,22),
 'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}
```




```
{'A': (None,0),
 'B': (L1,18),
 'C': (L1,7),
 'D': (L1,22),
 'E': (L1,12)
}
```

```
{'A': (L0,7),
 'B': (L1,11),
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}
```

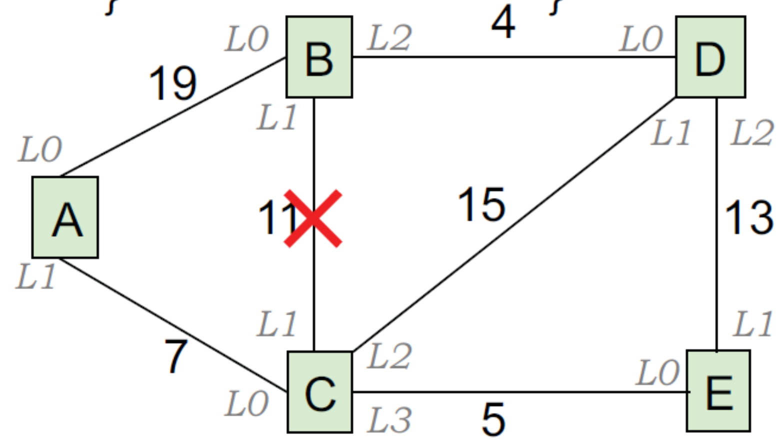
```
{'A': (L0,12),
 'B': (L0,16),
 'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}
```

Node A: no updates
Node B: no updates
Node C: no updates
Node D: no updates
Node E: no updates

DV Example: Breaking a Link

~~{'A': (L1,18),~~
 'B': (None,0),
~~'C': (L1,11),~~
 'D': (L2,4),
~~'E': (L1,16)~~
}

{'A': (L1,22),
 'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}



{'A': (None,0),
 'B': (L1,18),
 'C': (L1,7),
 'D': (L1,22),
 'E': (L1,12)
}

{'A': (L0,7),
~~'B': (L1,11),~~
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}

{'A': (L0,12),
 'B': (L0,16),
 'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}

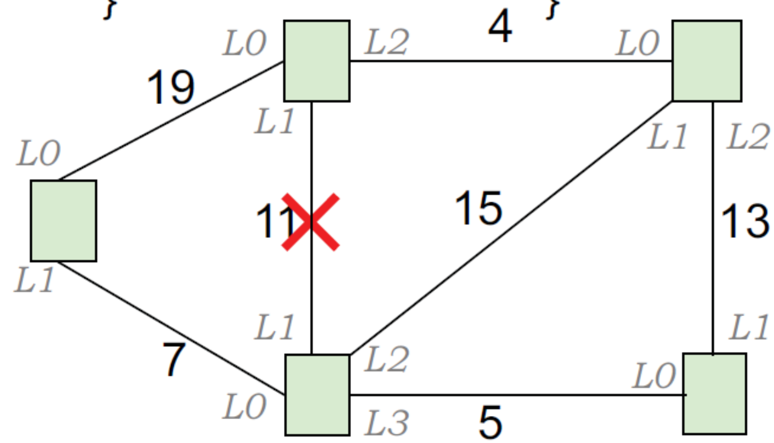
When link breaks: eliminate routes that use that link.

DV Example: Round 4

```
{'A': (None,∞),
 'B': (None,0),
 'C': (None,∞),
 'D': (L2,4),
 'E': (None,∞)
}
```

```
{'A': (L1,22),
 'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}
```

```
{'A': (None,0),
 'B': (L1,18),
 'C': (L1,7),
 'D': (L1,22),
 'E': (L1,12)
}
```




```
{'A': (L0,7),
 'B': (None,∞),
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}
```

```
{'A': (L0,12),
 'B': (L0,16),
 'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}
```

Node A: update cost to B_C
Node B: update routes to A_A, C_D, E_D
Node C: update routes to B_D
Node D: no updates
Node E: update routes to B_D

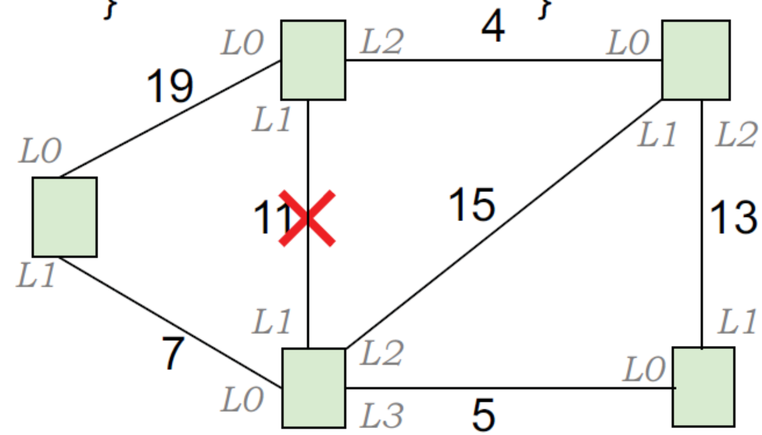
DV Example: Round 5

```
{'A': (L0,19),
 'B': (None,0),
 'C': (L2,19),
 'D': (L2,4),
 'E': (L2,17)
}
```

```
{'A': (L1,22),
 'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}
```

Update cost

```
{'A': (None,0),
 'B': (L1, ∞),
 'C': (L1,7),
 'D': (L1,22),
 'E': (L1,12)
}
```




```
{'A': (L0,7),
 'B': (L2,19),
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}
```

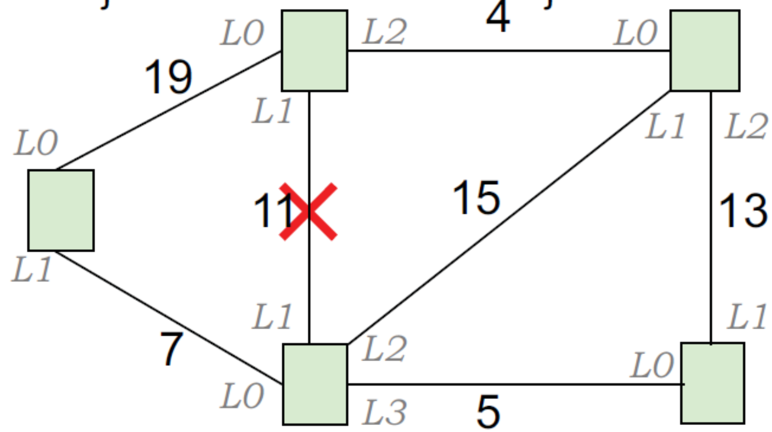
```
{'A': (L0,12),
 'B': (L1,17),
 'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}
```

Node A: update route to B$_B$
Node B: no updates
Node C: no updates
Node D: no updates
Node E: no updates

DV Example: Final State

```
{'A': (L0,19),
 'B': (None,0),
 'C': (L2,19),
 'D': (L2,4),
 'E': (L2,17)
}
```

```
{'A': (L1,22),
 'B': (L0,4),
 'C': (L1,15),
 'D': (None,0),
 'E': (L2,13)
}
```




```
{'A': (None,0),
 'B': (L0,19),
 'C': (L1,7),
 'D': (L1,22),
 'E': (L1,12)
}
```

```
{'A': (L0,7),
 'B': (L2,19),
 'C': (None,0),
 'D': (L2,15),
 'E': (L3,5)
}
```

```
{'A': (L0,12),
 'B': (L1,17),
 'C': (L0,5),
 'D': (L1,13),
 'E': (None,0)
}
```

Node A: no updates
Node B: no updates
Node C: no updates
Node D: no updates
Node E: no updates

Correctness and Performance

- Optimal substructure property fundamental to correctness of both Bellman-Ford and Dijkstra's shortest path algorithms
  - ***Suppose shortest path from X to Y goes through Z. Then, the sub-path from X to Z must be a shortest path.***
- Proof of Bellman-Ford via induction on number of walks on shortest (min-cost) paths
  - Easy when all costs > 0 and *synchronous model* (see notes)
  - Harder with distributed async model
- How long does it take for distance-vector routing protocol to *converge*?
  - Time proportional to largest number of hops considering all the min-cost paths

Link-State Routing

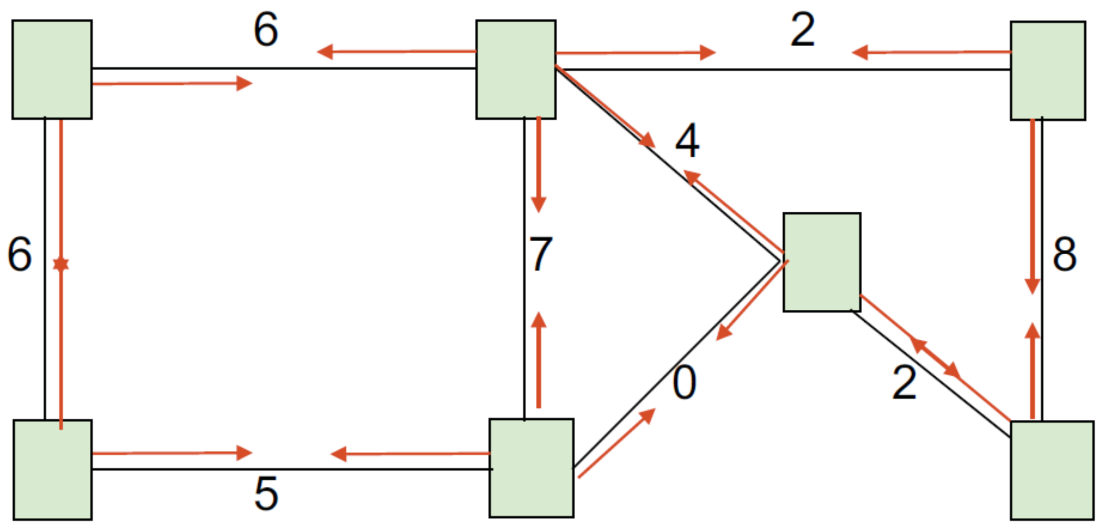
- Advertisement step
  - Send information about its links to its neighbors (aka **link state advertisement** or LSA):

    [seq#, [(nbhr1, linkcost1), (nbhr2, linkcost2), …]

  - Do it periodically (liveness, recover from lost LSAs)
- Integration
  - If seq# in incoming LSA > seq# in saved LSA for source node:
    update LSA for node with new seq#, neighbor list
    rebroadcast LSA to neighbors (→ **flooding**)
  - Remove saved LSAs if seq# is too far out-of-date
  - Result: Each node discovers current map of the network
- Build routing table
  - Periodically each node runs the same *shortest path algorithm* over its map (e.g., Dijkstra's alg)
  - If each node implements computation correctly and each node has the same map, then routing tables will be correct

LSA Flooding

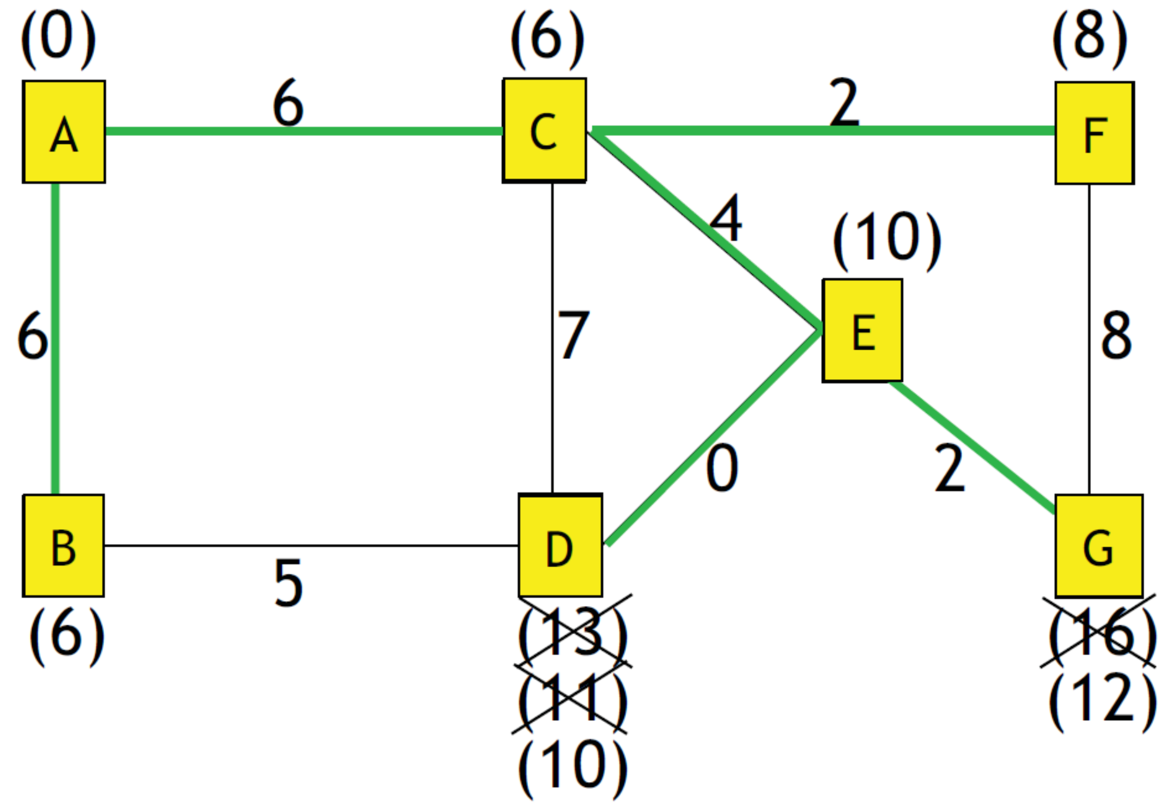
LSA: [F, seq, (G, 8), (C, 2)]



- Periodically originate LSA
- LSA travels each link in each direction
  - Don't bother with figuring out which link LSA came from
- Termination: each node rebroadcasts LSA exactly once
  - Use sequence number to determine if new, save latest seq
- Multiple opportunities for each node to hear any given LSA
  - Time required: number of links to cross network

Integration Step: Dijkstra's Algorithm

Suppose we want to find paths from A to other nodes



Dijkstra's Shortest Path Algorithm

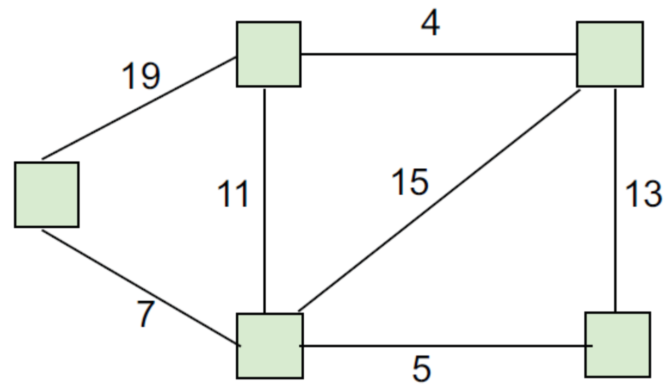
- Initially
  - nodeset = [all nodes] = set of nodes we haven't processed
  - spcost = {me:0, all other nodes: ∞} # shortest path cost
  - routes = {me:--, all other nodes: ?} # routing table
- while nodeset isn't empty:
  - find u, the node in nodeset with smallest spcost
  - remove u from nodeset
  - for v in [u's neighbors]:
    - d = spcost(u) + cost(u,v)   # distance to v via u
    - if d < spcost(v):   # we found a shorter path!
      - spcost[v] = d
      - routes[v] = routes[u] (or if u == me, enter link from me to v)
- Complexity: N = number of nodes, L = number of links
  - Finding u (N times): linear search=O(N), using heapq=O(log N)
  - Updating spcost: O(L) since each link appears twice in neighbors

Another Example

Finding shortest paths from A:

LSAs:

```
A: [(B,19), (C, 7)]
B: [(A,19), (C,11), (D, 4)]
C: [(A, 7), (B,11), (D,15), (E, 5)]
D: [(B, 4), (C,15), (E,13)]
E: [(C, 5), (D,13)]
```

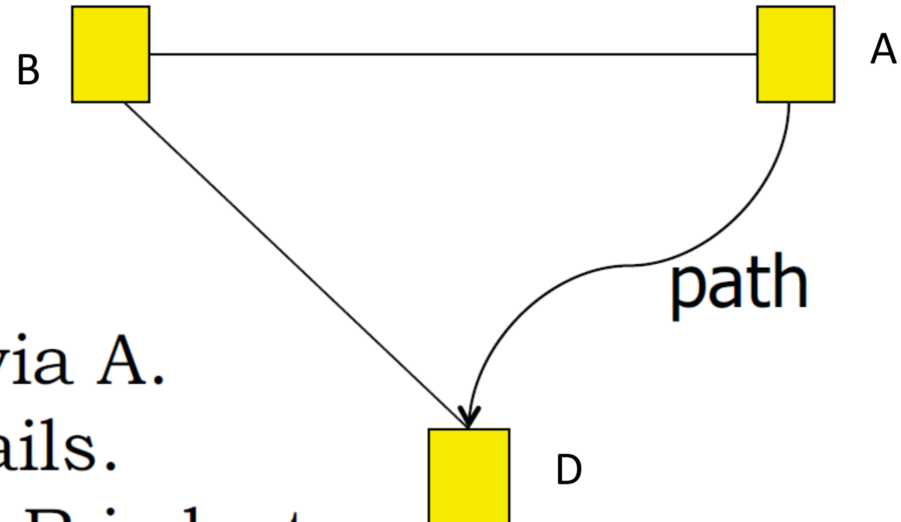


Step	*u*	*Nodeset*	*spcost*					*route*				
			A	*B*	*C*	*D*	*E*	*A*	*B*	*C*	*D*	*E*
0		[A,B,C,D,E]	0	∞	∞	∞	∞	--	?	?	?	?
1	A	[B,C,D,E]	0	19	7	∞	∞	--	L0	L1	?	?
2	C	[B,D,E]	0	18	7	22	12	--	L1	L1	L1	L1
3	E	[B,D]	0	18	7	22	12	--	L1	L1	L1	L1
4	B	[D]	0	18	7	22	12	--	L1	L1	L1	L1
5	D	[]	0	18	7	22	12	--	L1	L1	L1	L1

Failures

- Problems:Links and switches could fail
  - Advertisements could get lost
  - Routing loop
    - A sequence of nodes on forwarding path that has a cycle (so packets will never reach destination)
  - Dead-end: route does not actually reach destination
  - Loops and dead-ends lead to *routes not being valid*
- Solution
  - HELLO protocol to detect neighbor liveness
  - *Periodic advertisements* from nodes
  - *Periodic integration at nodes*
  - Leads to *eventual convergence to correct state*

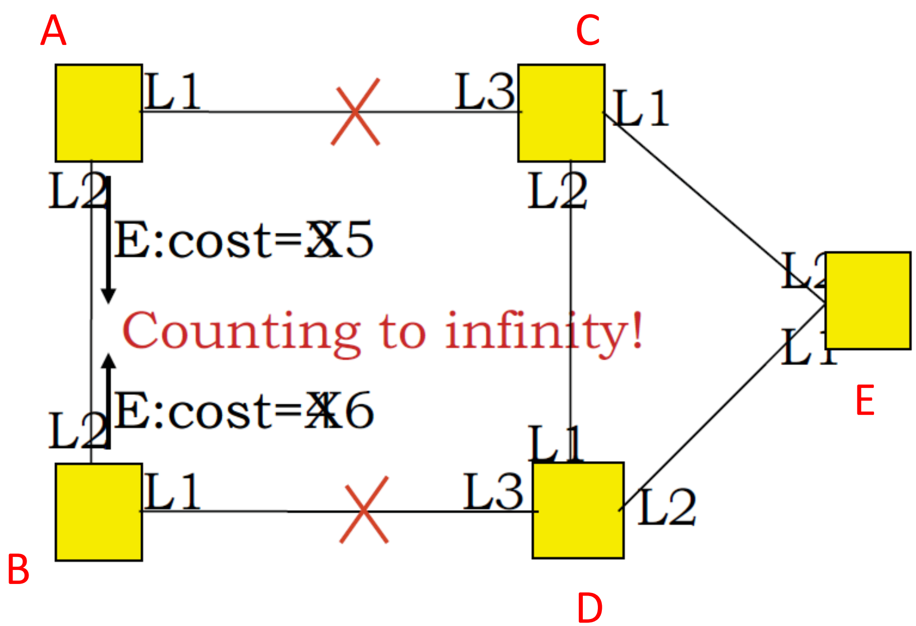
Routing Loop in Link-State Protocol



B to D is via A.
Link AD fails.
A's LSA to B is lost.
A now uses B to get to D.
But B continues to use A.
Routing loop!
Must wait for eventual arrival
of correct LSAs to fix loop.

Distance-Vector: Pros, Cons, and Loops

- \+ Simple protocol
- \+ Works well for small networks
- \- Works only on small networks



Suppose link AC fails.

When A discovers failure, it sends E: cost = INFINITY to B.

B advertises *E: cost=2* to A

A sets E: cost=3 in its table

Now suppose link BD fails.
B discovers it, then sets
E: cost = INFINITY.
Sends info to A, A sets
E: cost = INFINITY.

But what if A had advertised to B <u>before</u> B advertised to A?

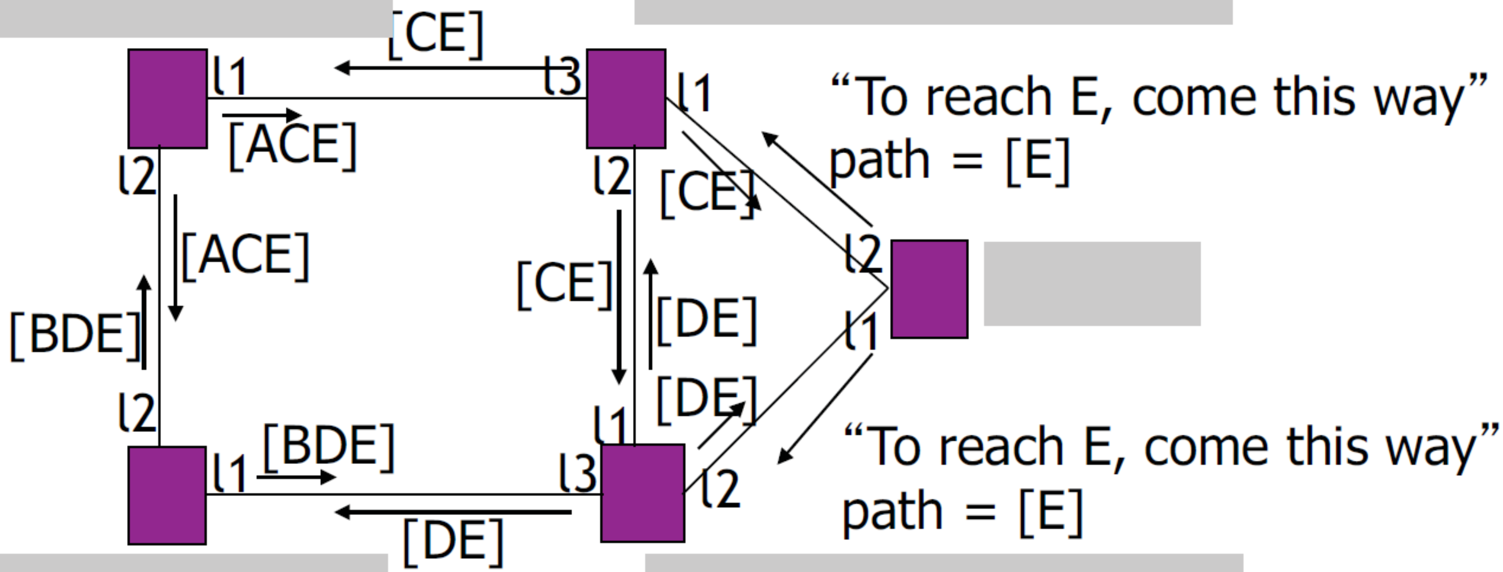
Fixing “Count to Infinity” with Path Vector Routing

- In addition to (or instead of) reporting costs, advertise the *path* discovered incrementally by the Bellman-Ford update rule

- Called “path-vector”

- Modify Bellman-Ford update with new rule: a node should ignore any advertised route that contains itself in the advertisement

Path Vector Routing



- For each advertisement, run "integration step"
  - E.g., pick shortest, cheapest, quickest, etc.
- **Ignore advertisements with own address in path vector**
  - Avoids routing loops that "count to infinity"

Summary

- The network layer implements the "glue" that achieves connectivity
  - Does addressing, forwarding, and routing
- Forwarding entails a routing table lookup; the table is built using *routing protocol*
- DV protocol: distributes route computation; each node advertises its best routes to neighbors
  - Path-vector: include path, not just cost, in advertisement to avoid "count-to-infinity"
- LS protocol: distributes (floods) neighbor information; centralizes route computation using shortest-path algorithm