CS 138: Communication I

In this lecture we cover a number of networking issues pertinent to the support of distributed computing. Much of the material is covered in more detail in CS 168, and many slides are taken from there. This is the '168 in a lecture' lecture.

Topics

- Network Metrics
- Layering
- Reliability
- Congestion Control
- Routing

The Fallacies of Distributed Computing

1. *The network is reliable*
2. *Latency is zero*
3. *Bandwidth is infinite*
4. The network is secure
5. *Topology doesn't change*
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

The first 7 are from Peter Deutsch, from Sun Microsystem, the eighth one by James Goslin, one of Java's creators. The ones in italics are relevant to the lecture today.

The article 'Fallacies of Distributed Computing Explained' expands on these, you can find it here: http://www.rgoarchitects.com/Files/fallacies.pdf

Performance Metrics

- **Bandwidth – Number of bits/unit of time the medium can transmit**
- **Latency – How long for message to cross network**
  - **Process + Queue + Transmit + Propagation**
- **Throughput – Effective number of bits received/unit of time**
  - ***e.g.* 10Mbps**
- **Goodput - *Useful* bits received per unit of time**
  - **Discounts protocol overhead**
- **Jitter – Variation in latency**

By the definitions, Bandwidth >= Throughput >= Goodput. You can't send faster than the (digital) bandwidth, and the throughput calculation takes into account any time in which you are not sending bytes (such as when waiting for acknowledgments in a stop-and-go protocol), or when you have to retransmit due to lost or corrupted bytes. Goodput counts a subset of the bytes you do get through, just the bytes that matter for your application.

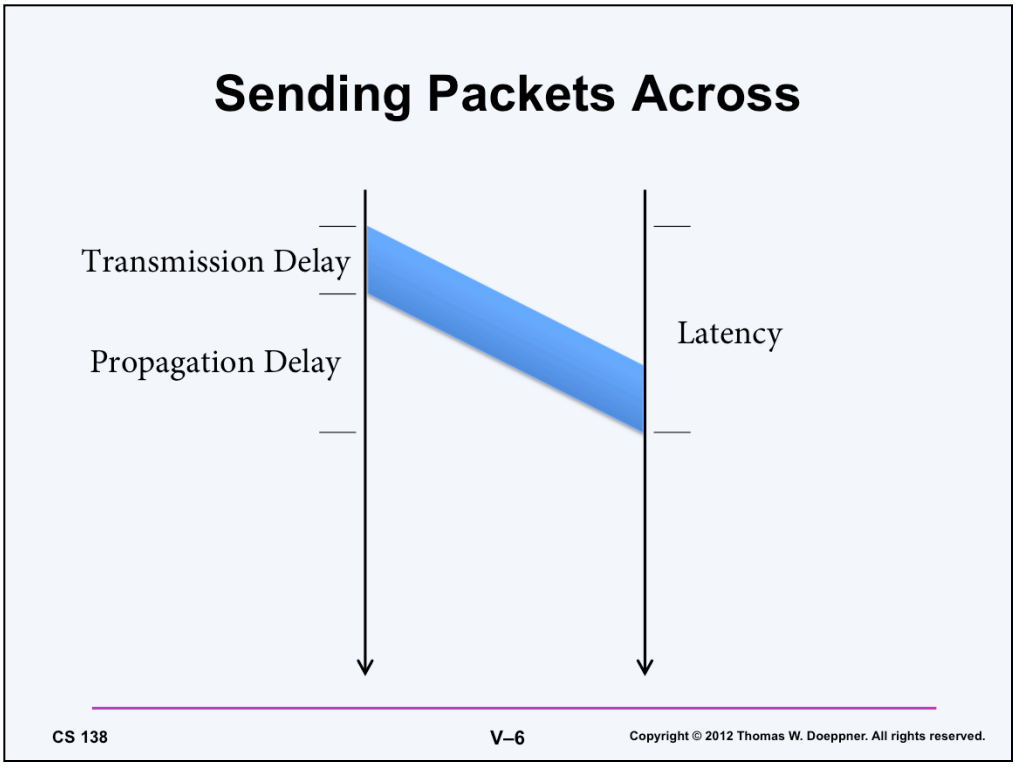
If your application needs a constant bitrate, then you have to mask jitter by adding buffering at the receiving end. Essentially, if the worst delay a bit can have is T, and the best t, then you have to have a buffer that accommodates BWx(T-t) bytes.

Latency

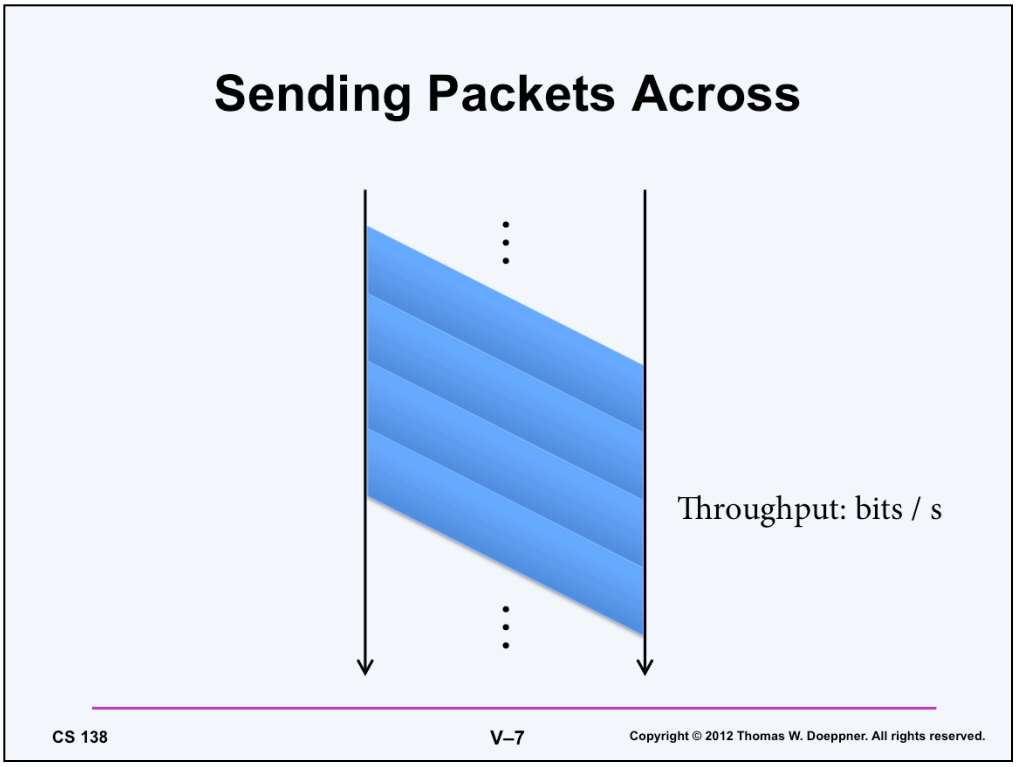
- **Processing Delay**
  - **Per message, small, limits throughput**
  - ***e.g. to achieve full rate at a 100Mbps link, you have a budget of* 120µs/pkt:**

$$\frac{100\,Mb}{s} \times \frac{pkt}{1500B} \times \frac{B}{8b} \approx 8{,}333\,pkt/s$$

- **Queueing Delay**
  - **Highly variable, offered load vs outgoing b/w**
- **Transmission Delay – depends on medium**
  - **Size/Bandwidth**
- **Propagation Delay – depends on medium**
  - **Distance/Speed of Light**



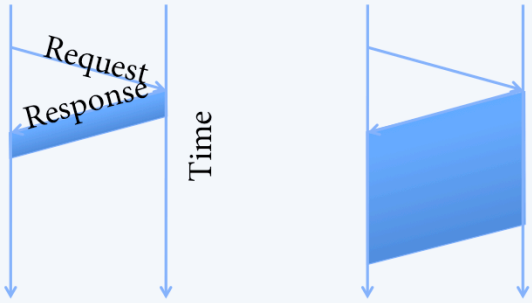
How long does it take to send a frame across?



How fast can we receive bytes?

Which matters most, bandwidth or delay?

- How much data can we send during one RTT?
- *E.g.*, send request, receive file

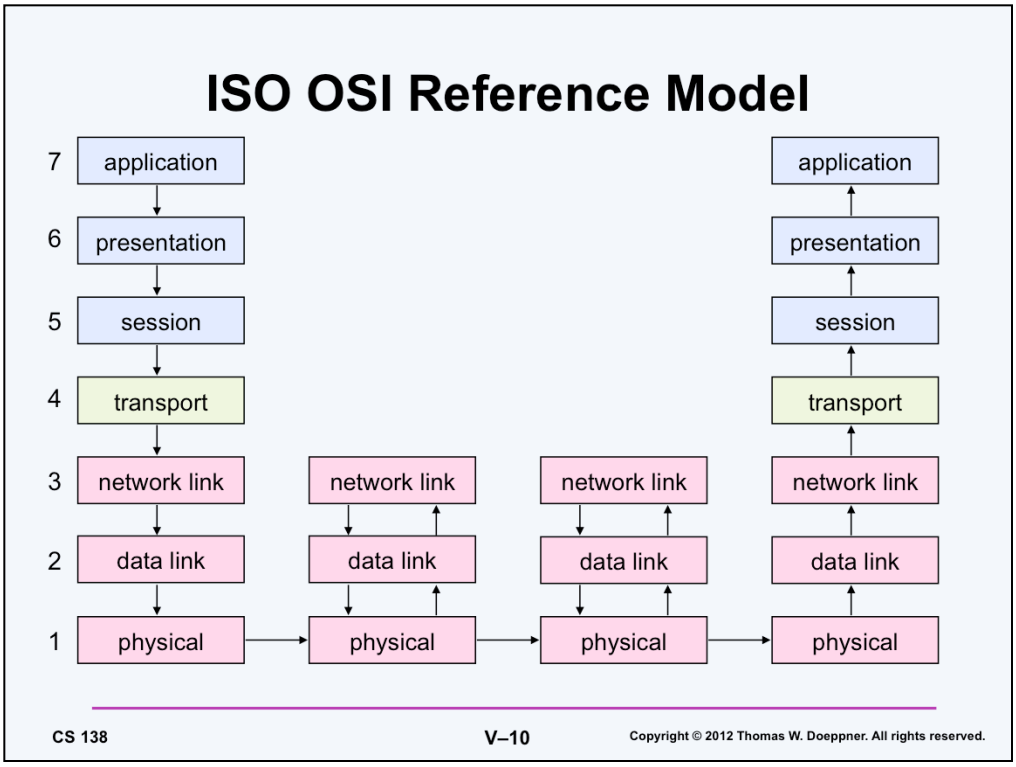


- For small transfers, latency more important, for bulk, throughput more important

Many Requirements

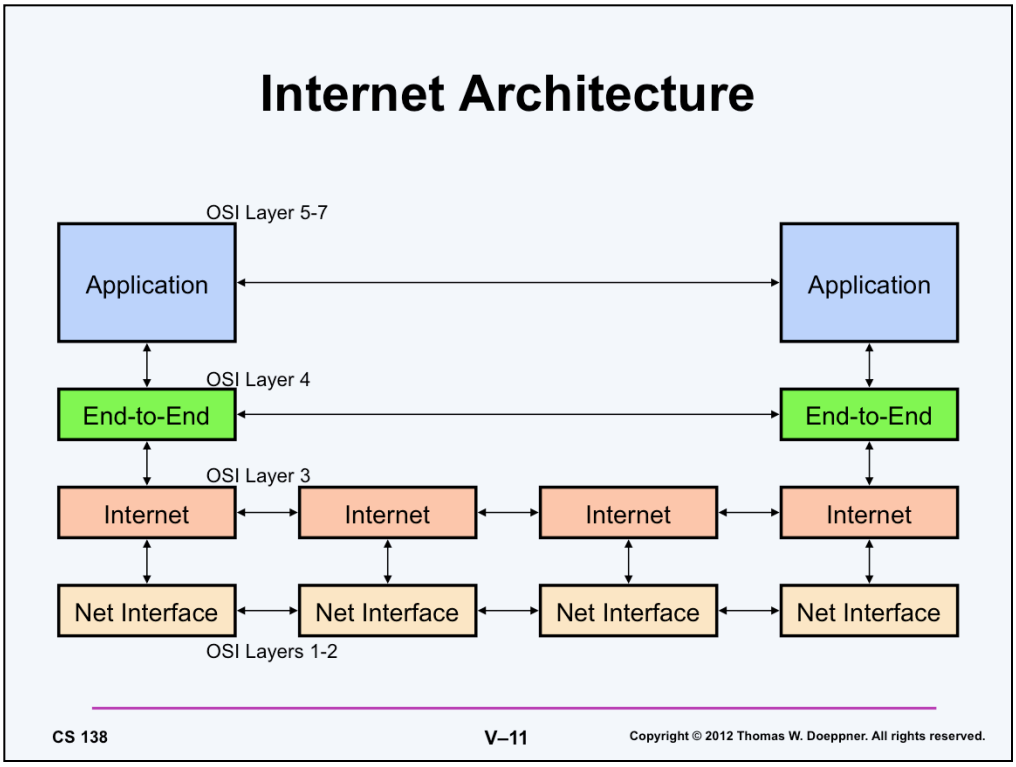
- Modulation, encoding, framing
- Routing
- Reliability
- Flow control
- Congestion control
- Security
- …

- How to organize all of these?



The *International Organization for Standardization* has devised a model for the design of communication protocols (known as the *open systems interconnection* (OSI) model). In this model each communicating entity has seven layers of protocol. The bottom layer (1) is known as the *physical layer*, and it essentially corresponds to the wire. The next layer (2) is known as the *data link layer*. It provides the means for putting data on the wire (and for taking it off). An example is *Ethernet*. The next layer (3) is known as the *network layer*. Its primary responsibility is to see that the data travels to the intended destination (perhaps via a number of intermediate points). The next layer is known as the *transport layer*. Its job is to see to it that the data, which is transferred between machines by the network layer, reaches the desired party at the destination machine. The notion of a "connection" is maintained by this layer. The next layer is known as the *session layer*. This layer doesn't really do very much (if anything); it is responsible for maintaining the notion of a "session." Sessions might be in one-to-one correspondence with transport connections; there might be two successive sessions on the same connection; or one session might span multiple connections (e.g., the first connection was terminated due to a communication failure, the session continues as soon as communication is reestablished). The next layer is known as the *presentation layer*. Its job is to deal with the fact that different machines have different representations for data (i.e. it must somehow translate between data representations) and to deal with such concerns as compression or encryption of data. Finally, the *application layer* is where all other software resides. However, it has been discovered that there is "system software" that logically fits above the presentation layer. The only place to put it is in the application layer, so "real" application software sits on top of the application layer.

The bottom three layers are sometimes known as the communications subnet. If our data must pass through a number of machines on their way to the destination, each intermediate machine has an implementation of these lower layers to forward the data on.



The OSI model is a classic example of something designed by committee. The model it describes does not fit exactly with the model of TCP/IP, but its terminology has become so commonplace that its exact requirements are conveniently stretched so that TCP/IP fits in. Originally it was thought (by the designers of the model) that protocols designed to fit in the OSI model would be not only competitors of TCP/IP, but would be replacing TCP/IP. The rumors of TCP/IP's death were greatly exaggerated; one rarely hears about its competitors these days.

Today one hears very little of the OSI protocols (though the OSI seven-layer terminology is much used); whatever competition there was between the OSI protocols and the internet protocols was definitely won by the latter.

The protocols used on the Internet are, strangely enough, known as the *Internet protocols* (also known as TCP/IP). They don't fit precisely into the OSI model (there is no analog of the session and presentation layers), but the rough correspondence is shown in the picture.

Layers

- Application – what the users sees, *e.g.*, HTTP
- Presentation – crypto, conversion between representations
- Session – can tie together multiple streams (*e.g.*, audio & video)
- Transport – demultiplexes, provides reliability, flow and congestion control
- Network – sends *packets*, using *routing*
- Data Link – sends *frames*, handles media access
- Physical – sends individual bits

How to Place Functionality

- Don't provide functionality on a layer that some of the users won't need
  - E.g. security, in-order-delivery conflicts with timeliness
- Don't provide functionality on a layer when it is insufficient
  - E.g. error correction, reliability
  - This is the "End-to-end" Principle
  - Can violate when there is a performance gain
- Do provide a functionality that can be reused
  - E.g., IP routing and forwarding is useful to many layers above

These are at best "guidelines".

It is interesting to note that the first and the second want to push functionality to higher layers, while the third wants to push functionality to lower layers. The Internet architecture, in contrast to the telephony network, and even to some other architectures that tried to provide too much in the lower layers, went radically to pushing functionality to higher layers. This is one of the great reasons of its success, as it lowered the barrier to entry and fostered innovation, as new functionality could be added at the higher layers without impacting other components at the same level.

The paper 'End-to-end Argument in System Design' by Saltzer, Reed, and Clark is an EXCELLENT read on this topic.

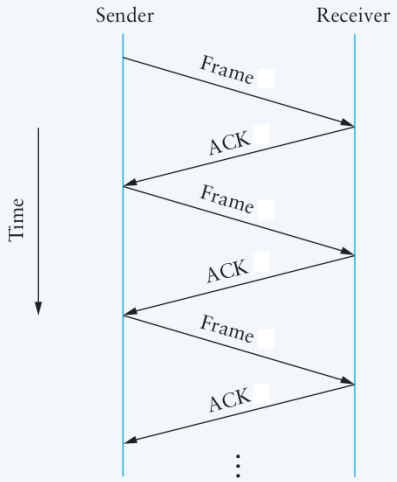
Reliable Delivery

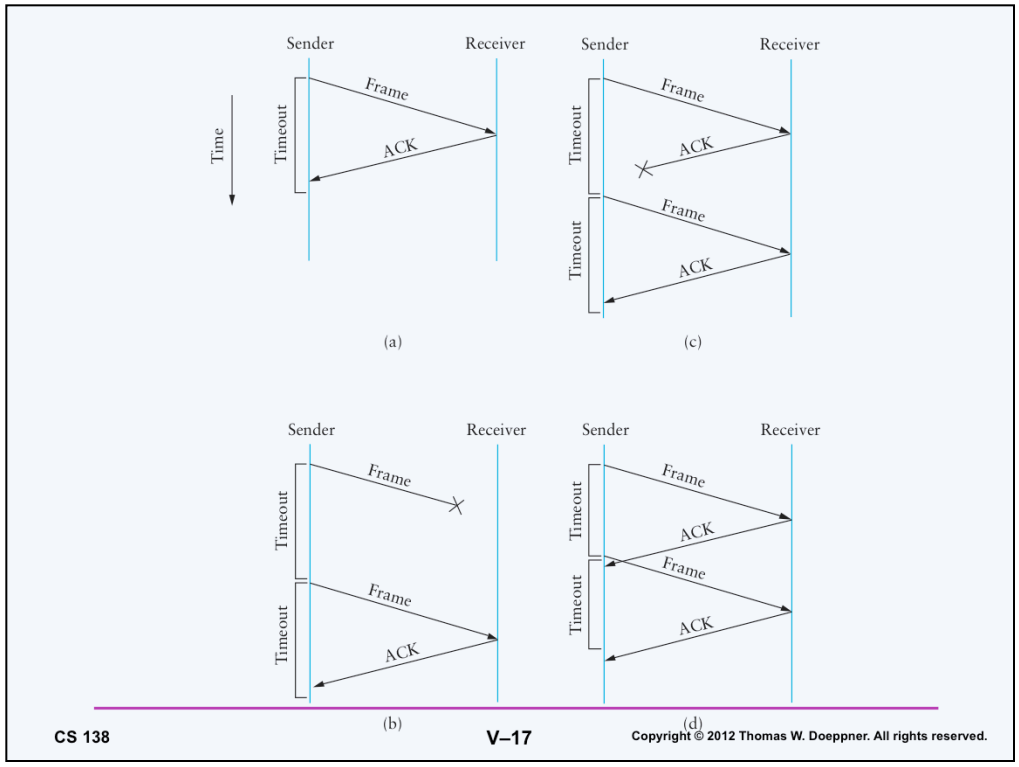
- Several sources of errors in transmission
- Error detection can discard bad frames
- Problem: if bad packets are lost, how can we ensure reliable delivery?
  - Exactly-once semantics = at least once + at most once

We will come back to this topic in the context of RPCs as well.

At Least Once Semantics

- How can the sender know packet arrived *at least once*?
  - Acknowledgments + Timeout
- Stop and Wait Protocol
  - S: Send packet, wait
  - R: Receive packet, send ACK
  - S: Receive ACK, send next packet
  - S: No ACK, timeout and retransmit





In (a) you have the expected case in which the transmission and the acknowledgment succeed, all before the timeout.

In (b) you have the case in which you NEED a retransmission, because the transmission failed.

In (c) and (d) you wouldn't need the retransmission, because the receiver got the packet, BUT THE SENDER CAN'T distinguish (c) from (b).

In (d) the timeout value is exceeded, and a premature retransmission goes out.

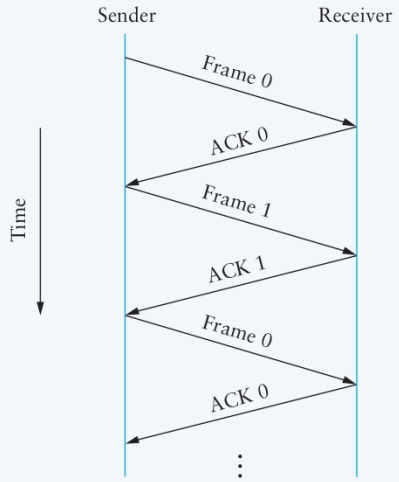
In these two latter cases, the receiver gets two copies of the transmission.

In these cases, DOES THE RECEIVER KNOW WHETHER THE FRAME IS A NEW FRAME OR A RETRANSMISSION?

Stop and Wait Problems

- Duplicate data
- Duplicate acks
- Slow (channel idle most of the time!)
- May be difficult to set the timeout value

Duplicate data: adding sequence numbers



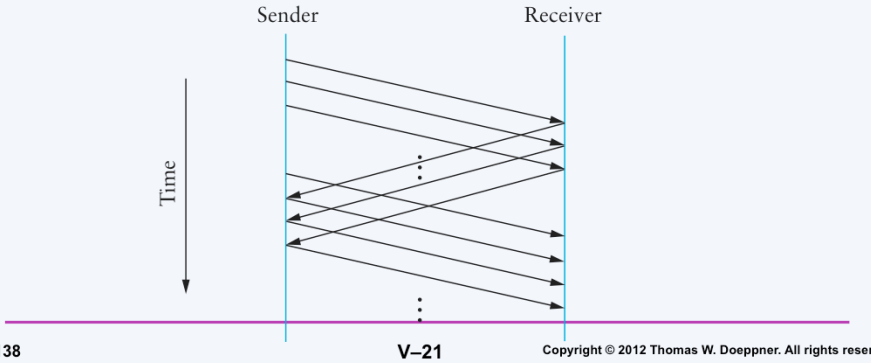
If you add a bit to the packets, in this protocol you can distinguish between a retransmission and a new transmission.

At Most Once Semantics

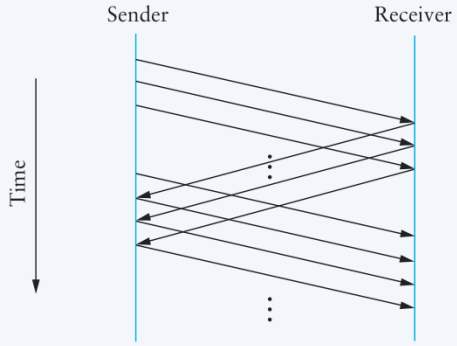
- How to avoid duplicates?
  - Uniquely identify each packet
  - Have receiver and sender remember
- Stop and Wait: add 1 bit to the header
  - Why is it enough?

Going faster: sliding window protocol

- Still have the problem of keeping pipe full
  - Generalize approach with > 1-bit counter
  - Allow multiple outstanding (unACKed) frames
  - Upper bound on unACKed frames, called *window*

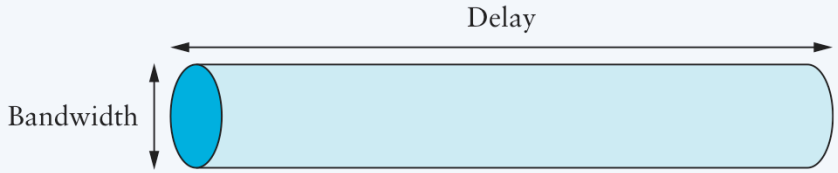


How big should the window be?



- How many bytes can we transmit in one RTT?
  - BW B/s x RTT s => “Bandwidth-Delay Product”

Maximizing Throughput



- Can view network as a pipe
  - For full utilization want bytes in flight ≥ bandwidth × delay
  - But don't want to overload the network
- What if protocol doesn't involve bulk transfer?
  - Get throughput through concurrency – service multiple clients simultaneously

Problem: on the Internet, the pipe varies

- Different paths
- Queues along the way
- Other flows

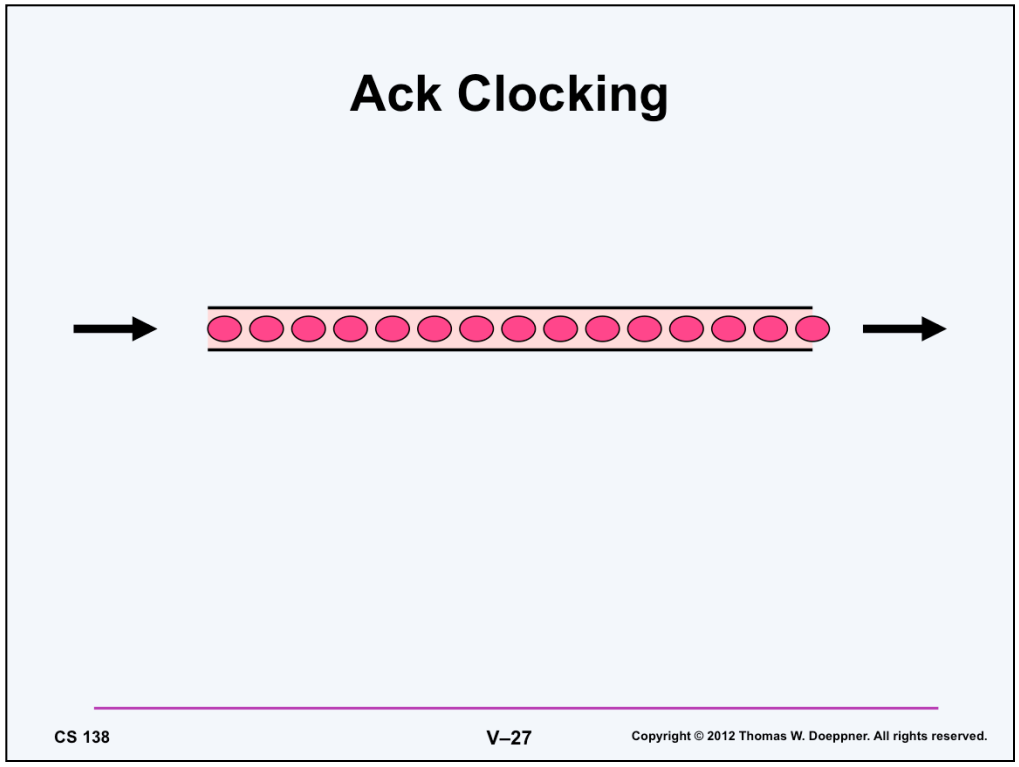
- Q: How to set the window size then?

Efficiency

- 3 goals:
  - Utilize the network
  - Don't overwhelm the receiver: flow control
  - Don't overwhelm the network: congestion control

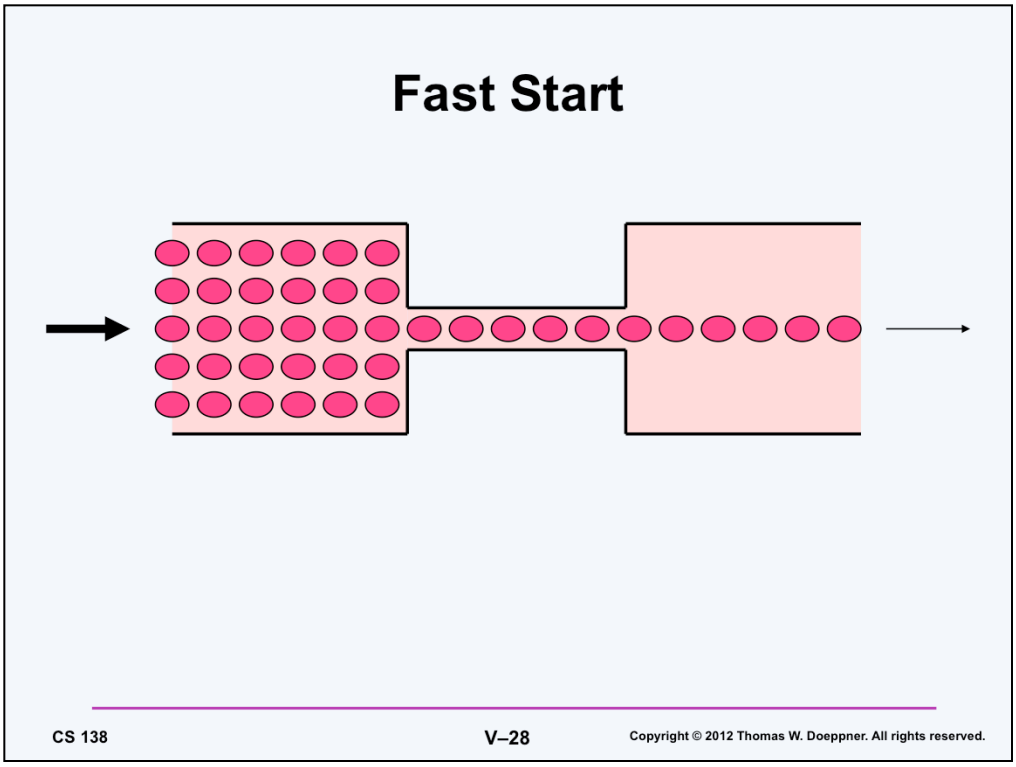
Congestion Control

- 3 Key Challenges
  - Determining the available capacity in the first place
  - Adjusting to changes in the available capacity
  - Sharing capacity between flows
- Idea
  - Each source determines network capacity for itself
  - Rate is determined by window size
  - Uses implicit feedback (drops, delay)
  - ACKs pace transmission (self-clocking)

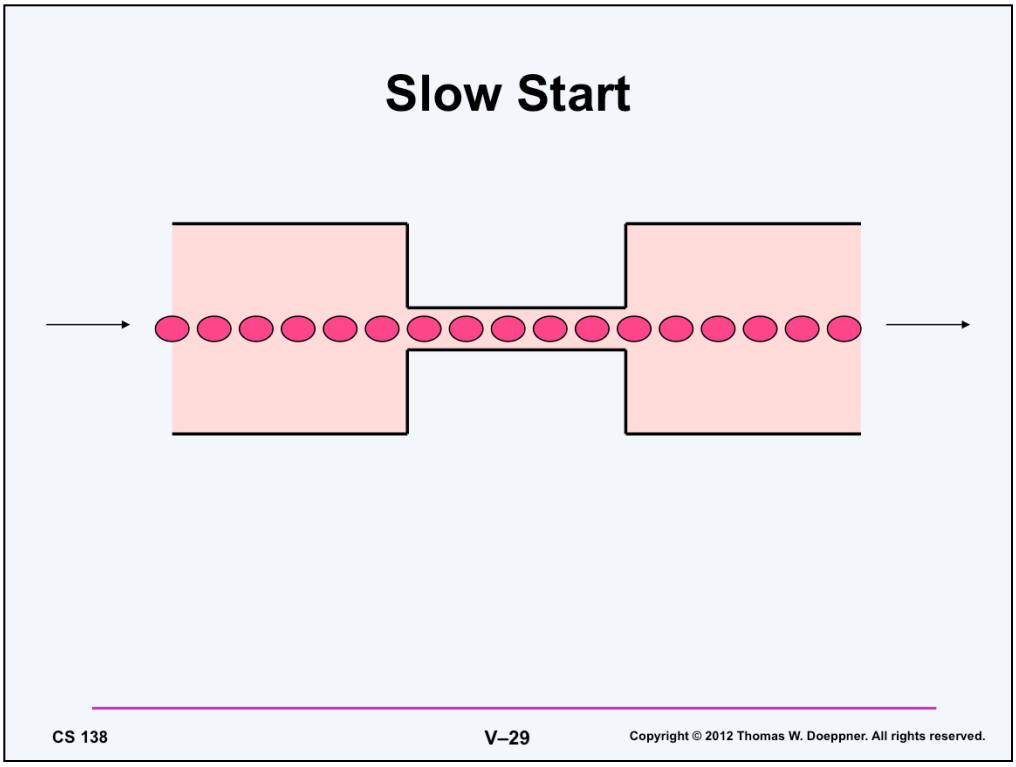


Conservation of packets: When a data pipeline is running at capacity, a new packet should be transmitted at one end for every packet received at the other.

One might think of network connections as being like pipelines — they have finite and varying capacities. Data flows through pipelines and obeys Kirchoff's current laws: the flows in must equal the flows out. Since acks are being generated for the packets being received, the sender simply transmits a new packet every time it receives an ack.



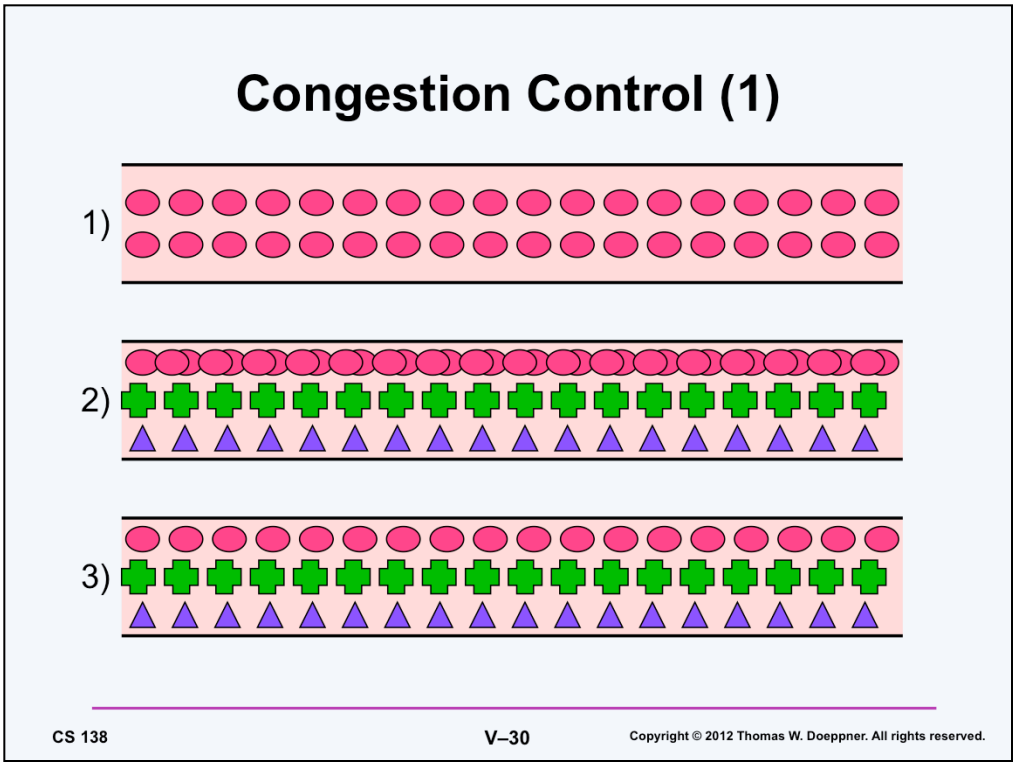
The problem with the "ack clocking" approach is getting the pipeline to capacity (and determining when it's there). Suppose a pipeline has a capacity of one gigabyte (capacity is bandwidth times the time it takes a packet to traverse the pipeline; this is known as the *bandwidth-delay product*). A naive (and fast) sender might start transmitting at its maximum speed, say 10 megabytes/second — since the pipeline is empty, it seems reasonable to fill it as quickly as possible. However, one or more downstream routers might be able to handle data at only 1 megabyte/second. Packets will start piling up at the first such router, which will soon start discarding them, forcing retransmissions by the sender.



A better approach is the one called *slow start*: the sender starts transmitting slowly but speeds up until it fills the pipeline and is transmitting no faster than the pipeline can handle (as governed by ack clocking).

More precisely, each sender maintains a *congestion window*. The amount of data the sender is allowed to transmit is the minimum of the sizes of the congestion and send windows. When transmitting data into an idle connection (i.e., an empty pipeline — one with no acks outstanding), the congestion window is set to one. Each time an ack is received, the congestion window size is increased by one. This results in exponential growth of the window size over time: after each roundtrip time, assuming no packets are lost, the window size doubles.

Of course, we need to know when to stop this exponential growth; we take this up in the next pair of slides.



Let's assume that, somehow, we have found the capacity of the pipeline. Using ack clocking, we are maintaining an optimal data rate (line 1 of the slide). Suddenly, some additional connections are competing with us for bandwidth — congestion occurs and many of our packets are dropped (line 2 of the slide).

In earlier versions of TCP, it was up to the router that dropped a packet to inform the sender that there is a problem. The sender was then to slow down. In practice, this approach failed miserably. A better approach, by Van Jacobson (see "Congestion Avoidance and Control", Proceedings of SIGCOMM '88) is for the sender to recognize that there is a problem **as soon as one of its packets is not acknowledged.** That a packet is lost might be due to its being corrupted, but the far most likely cause is that it was dropped by a router (due to congestion). The question is, how much to reduce the rate by?

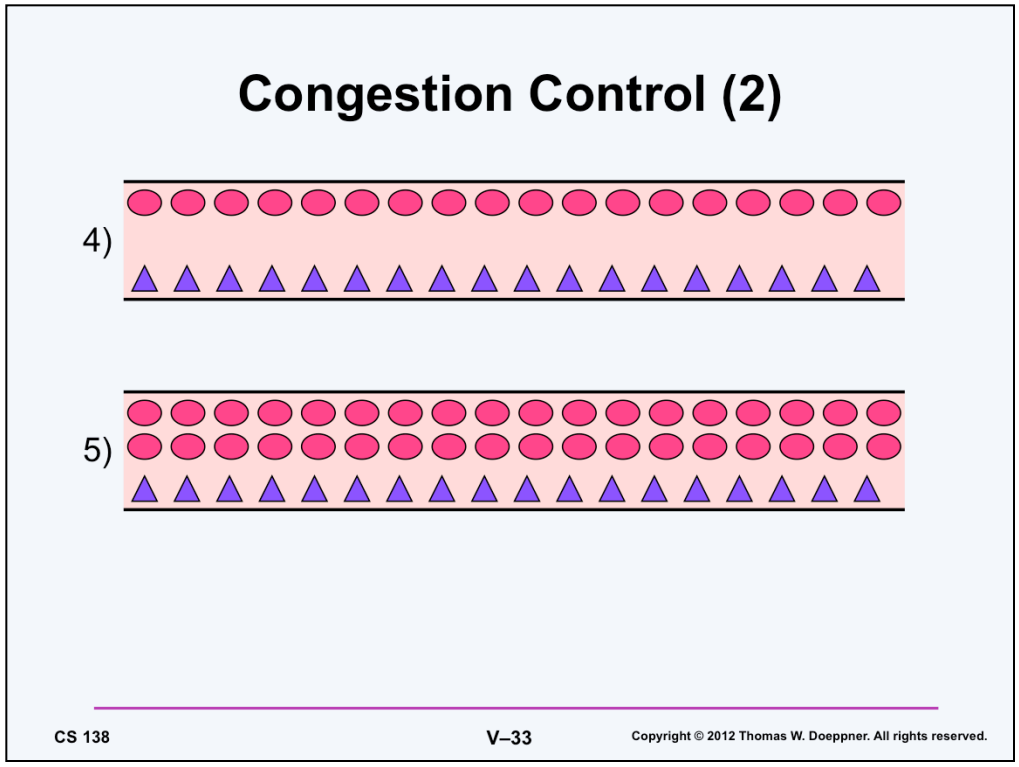
An unfortunate complication is that by the time the sender times out waiting for the dropped packet to be ack'd, the pipeline has emptied. Thus it must now use the slow-start approach to get the pipeline going again.

Dealing with Congestion

- Assume losses are due to congestion
- After a loss, reduce congestion window
  - How much to reduce?

How much to reduce window?

- Crude model of the network
  - Let L_i be the load (# pkts) in the network at time I
  - If network uncongested, roughly constant $L_i = N$
- What happens under congestion?
  - Some fraction γ of packets can't exit the network
  - Now $L_i = N + \gamma L_{i-1}$, or $L_i \approx g^i L_0$
  - Exponential increase in congestion
- Sources must decrease offered rate exponentially
  - i.e, multiplicative decrease in window size
  - TCP chooses to cut window in half



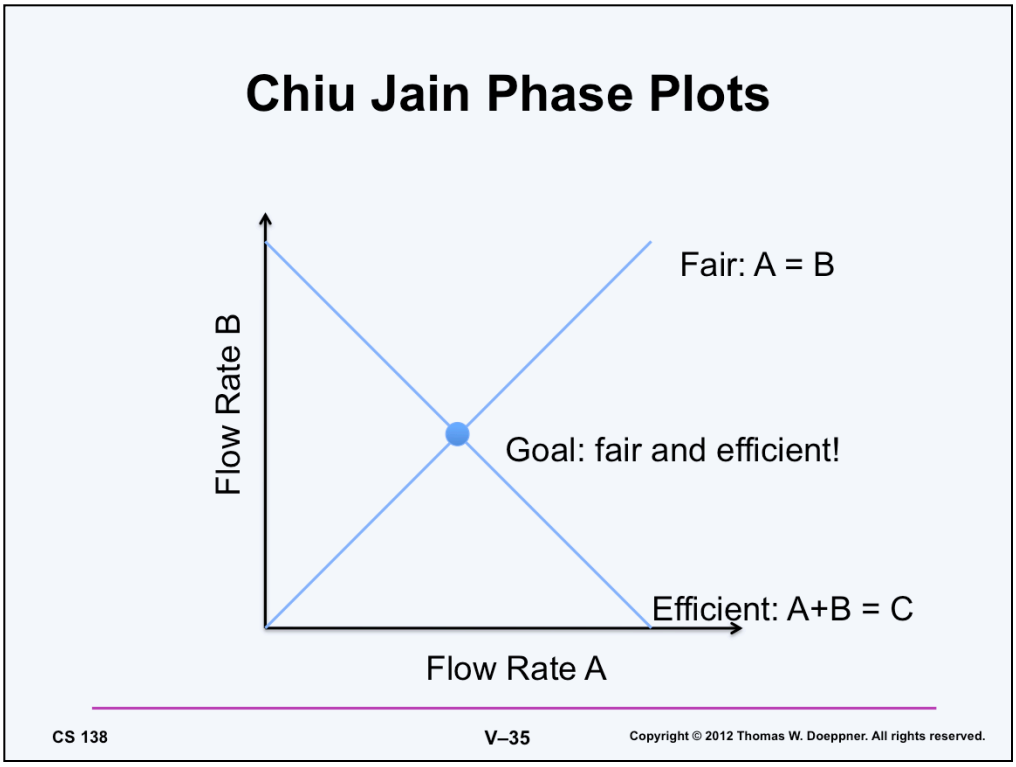
Let's assume that the sender has once again found the capacity of the pipeline and is transmitting according to ack clocking. Now (line 4), a connection that was using the pipeline drops out and there is surplus bandwidth. It's important that our sender discover this so it can transmit at a faster rate. Since the parties that ceased to communicate are not going to announce this fact to others, our sender must discover it on its own.

This discovery is implemented by having our sender gradually raise its transmission rate. It might do this exponentially, as in slow start, but this would quickly cause it to exceed the pipeline's capacity and it would have to lower the rate by a factor of two. This would result in fairly wild oscillations. A better approach, suggestion in the previously cited paper by Jacobson and implemented in most TCP implementations, is to raise the transmission rate linearly, as opposed to exponentially. Thus the rate rises slowly until it's too high and a packet is dropped, at which point it is again halved. The net effect is a slow oscillation around the optimal speed.

To deal with both slow start and congestion control, we must remember that whenever we timeout on an ack, the pipeline is emptied. When this happens, the current value of the congestion window size is halved and stored in another per-connection variable, *ssthresh* (slow-start threshold). The congestion window is then set to one and the slow-start, exponential growth approach (which isn't all that slow) is used until the window size reaches ssthresh. At this point the transmission rate through the pipeline has reached half of what it was before and we turn off slow start and switch from exponential growth of the congestion window size to linear growth, as discussed in the previous paragraph.

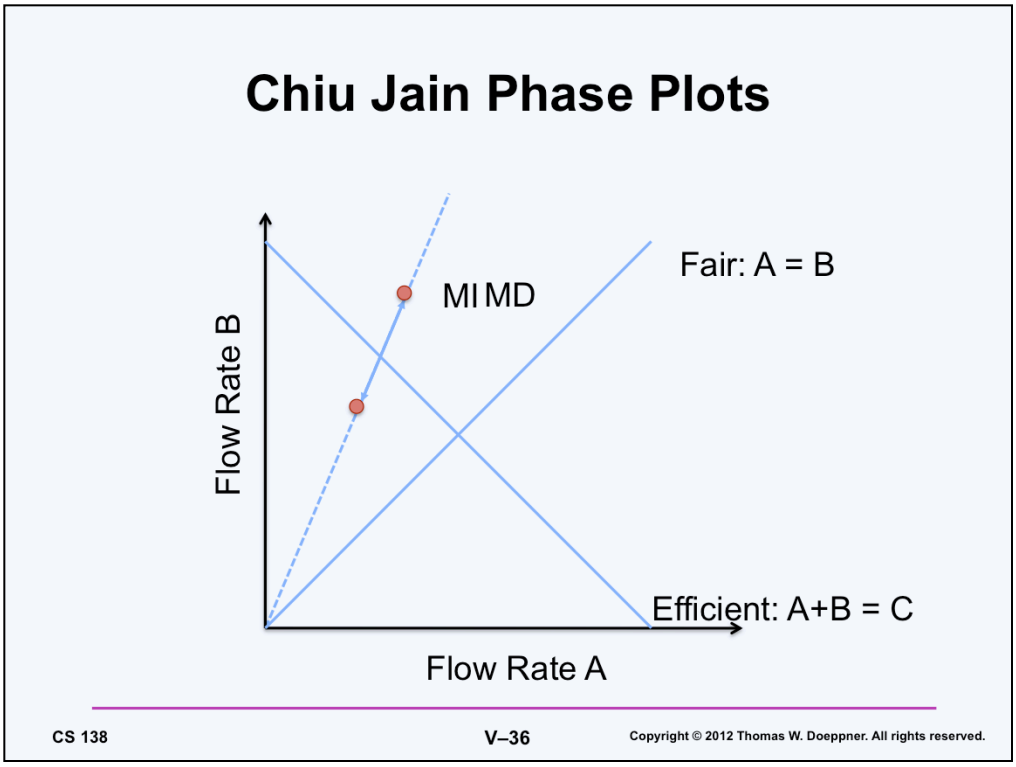
How to use extra capacity?

- Network signals congestion, but says nothing of underutilization
  - Senders constantly try to send faster, see if it works
  - So, increase window if no losses… By how much?
- Multiplicative increase?
  - Easier to saturate the network than to recover
  - Too fast, will lead to saturation, wild fluctuations
- Additive increase?
  - Won't saturate the network
  - Remember fairness?

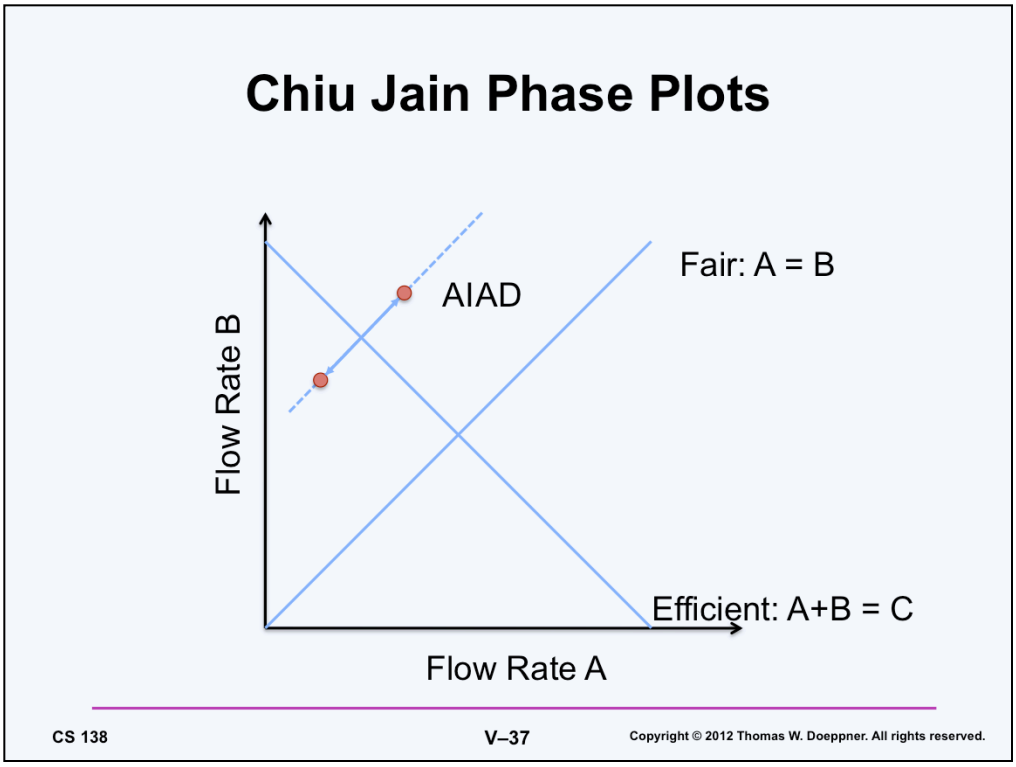


This is a simple model of two flows that adjust their rates synchronously and by the same policy. This also assumes that they detect congestion at the same time. The Efficient line is the line where the full rate of the bottleneck link is utilized, hence A+B = C. The region between this line and the origin is the feasible region, there will be congestion. The fair line is where the rates for the two flows are the same. We want the rates to converge to the intersection of these two lines, independent of the initial conditions.

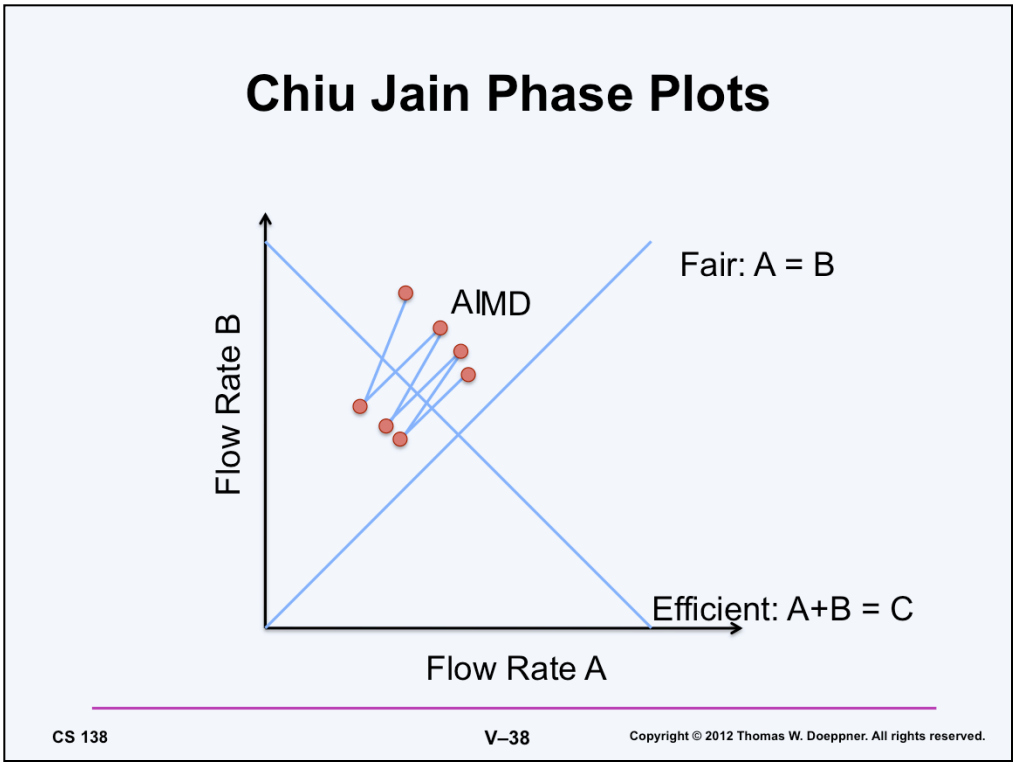
Why can't we just adjust the rates so that we jump directly to the goal? Because the nodes, individually, can't know what the goal is. This would imply that they knew exactly what the capacity was (they don't), and they knew all the other participants, and their current demands (which they also have no way of knowing). The only signal they have here is whether they've experienced congestion.



If the flows start at the top point, and multiplicatively decrease their rates, then they will scale the rate vector by a factor, going towards the origin on the dotted line. If they then increase their rates multiplicatively, they will go along the same line to the previous situation. This will not improve the fairness over time. Multiplicatively decreasing improves fairness (you get closer to the fair line) because the participant with the higher rate reduces more. The same doesn't hold for the additive case.



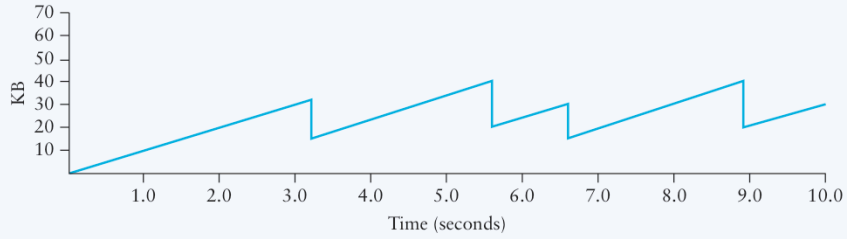
If instead they use additive decrease (i.e., by subtracting a constant amount every iteration) then they will move along a line parallel to the Fair line. Since the participant with the higher rate reduces by the same amount, fairness does not improve.



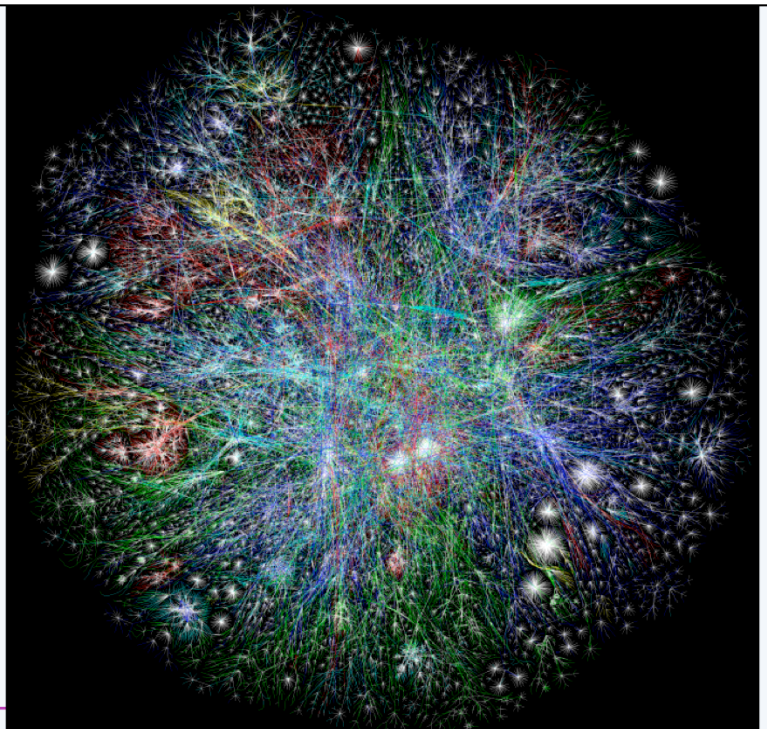
If instead they use addidive increase (AI) and multiplicative decrease (MD), they will approach the goal point. This is the only set of simple adjustments that will achieve this goal.

AIMD Trace

- **AIMD produces sawtooth pattern of window size**
  - **Always probing available bandwidth**



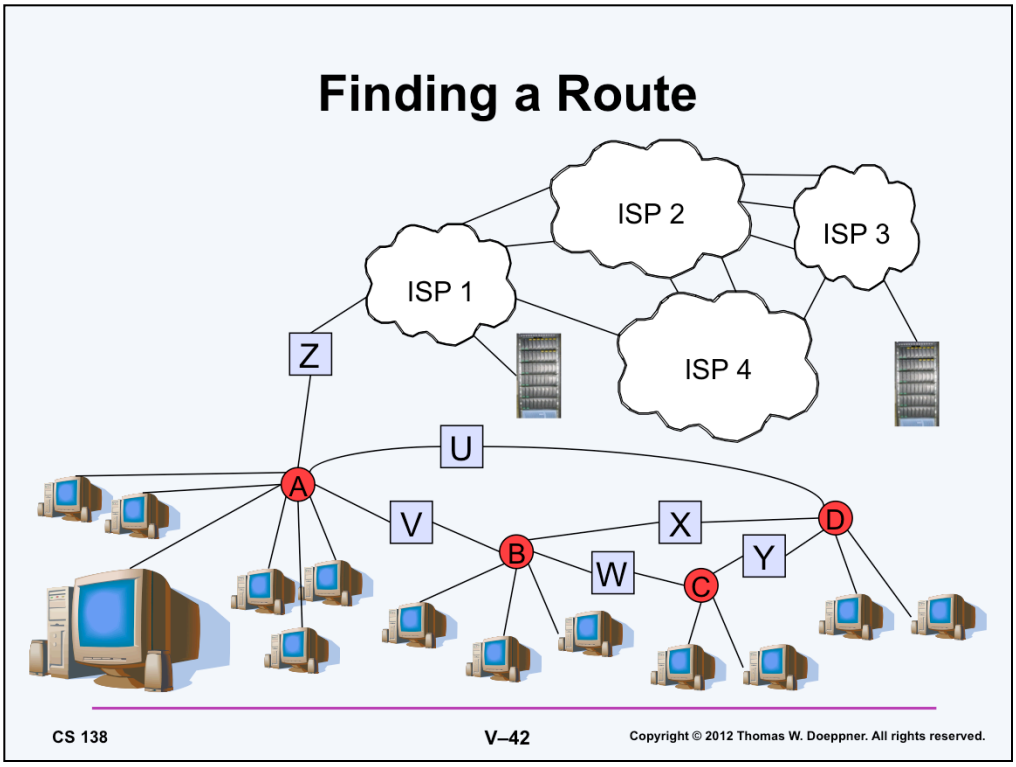
The additive increase is necessary so that nodes probe for new capacity available (maybe some flow just left!). If one looks at the window size over time for a node doing AIMD, it looks like the figure, in a characteristic sawtooth pattern.



Traceroute map of the Internet, ~5 million edges, circa 2003. opte.org

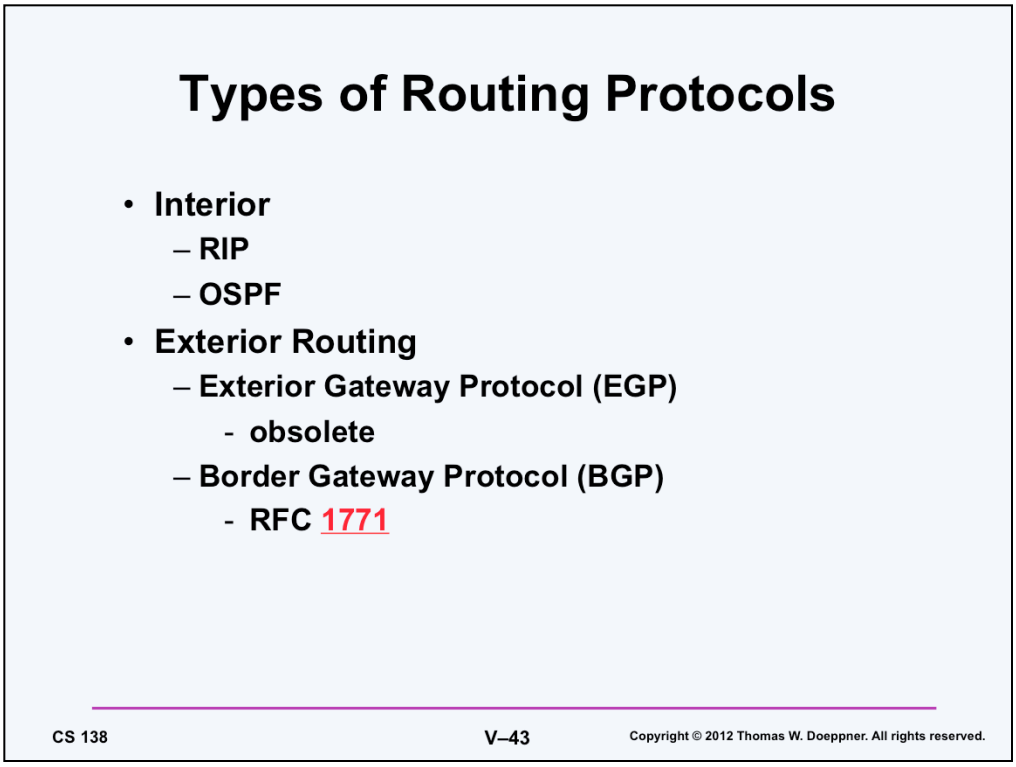
Problem: How to Route at Scale?

- All-to-all routes?
- Central directory?
- We saw one way of doing this scalably: structured P2P networks (Chord, Tapestry)
  - Assume any-to-any connectivity
- Internet uses another approach to scalability: hierarchy
  - Natural fit to how the world is organized
  - Matches how networks are connected and administered
  - IP addresses allocated in hierarchical blocks (mostly)
  - Unit of routing: prefixes



Suppose we're using the computer at the bottom left of the slide. The nodes A through D are Ethernet hubs; Nodes U through Z are routers. If we want to send packets to other machines connected to hub A, no routing is required, since all machines are on the same subnet. However, we'll need some means for determining the route to the machines connected to the other hubs. This routing information could be statically supplied, but, particularly if routers and connections might go down, we're better off if the routing information can be determined dynamically. To communicate with machines served by ISP 1, all our machine needs to know is to send all traffic to router Z.

This could be done by the use of **a default route** in our routing table. Similarly, traffic to all the other ISPs can be directed by simply sending traffic to Z. Within ISP 1 there must be information for routing to all networks served by it, including to the other ISPs. Note that since ISP 1 is connected to multiple other ISPs (and their networks) at multiple points, the "trick" of using a default route in its routers isn't going to work; there must be some routers that know about the networks in the other ISPs.



Routing domains are collections of systems. They exchange routing information with other domains. An *exterior router* (or gateway) is one that exchanges information with routers on other autonomous systems. Its availability and Internet routing tables for other exterior routers are the concern of the entire Internet. On the other hand, the *interior routers* are those IP nodes that are part of an autonomous system whose routing tables are only of concern to members of that IP network. Both interior and exterior routers have to provide their own routing information. In the case of the public Internet, the exterior routing protocol was EGP a number of years ago, but more recently *border gateway protocol* (BGP) has been adopted. Each autonomous system is free to choose its own interior routing (or gateway) protocol (IGP).

Though RIP is still used as an IGP, OSPF and IS-IS (another link-state protocol) are used on the major autonomous systems.

Dynamic Routing

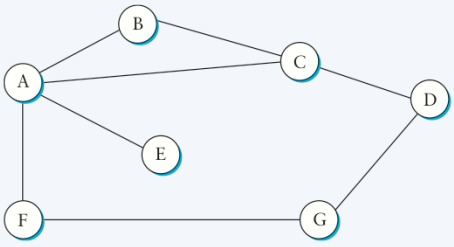
- Distance vector
  - each router maintains table containing best known distance to each destination and the first hop of the associated path
  - routers periodically exchange tables with their neighbors
- Link state
  - each router maintains table containing description of entire network
  - each router computes best route based on global information
  - routers periodically send all others their local network-description information (i.e., the state of their connections with neighbors)

There are two basic approaches to maintaining routing information. The first, the distance-vector approach, involves having neighboring routers periodically exchange their complete routing tables. When a router gets a neighbor's table, it examines it to see if it contains better routes than what it has now and to check if existing routes that go through that neighbor have changed (or have been eliminated).

The other approach, the link-state approach, has routers sending to all others only the state of their local connections, and then each router independently computes the shortest paths from it to all others.

Both approaches have significant challenges when it comes to scaling.

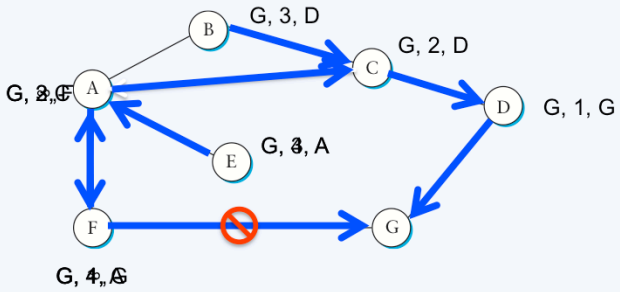
DV Example



B's routing table

Destination	Cost	Next Hop
A	1	A
C	1	C
D	2	C
E	2	A
F	2	A
G	3	A

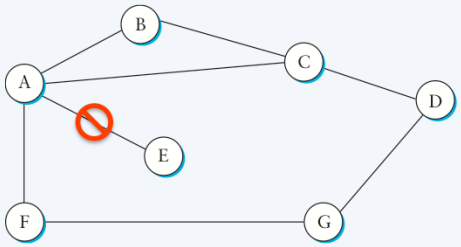
Adapting to Failures



- F-G fails
- F sets distance to G to infinity, propagates
- A sets distance to G to infinity
- A receives periodic update from C with 2-hop path to G
- A sets distance to G to 3 and propagates
- F sets distance to G to 4, through A

Does the order matter here?

Count-to-Infinity



- Link from A to E fails
- A advertises distance of infinity to E
- B and C advertise a distance of 2 to E
- B decides it can reach E in 3 hops through C
- A decides it can reach E in 4 hops through B
- C decides it can reach E in 5 hops through A, …
- When does this stop?

How to avoid loops

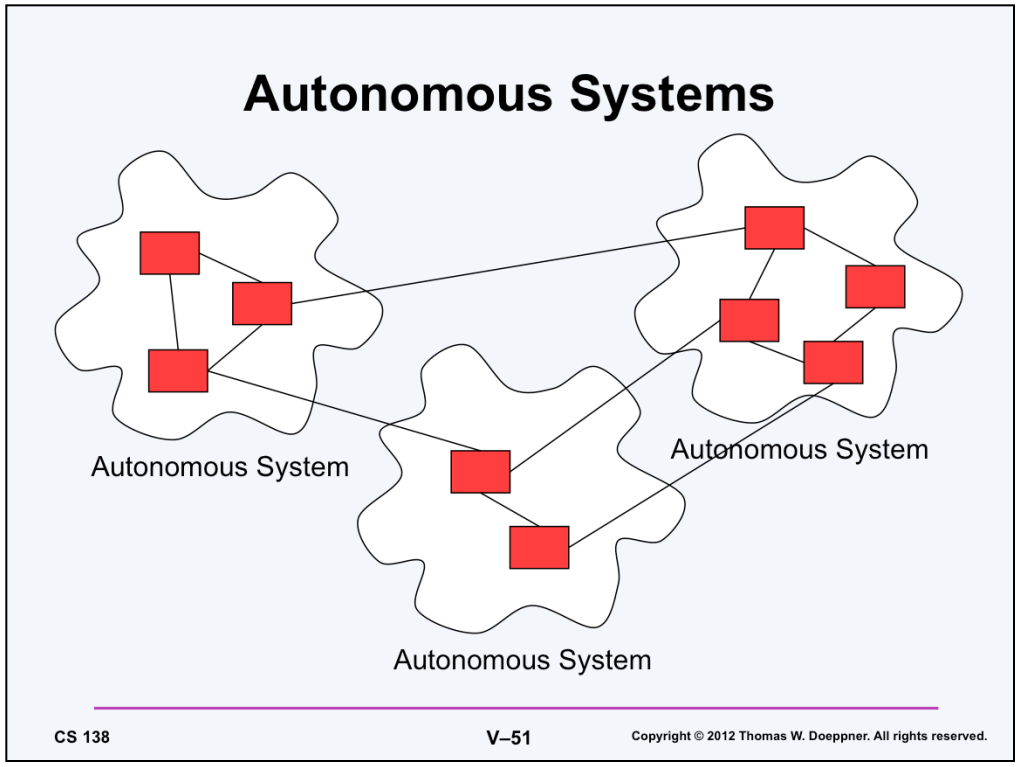
- IP TTL field prevents a packet from living forever
  - Does not *repair* a loop
- Simple approach: consider a small cost *n* (e.g., 16) to be infinity
  - After *n* rounds decide node is unavailable
  - But rounds can be long, this takes time
- Problem: distance vector based only on local information`

Better loop avoidance

- Split Horizon
  - When sending updates to node A, don't include routes you learned from A
  - Prevents B and C from sending cost 2 to A
- Split Horizon with Poison Reverse
  - Rather than not advertising routes learned from A, explicitly include cost of ∞.
  - Faster to break out of loops, but increases advertisement sizes

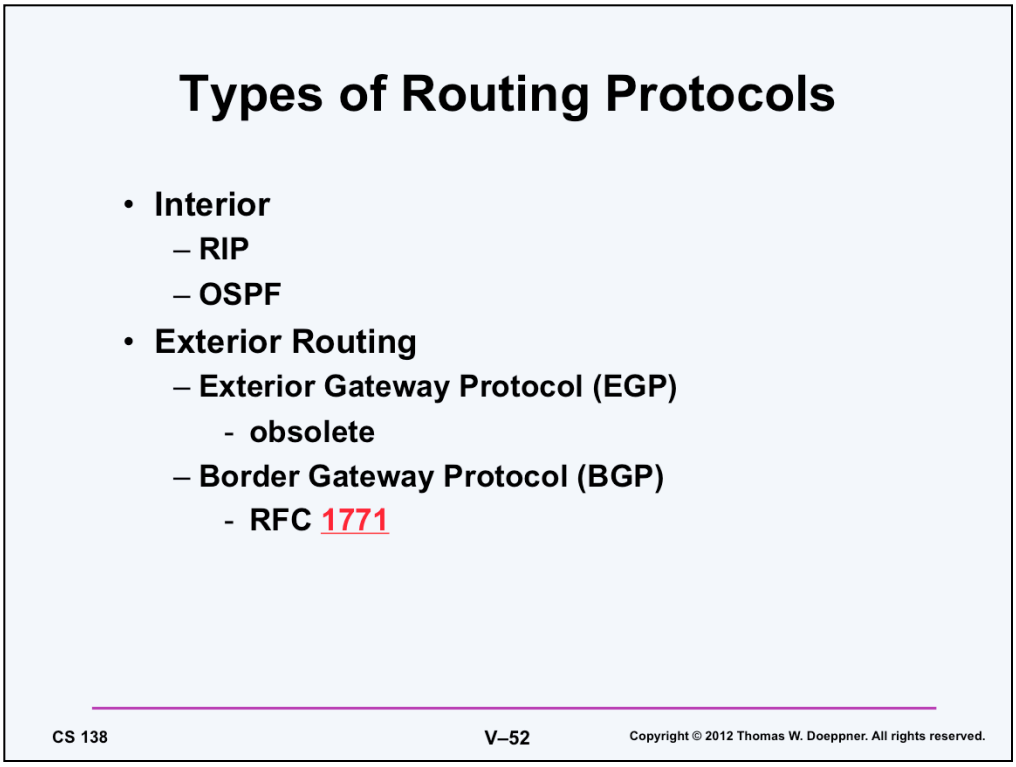
Warning

- Split horizon/split horizon with poison reverse only help between two nodes
  - Can still get loop with three nodes involved
  - Might need to delay advertising routes after changes, but affects convergence time



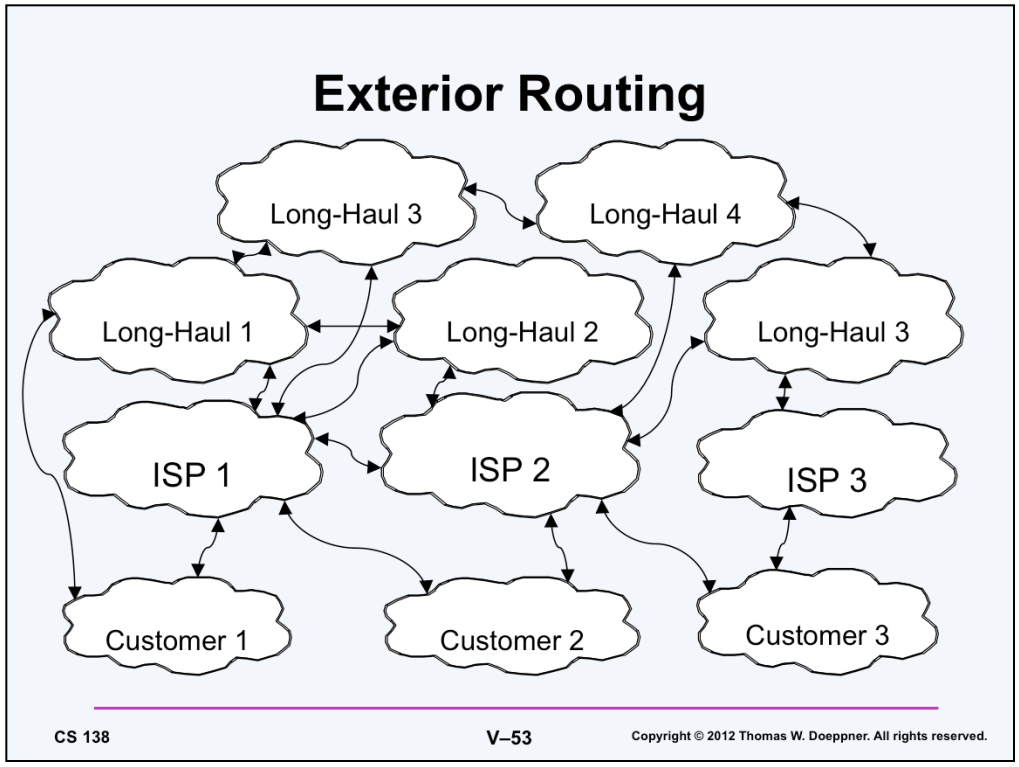
Dynamic routing involves **policy**: something must determine the routes for packets to take, taking into account current conditions.

Rather than use a single policy (and algorithm) for the Internet as a whole, **the Internet is divided into *autonomous systems*** (what, up to now, we've been calling *domains*), each of which does its own routing for internal traffic. Such an autonomous-system-specific routing algorithm is called an *interior routing protocol*. We, of course, must also worry about traffic between autonomous systems. This is dealt with by ***exterior routing protocols***.



Routing domains are collections of systems. They exchange routing information with other domains. An *exterior router* (or gateway) is one that exchanges information with routers on other autonomous systems. Its availability and Internet routing tables for other exterior routers are the concern of the entire Internet. On the other hand, the *interior routers* are those IP nodes that are part of an autonomous system whose routing tables are only of concern to members of that IP network. Both interior and exterior routers have to provide their own routing information. In the case of the public Internet, the exterior routing protocol was EGP a number of years ago, but more recently *border gateway protocol* (BGP) has been adopted. Each autonomous system is free to choose its own interior routing (or gateway) protocol (IGP).

Though RIP is still used as an IGP, OSPF and IS-IS (another link-state protocol) are used on the major autonomous systems.



The Internet is composed of a large number of autonomous systems with a fairly complicated set of connections. Given that we solved the routing problems within autonomous systems, how do we route across autonomous systems? It's not going to be as simple (or as complicated) as finding the shortest possible path: we have to be concerned about a fair number of policy issues, such as which autonomous systems are allowed to carry the traffic of other autonomous systems.

Addressing

- Each AS has a 16-bit AS number
  - in short supply … later increased to 32 bits
- AS advertises which prefixes it owns
- Ideally
  - easy mapping from IP address to AS number
- But no …
  - people move machines
  - highly decentralized (hierarchical) process
  - change ISPs
- Result
  - routers with no default entries have *lots* of entries
    - Over 500,000 entries as of 2015

In an ideal world, the routing tables of exterior routers would contain one entry for each autonomous system (and no other entries). This would be possible if there were a well defined mapping from IP address to autonomous-system number. Unfortunately there isn't. So, there must be a separate entry for each active class-A, -B, and -C address, as well as each CIDR prefix. The information on the current size of the tables are from http://www.telstra.net/ops.

BGP

- References: RFCs 1771, 1772, 1773, 1774
- A path-vector protocol
  - unit of routing is the AS
  - avoids count-to-infinity problems by using path vectors (of ASs)
  - routers must enforce policy constraints
  - no notion of determining optimal routes: reachability is the goal

BGP (border gateway protocol) is the standard exterior routing protocol in use on the Internet, replacing EGP. A good description of the protocol is given in "BGP4: Inter-Domain Routing in the Internet," John W. Stewart III, Addison-Wesley 1999.

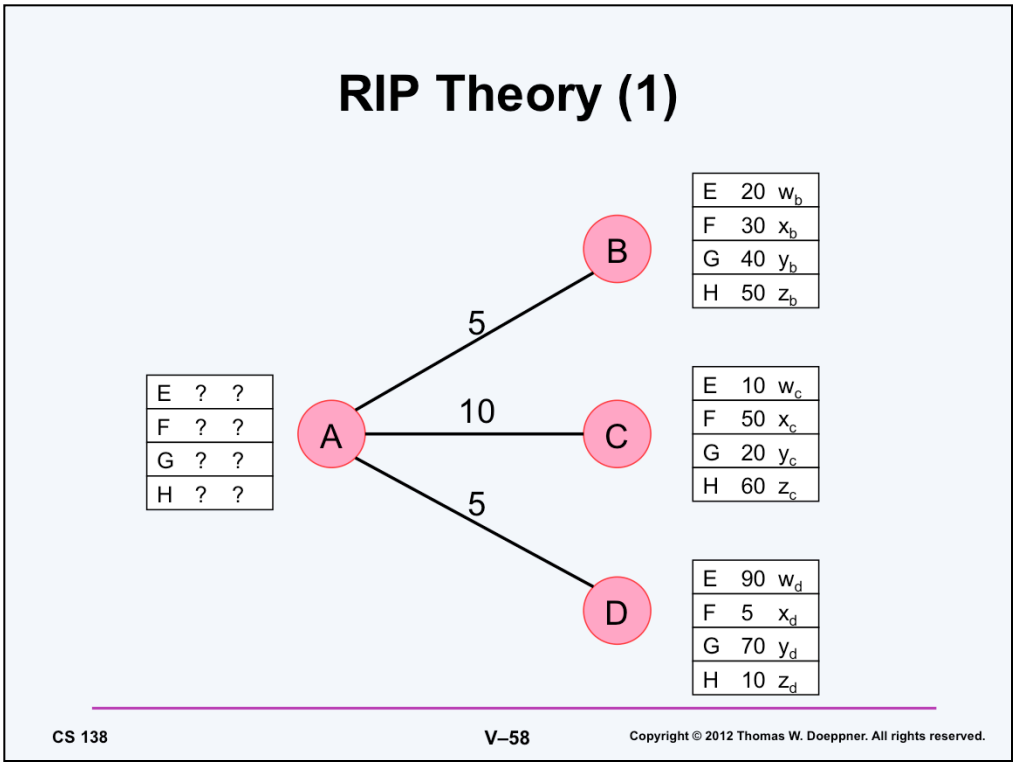
Extra Slides

- These were not covered in class, but are really good if you want to understand classic routing protocols better.

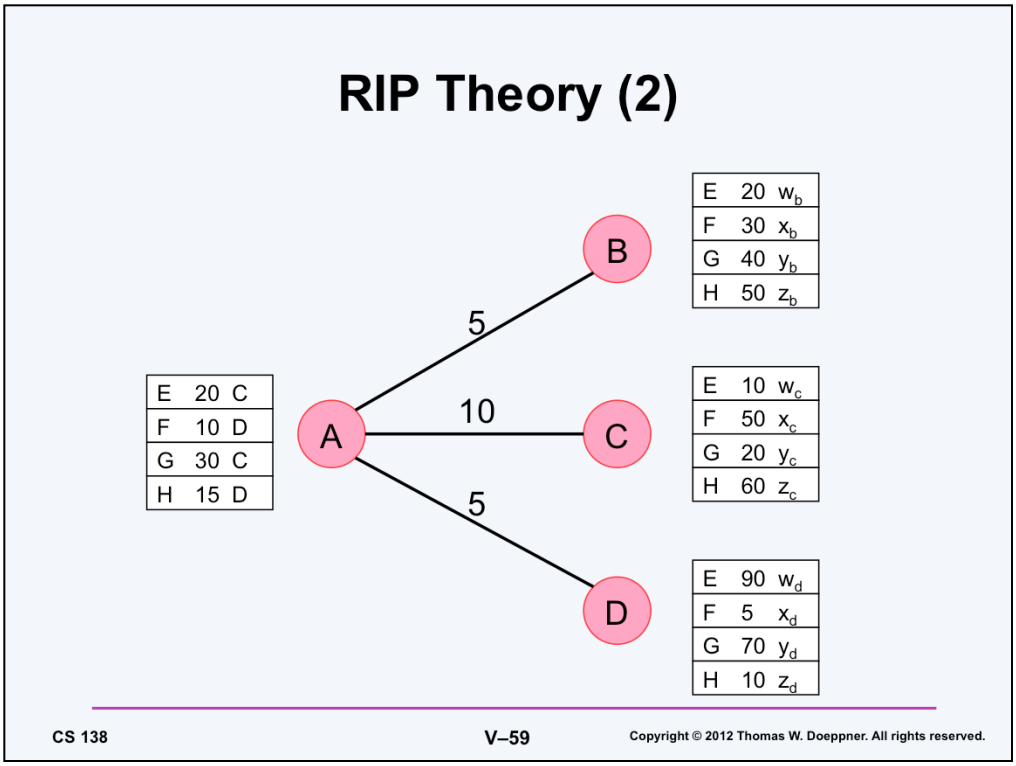
Routing Information Protocol (RIP)

- Distance-vector protocol
- In use since the ’70s
- Formerly in widespread use
  - implemented on Unix as *routed*
- Described in RFCs 1058, 1723

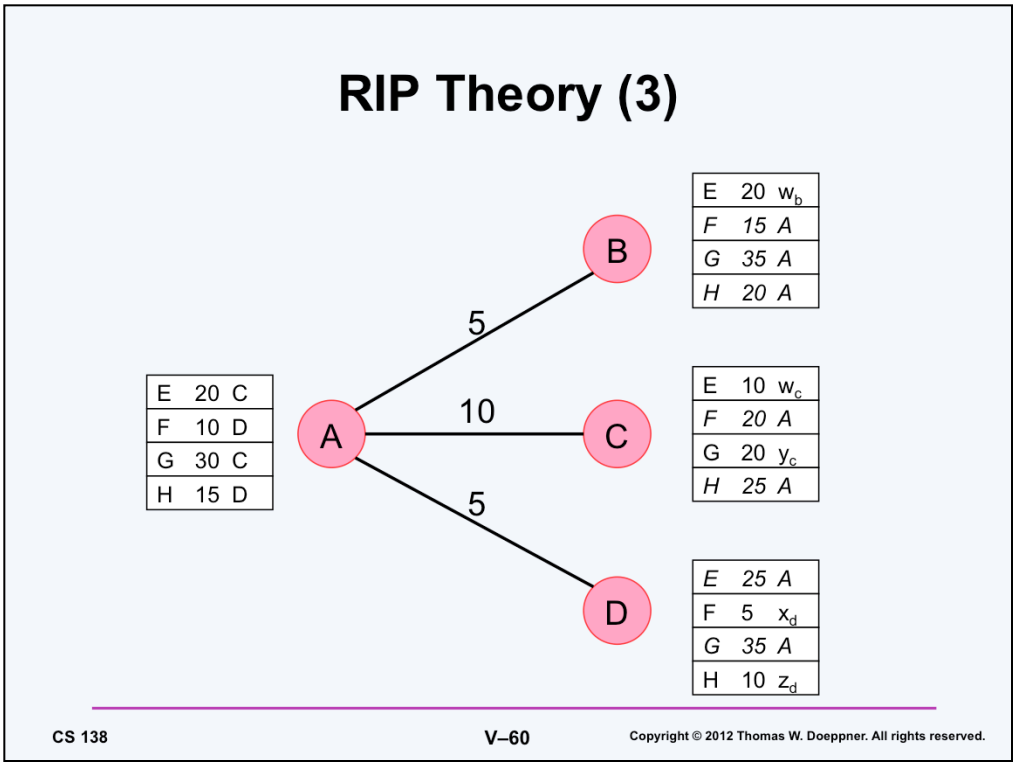
The first routing protocol we examine is *routing information protocol*, otherwise known as *RIP*. Although there are many limitations on what it can do, it has had widespread use, primarily because of its simplicity, though its use is dwindling. (It was originally designed to run on LSI-11 computers, a very limited (in speed and memory) minicomputer built in the 1970s.)



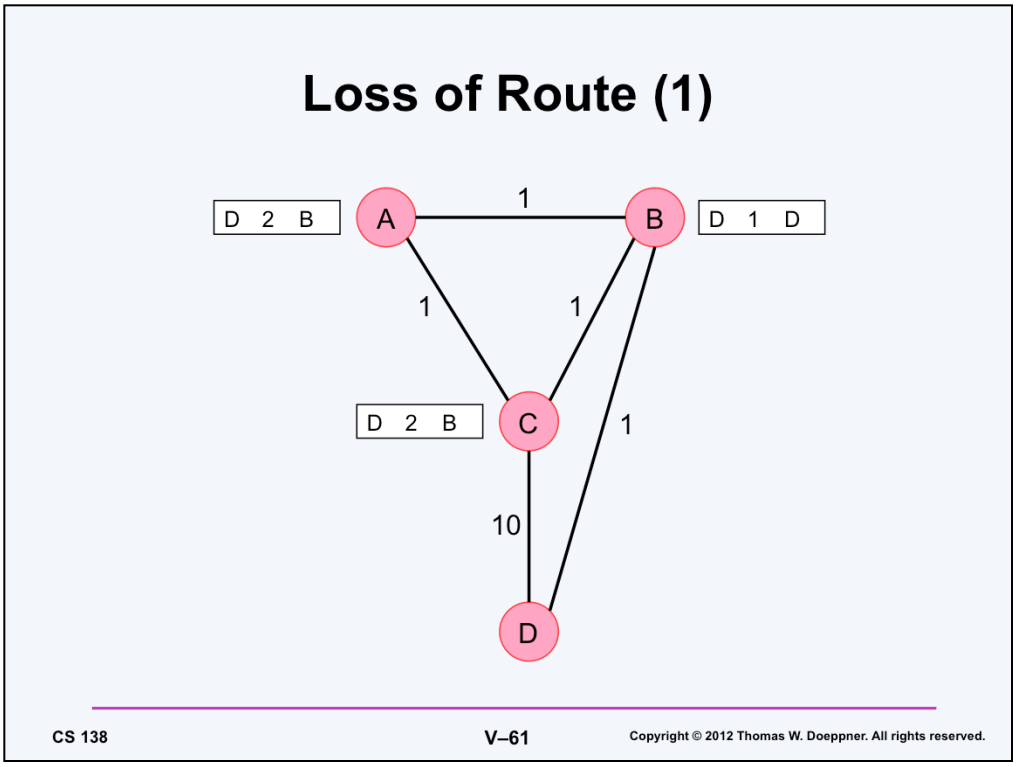
Being a distance-vector protocol, each RIP router periodically exchanges its routing table with its neighbors. The routing tables contain one entry for each destination, containing the cost of the best known path to that destination and the first hop (i.e., the neighboring router to send packets to) for that path. In the example of the slide, A is obtaining the route tables of B, C, and D. The costs of the connections between A and these nodes are marked on the edges between them. The table entries are triples indicating a path to the first component, of cost given by the second component, whose first hop is to the third component.



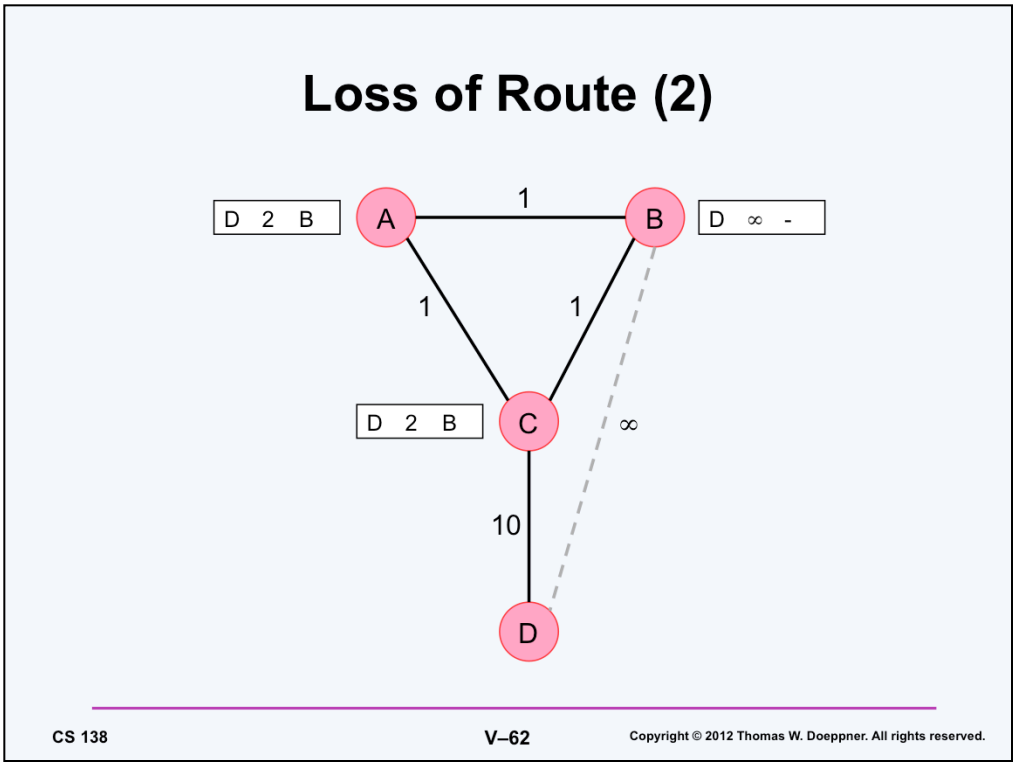
A's routing table is initialized to the shortest known paths according to information received from its neighbors.



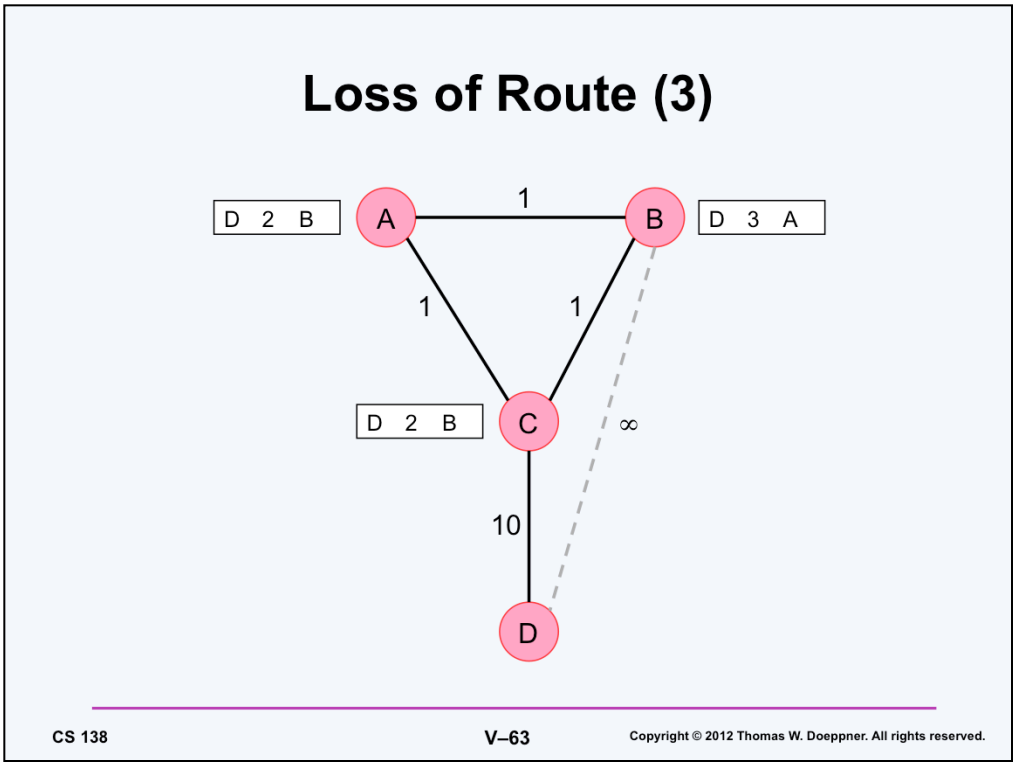
A sends its routing table to its neighbors, who modify theirs to account for routes through A.



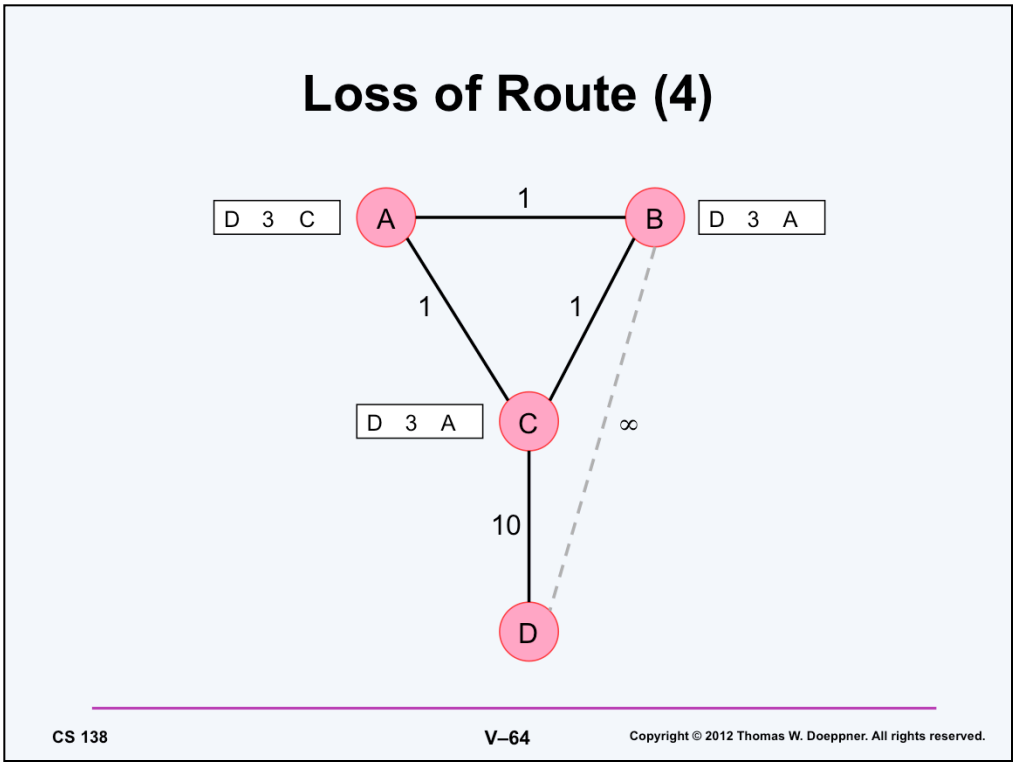
Besides learning about new routes, the table updates inform routers about the loss of routes. Consider the situation shown here, in which we are interested in routes to node D.



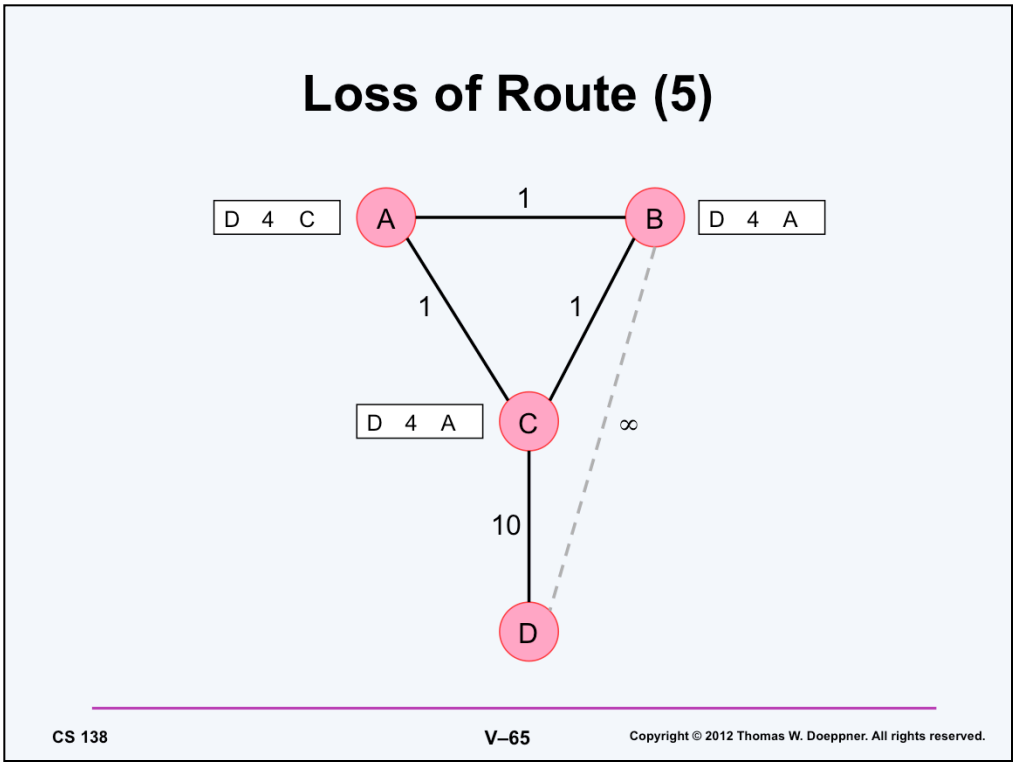
The low-cost route from B to D suddenly vanishes. B must remove its route to D from its routing table.



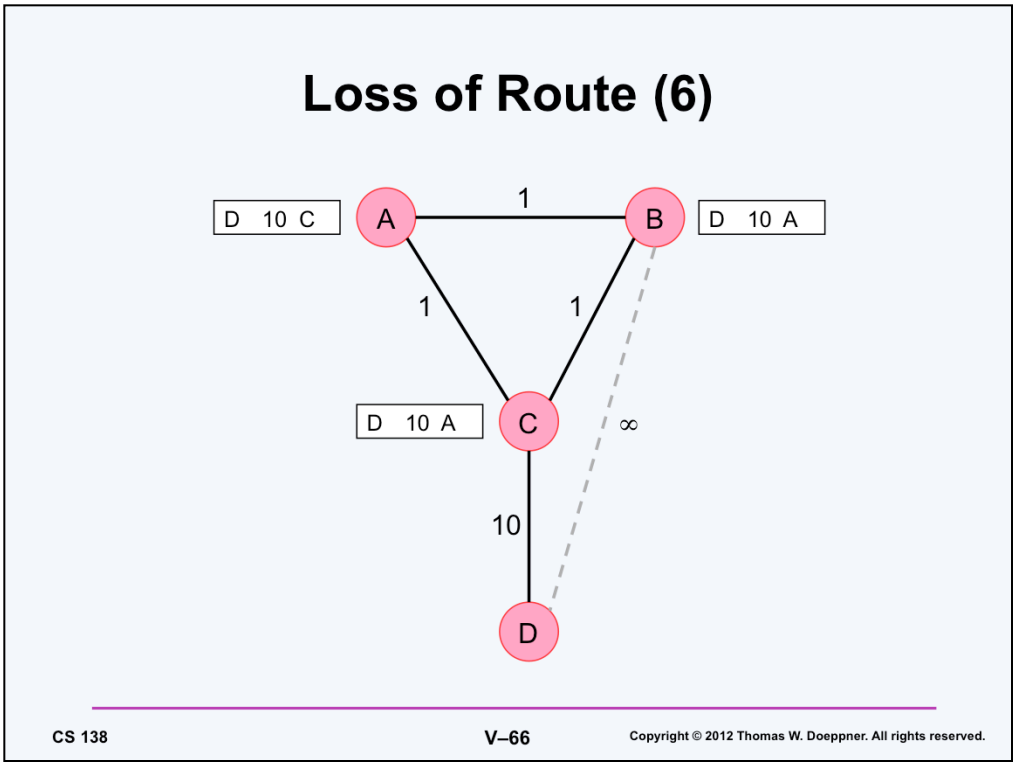
B gets a routing table update from A, which claims to have a route to D of cost 2 (it doesn't realize that it goes through B …); so B updates its route to D to be a cost-3 route through A.



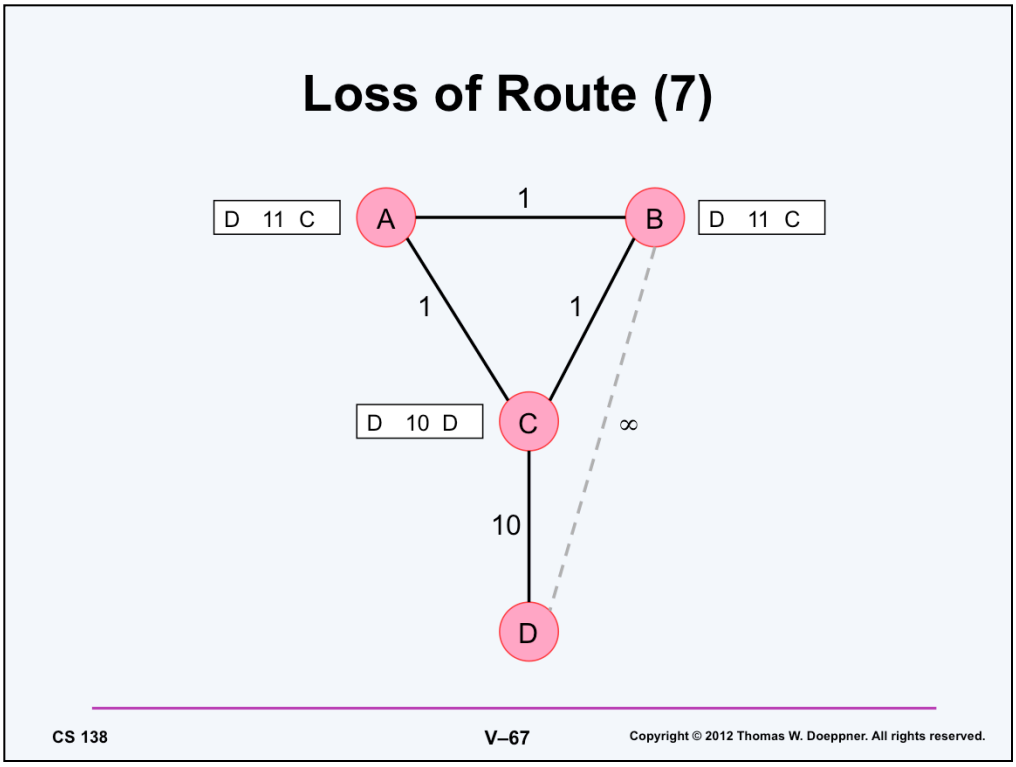
A and C discover that there no longer is a low-cost route through B, but each sees the other's cost-2 route to D through B, so they update their routing tables to get to D through each other via a cost-3 route.



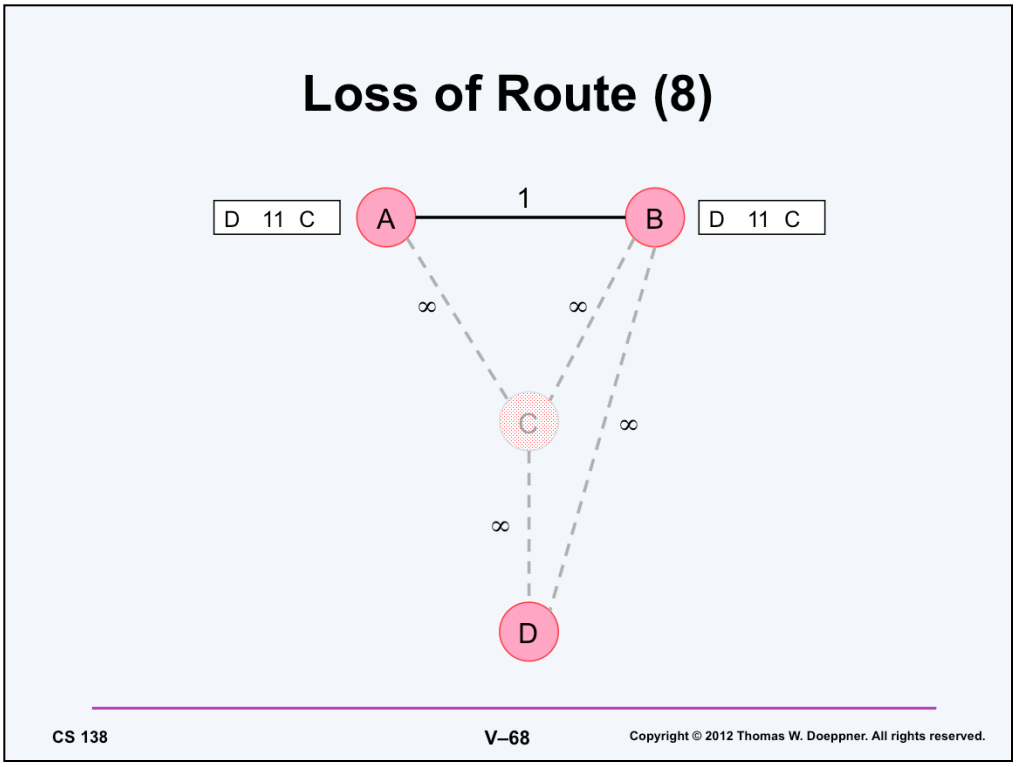
Nodes A, B, and C, after again exchanging routing tables, realize that the nodes through which their routes pass through are more expensive than the last time, so they update their routing tables accordingly.



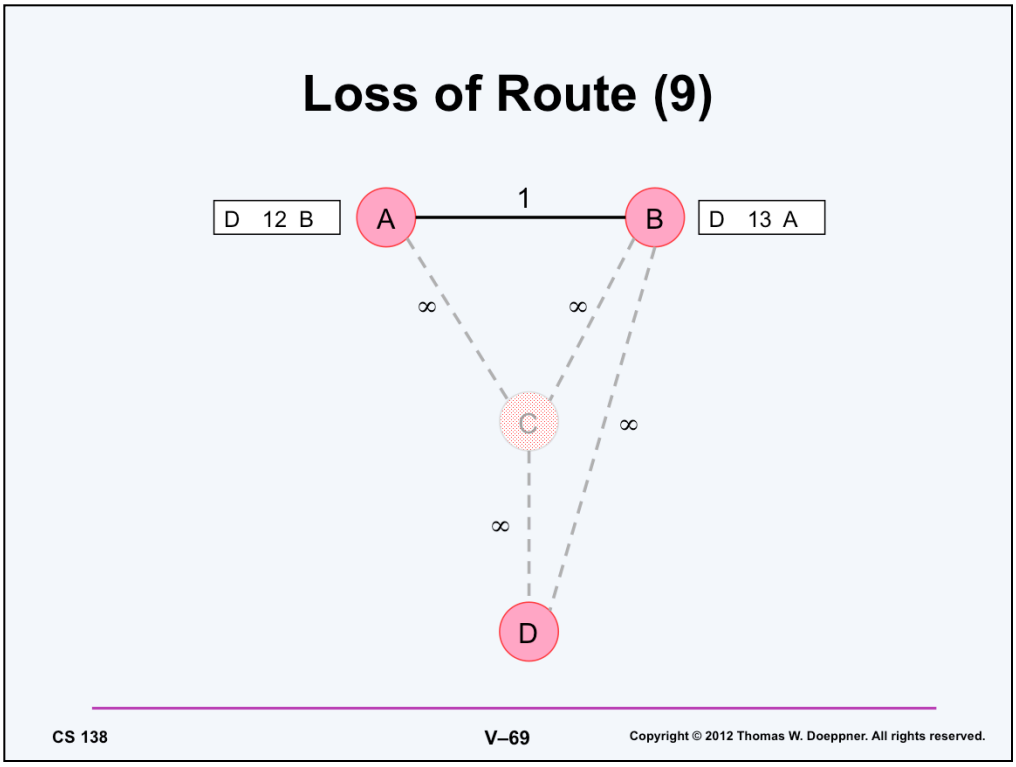
After this procedure is repeated a number of times, each node's table now has a cost-10 route to D, at which point C's direct connection (of cost 10) begins to look attractive.



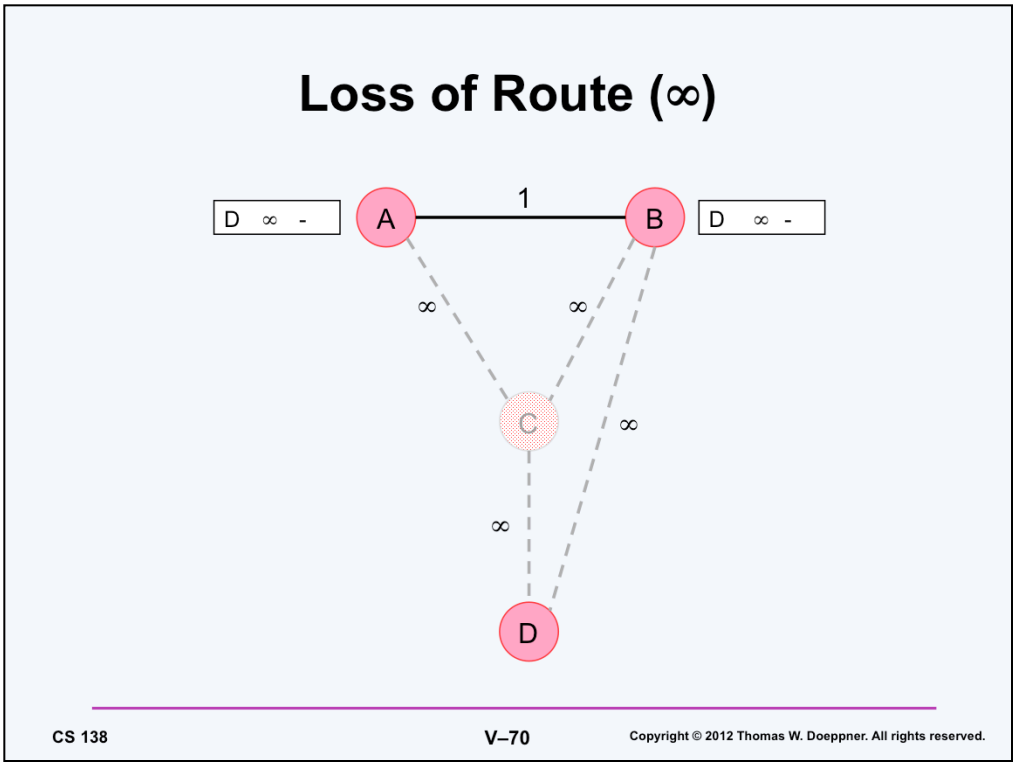
C now advertises its cost-10 route, which is adopted by the others. Things are now stable again.



However, node C suddenly disappears ...



A and B repeatedly exchange routing information; each wants to route to D via the other.



Since there is no alternative, high-cost route to D, this repeated exchange of routing tables with corresponding increase in route cost never terminates (or, at least not until the route cost reaches infinity …).

Counting to Infinity

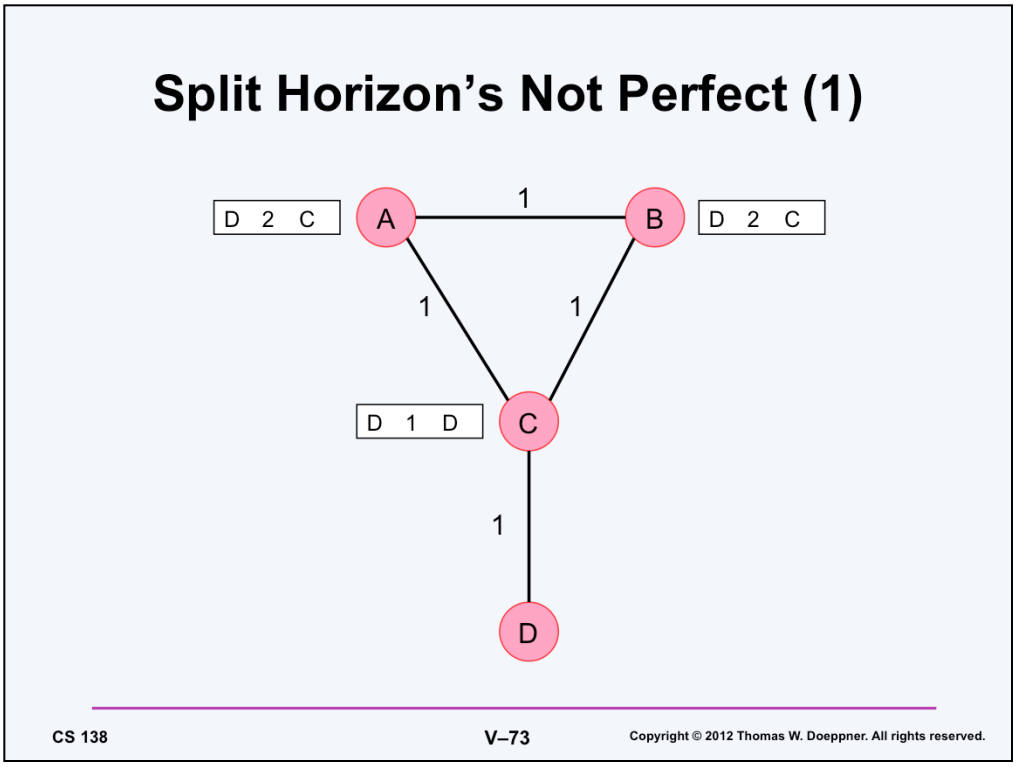
- Make infinity small
  - 16 is chosen as infinity in RIP
  - longest path in network can be no longer than 15

The only way out of this problem is to redefine infinity to be a small number (if only everything else was so easy …).

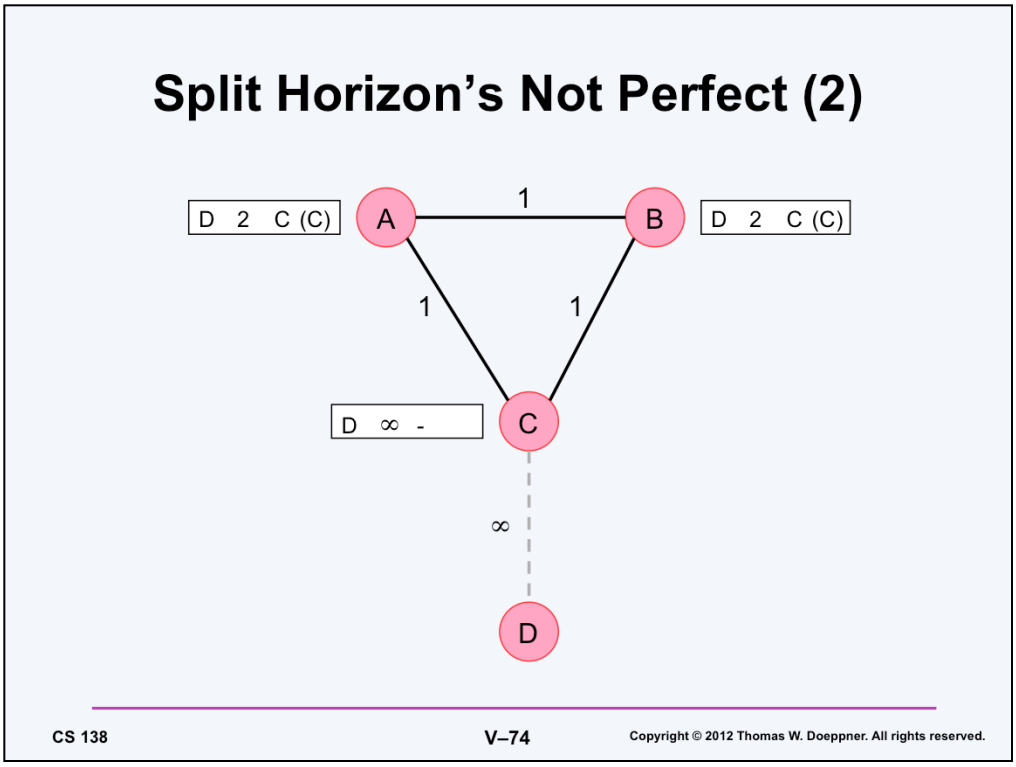
Split Horizon

- Don't tell neighbor about routes obtained from it
  - quicker convergence in many cases

One way to improve convergence is to use a technique known as *split horizon* (a technique used by RIP). It uses the common-sense principle that it's not necessary to inform a neighbor about routes that the neighbor provided in the first place. Thus in our previous example, node A would not report to node B that it has a route to D, since it obtained that route from B itself. Thus B would discover quickly that it has no route to D. A, of course, must realize that when B reports no route to D, this means that the route that A previously had to D via B is now gone.

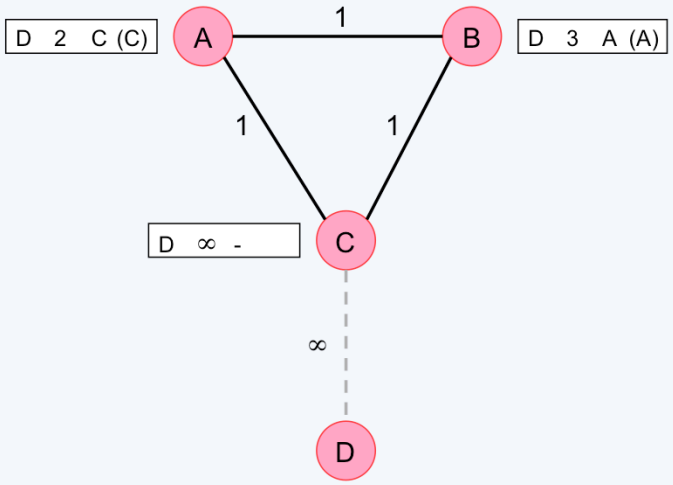


However, even with split horizon we're still not immune from the counting-to-infinity problem, as shown in the example in this and the next few slides.

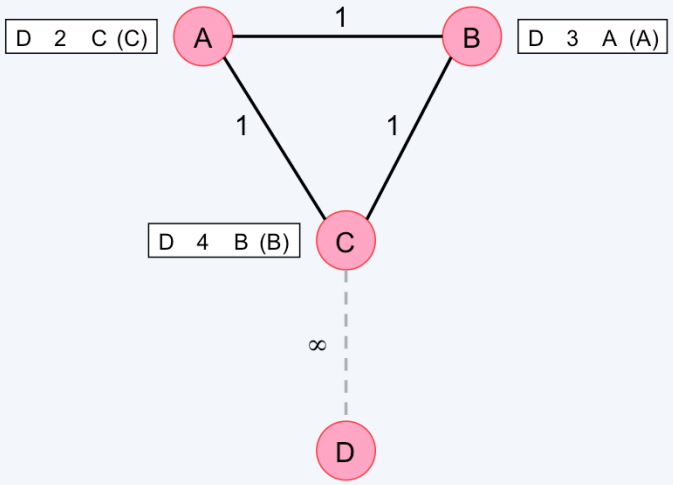


Here we augment our routing-table entries with a field indicating who supplied the route.

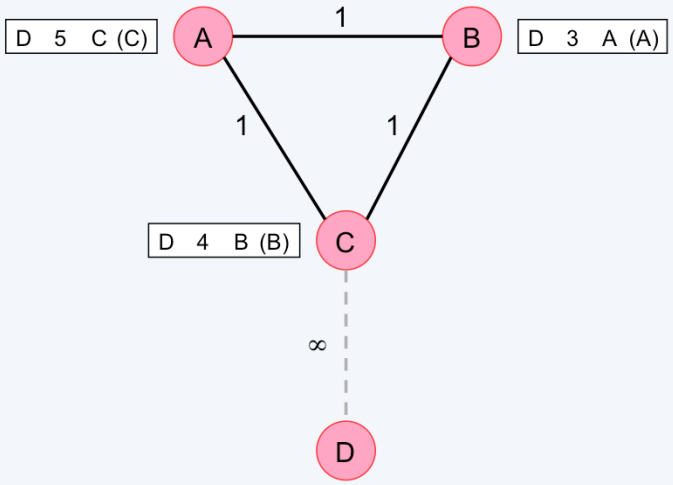
Split Horizon's Not Perfect (3)

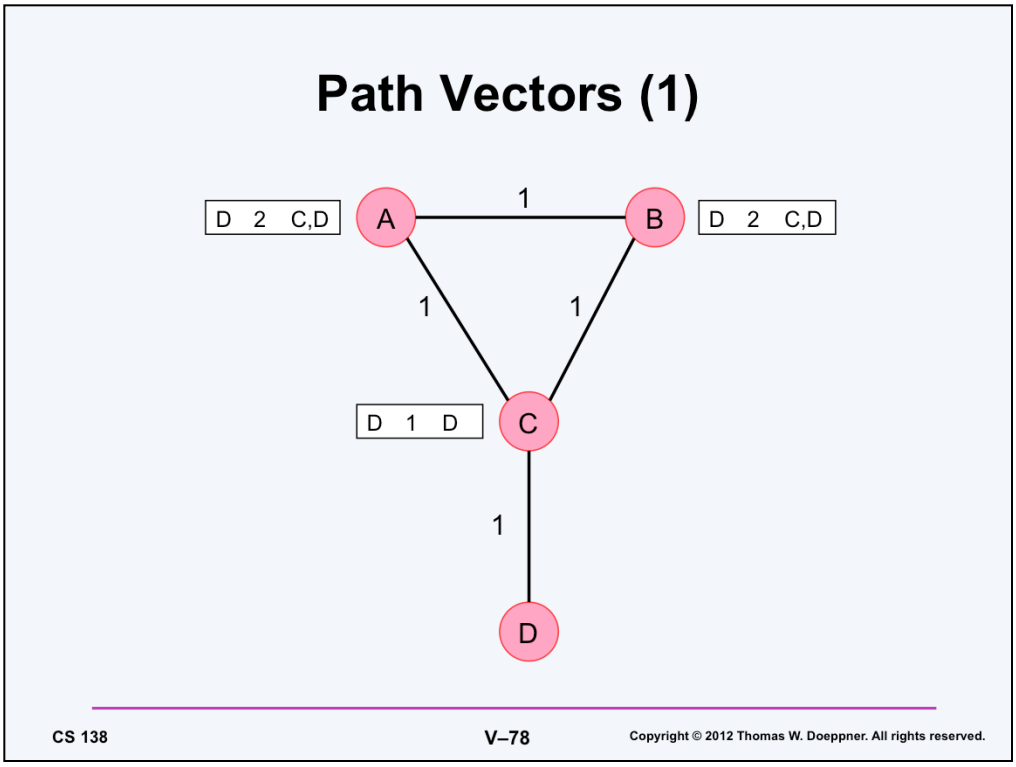


Split Horizon's Not Perfect (4)

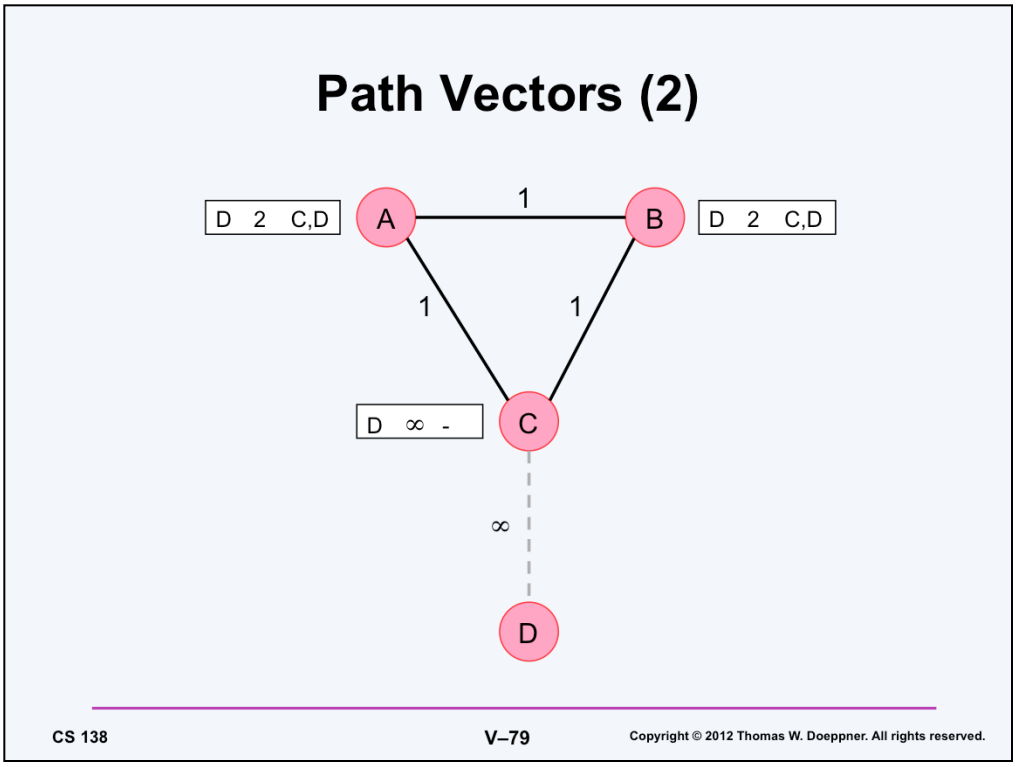


Split Horizon's Not Perfect (5)

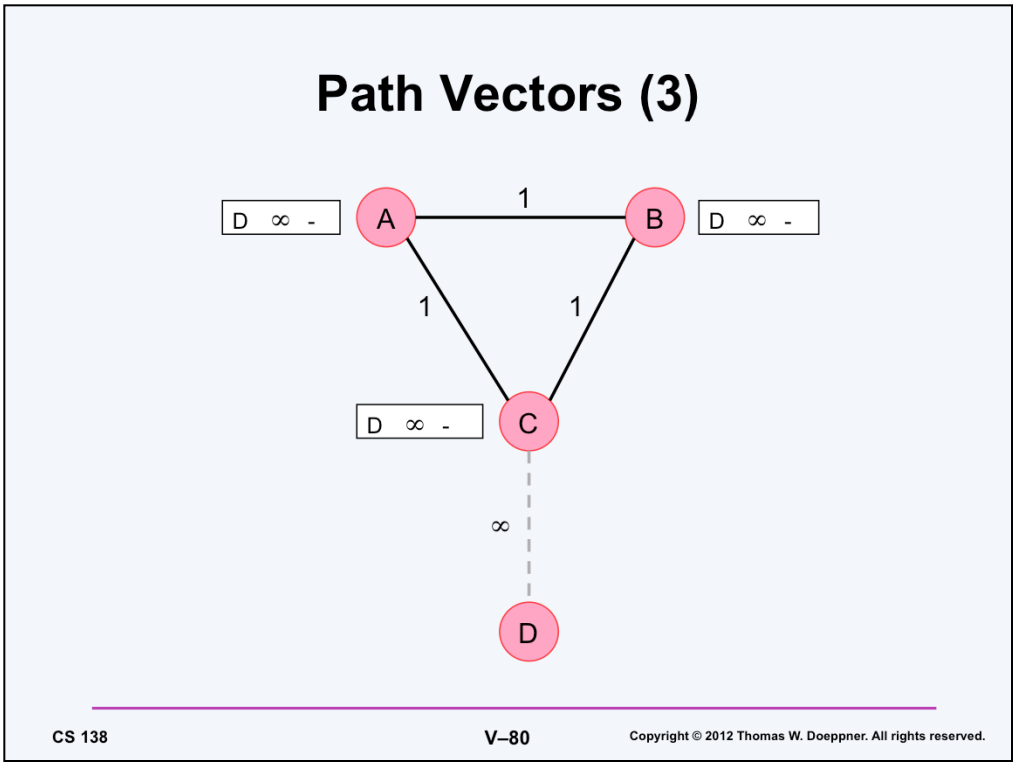




An approach that actually prevents counting-to-infinity problems is the *path-vector* approach, in which nodes store the complete paths in the routing-table entries, not just the first hop.



Here the link from C to D goes down.



Nodes A and B discover that C no longer advertises a route to D. They might both try to route through the other to get to D. (For example, as soon as A discovers that C is down, it might route packets headed for D through B.) However, at the next exchange of routing information, A will see that B's route goes through A, and vice versa. Since such routes would be infinite in length, they are quickly eliminated.

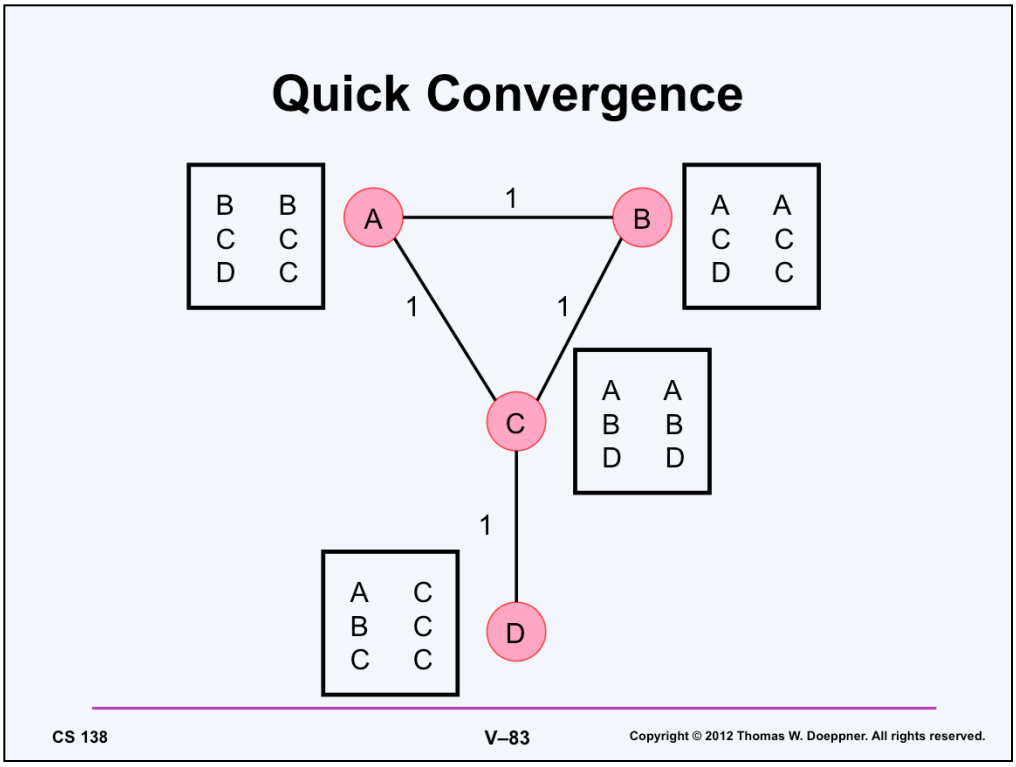
Note that path vectors are not used in RIP, presumably because they would increase the size of routing tables by a fair amount.

RIP Problems

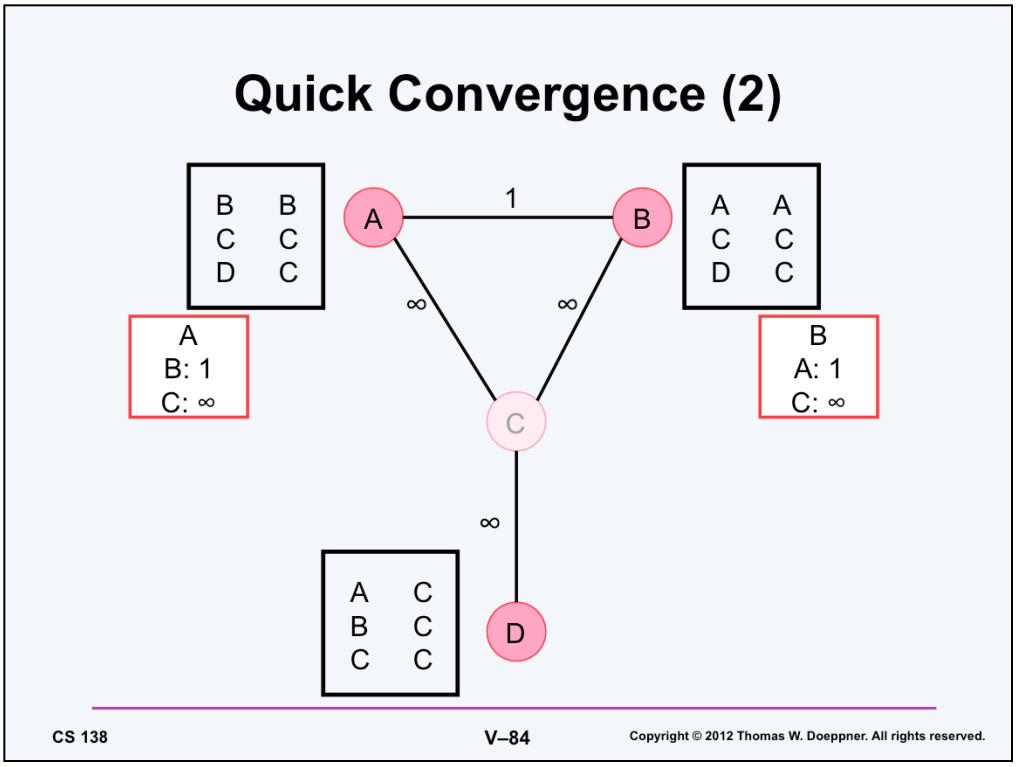
- Slow convergence
- Infinity is too small
  - increasing infinity causes backwards compatibility problems and lengthens convergence
- Routing metric is number of hops
  - could easily be modified …
- Utilizes only a single path, even if multiple paths are possible

To the Rescue: Link-State Routing

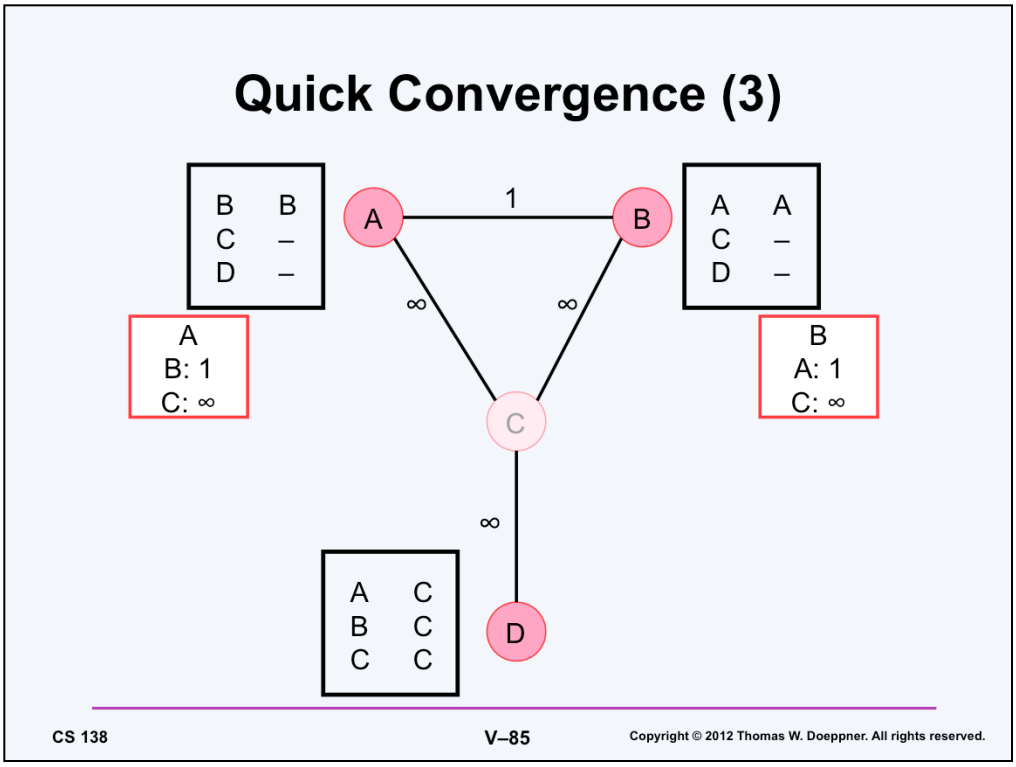
- Routers flood internet with information about their direct links
- Each router computes shortest path independently, using global information



Here we have a simple network with each node's routing table.



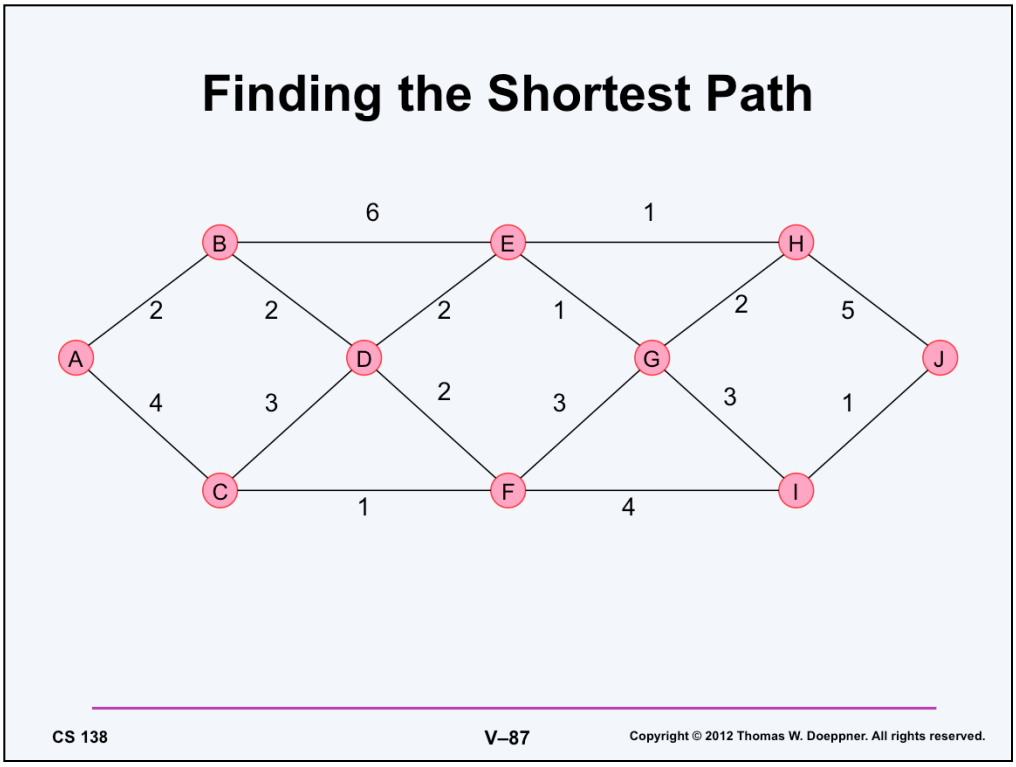
Node C disappears. Nodes A and B send link-state messages to each other.



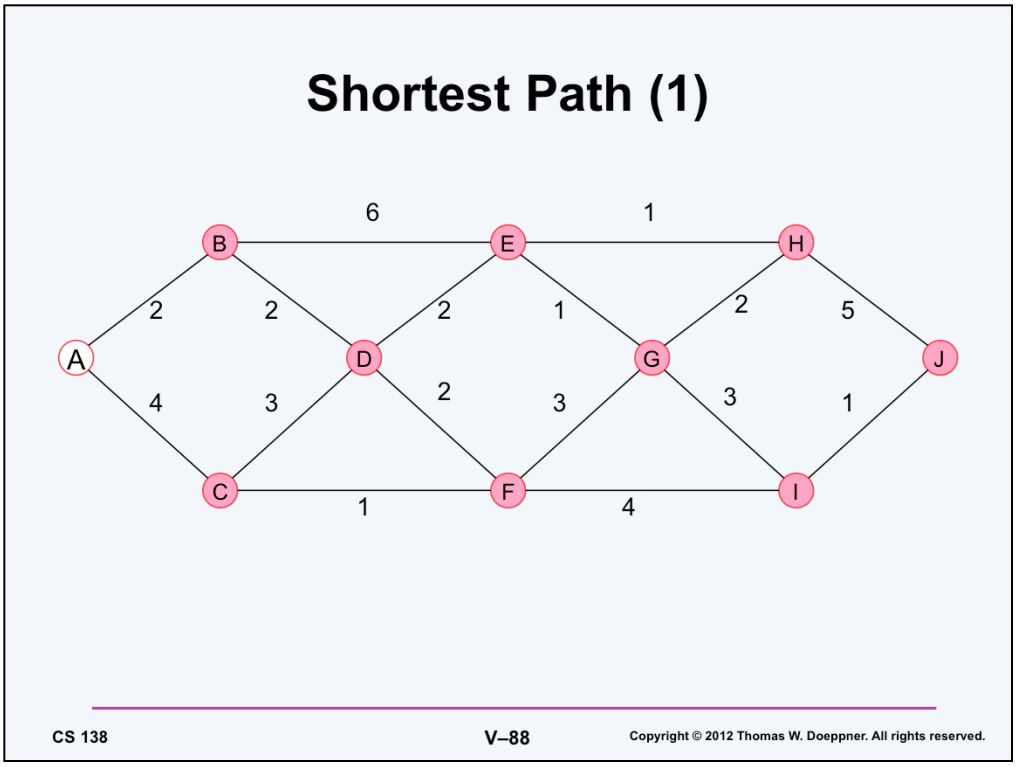
On receipt of the link-state messages, nodes A and B quickly revise their routing tables.

Link-State Advantages

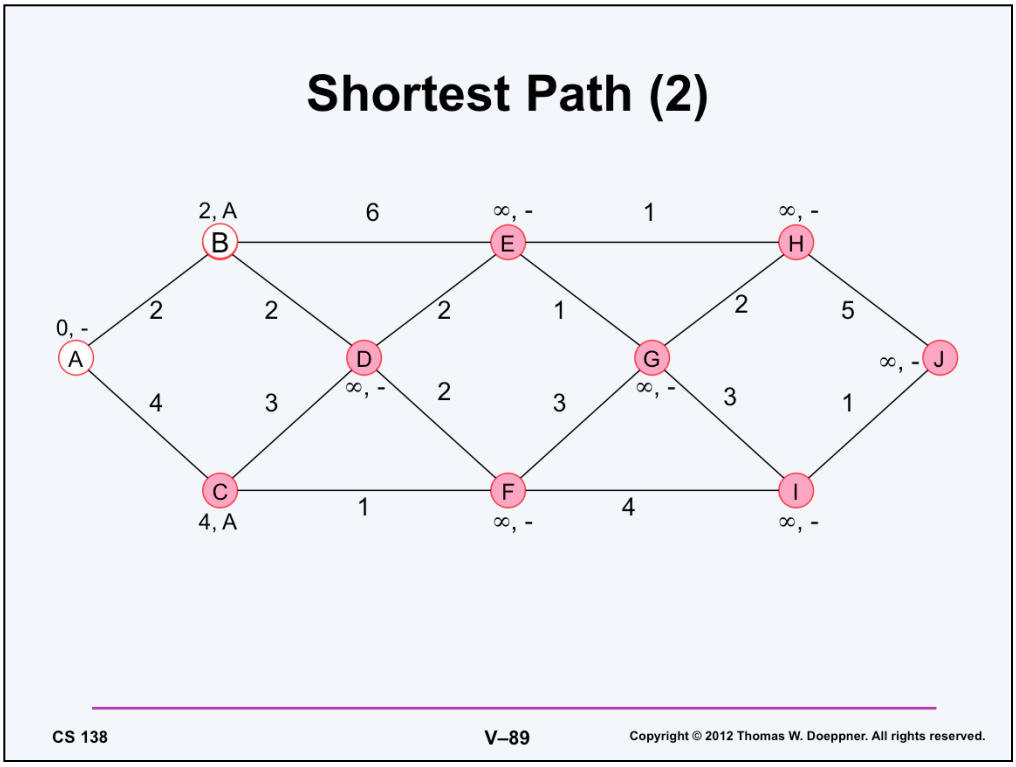
- Quicker convergence
- Load sharing
- Better scaling



In link-state routing, each router computes the shortest path from itself to all other nodes using global information about the network. The slide shows a sample network, with the edges representing communication connections between computers, labeled with the cost of using that connection.

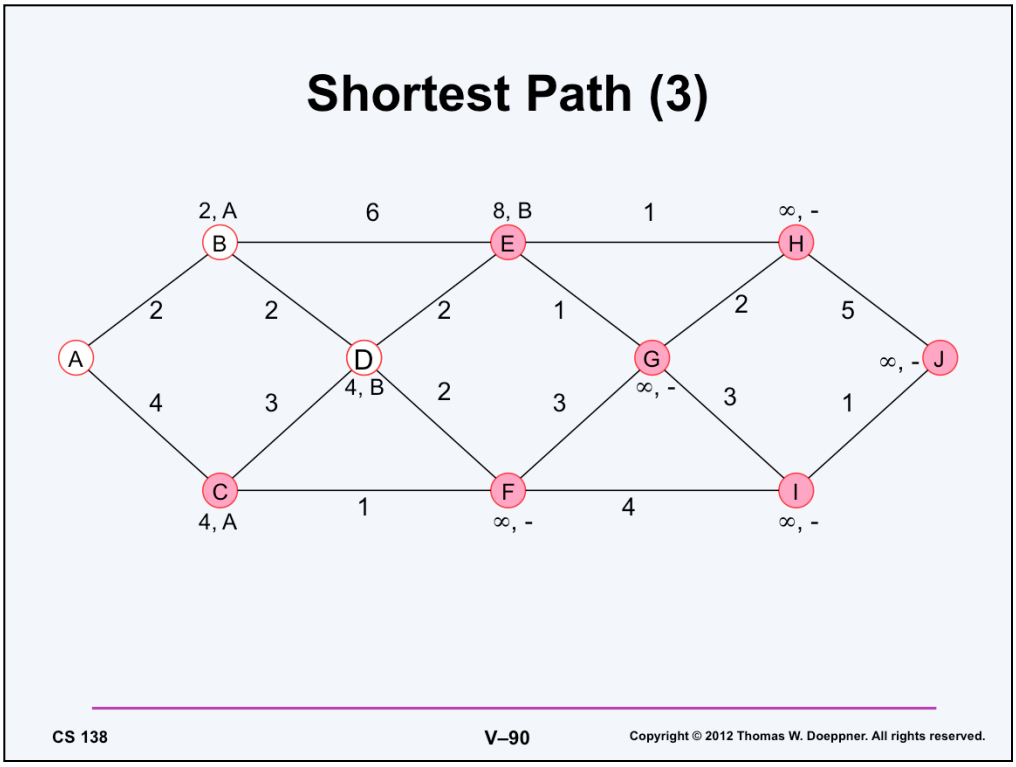


In this and the next few slides, we use *Dijkstra's algorithm* to compute the shortest path between *A* and each of the other computers in the network.



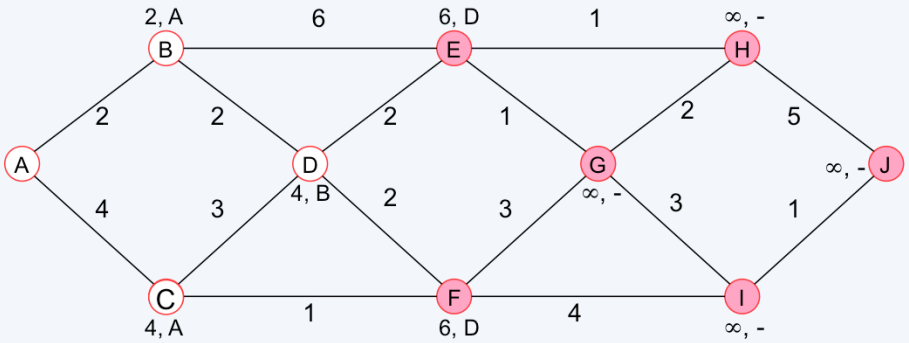
We label each node with a pair indicating the cost of the best known path to that node (from A) and the name of the node just before this one on that path. A labeling is marked *permanent* if we've computed the cheapest path to that node, otherwise it is marked *tentative* (shown as a dark (red) node in the slide). We start with A and set its cost to zero. We take the tentative node with the lowest-cost path from A (A itself, initially), mark it as permanent, then examine the cost of extending this lowest-cost path to each node adjacent to it. I.e., we add to the path the cost of the connection to the adjacent node. If this new path is cheaper than the one already in place for the adjacent node, we change the node's labeling to reflect the new, cheaper path. We continue this procedure until all nodes are permanent.

Thus, in our example, we start with node A and modify the labels of nodes B and C. B now is the tentative node with the lowest cost, so we make it permanent and continue the process with its neighbors.



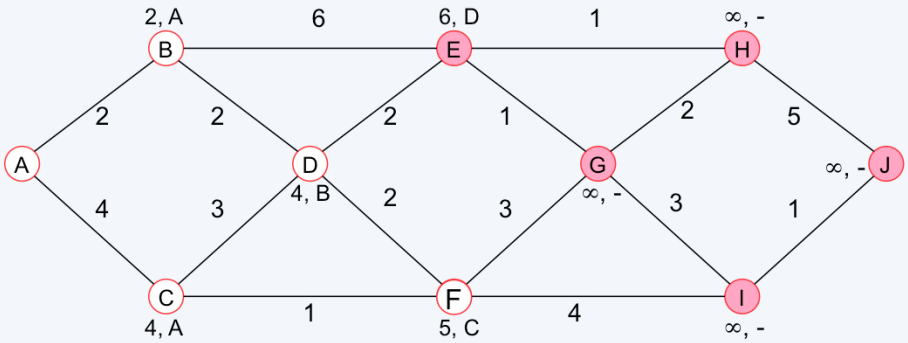
D becomes the cheapest (or tied for cheapest) tentative node, so we continue by using it.

Shortest Path (4)

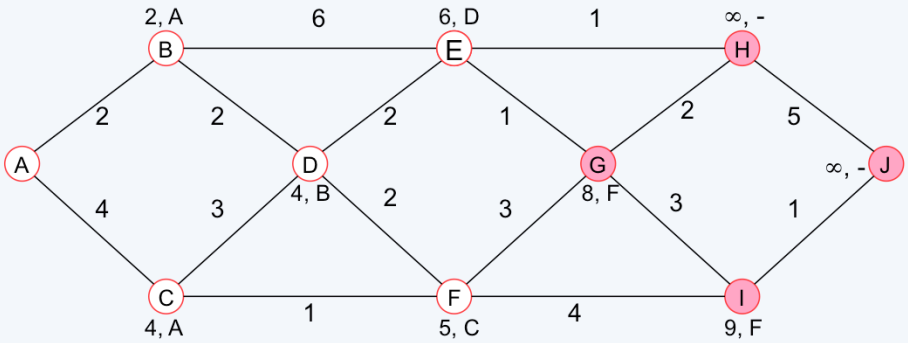


We continue with C ...

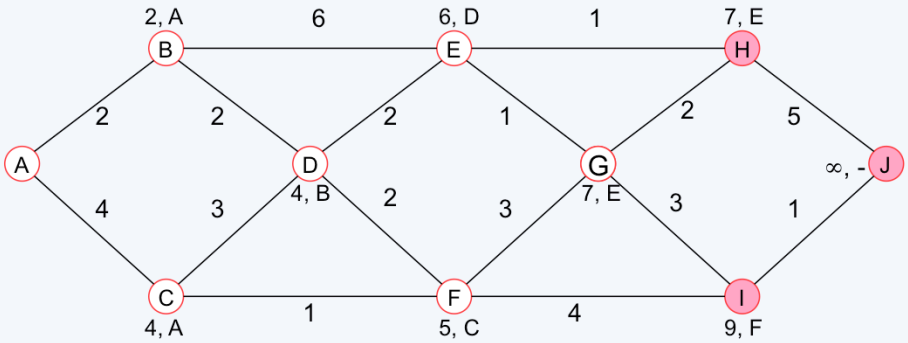
Shortest Path (5)



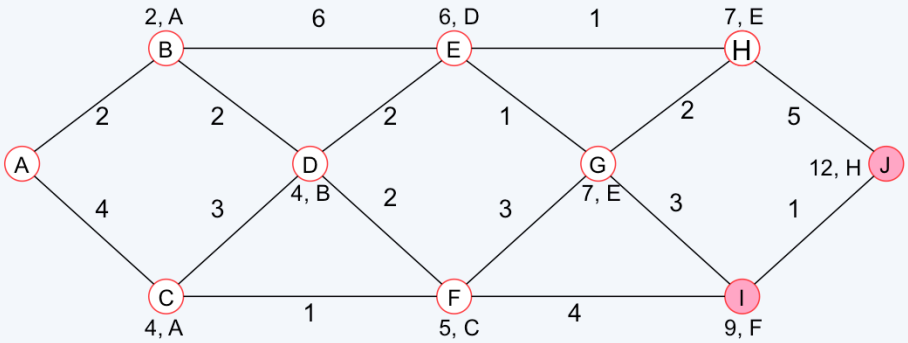
Shortest Path (6)

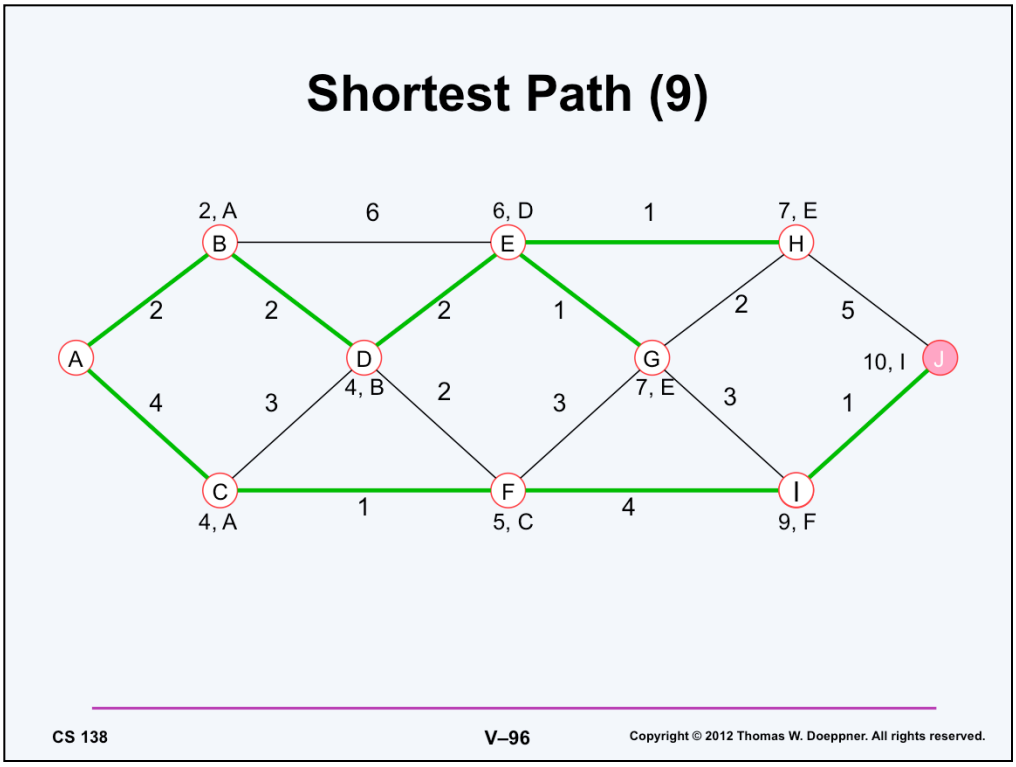


Shortest Path (7)

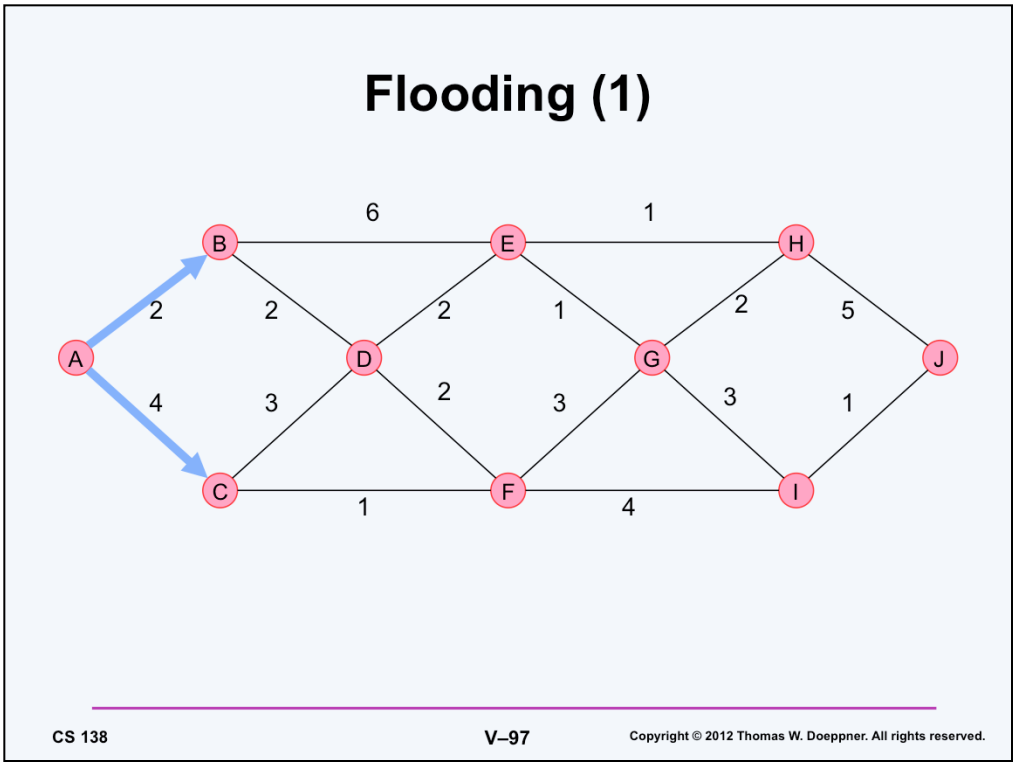


Shortest Path (8)

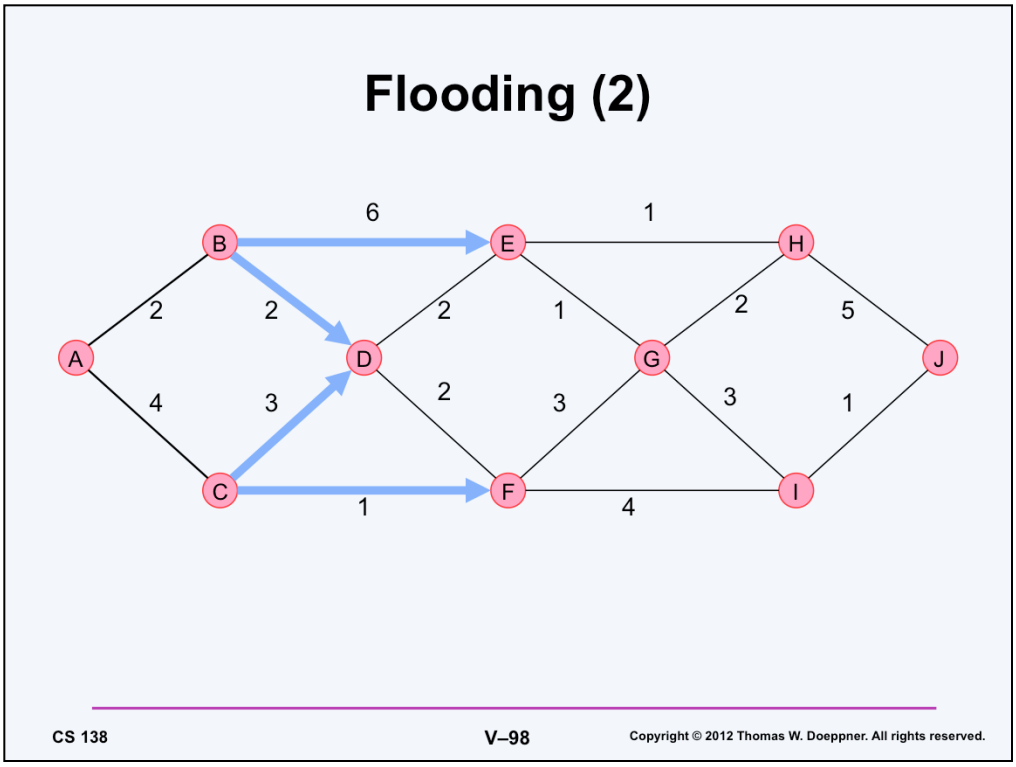




At this point we've done all but J, which we can mark as permanent as well. From the labels on each of the nodes, we can now find the shortest path to them from A.



So that each node can compute the shortest path to all others, we need a means for nodes to propagate their link-state information to the others. This information is not just sent to a node's neighbors, but *flooded* throughout an network: the source node sends a packet to each of its neighbors, which in turn send packets to each of their neighbors, etc.



We've got to make certain that flooded information doesn't circulate forever, but reaches each node no more than necessary. Thus a node doesn't forward flooded information on the link it came on. It also doesn't forward flooded information it's already seen. For example, node *D* above receives two copies, but forwards only the first.

Flooding Problems

- Reliability
  - new information must replace old
- Robustness
  - misbehaving routers don't bring down everyone

Flooding isn't as easy as it seems at first glance. This discussion is adapted from *Interconnections*, Second Edition, by Radia Perlman, Addison Wesley, 2000.

Reliability

- New information must replace old
  - use sequence numbers and acks
  - but …
    - sequence-number wraparound
    - restarting a router
      - state info is lost

General Link-State Flooding Strategy

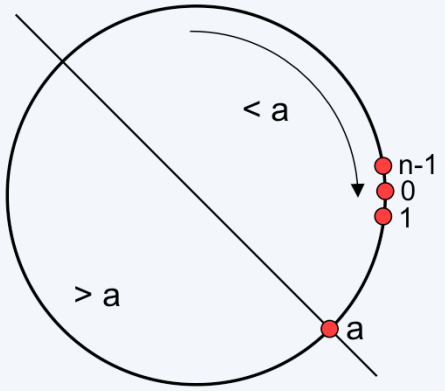
- Each router:
  - periodically and whenever there's a change:
    - flood internet with new link-state information
  - when receive link-state update:
    - if update is new, copy it into database
      - recompute routes

Approach 1

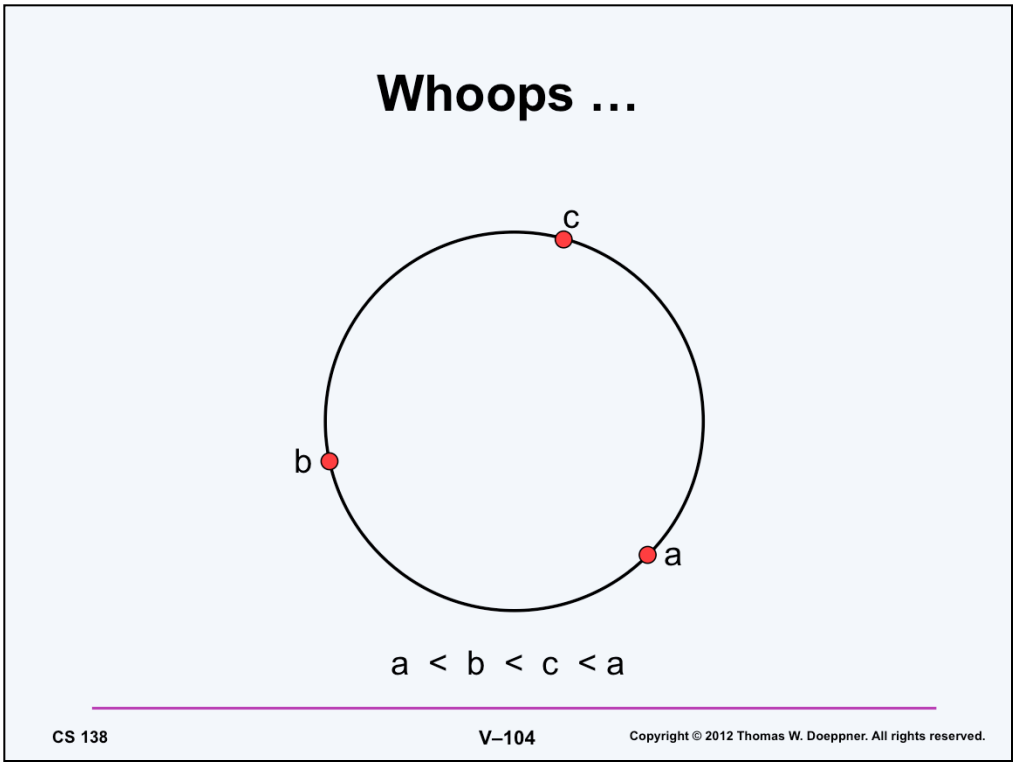
- Sequence numbers
  - each router assigns its next highest sequence number to its link-state updates (LSUs)
  - when receiving an LSU, ack it, but ignore its contents unless it has a higher sequence number than what's already received (in which case, forward it on)
- Restart
  - each router gives its LSUs an "age"
    - each router decrements the age of its copy of an LSU until it reaches zero or is replaced
  - if age is zero, then new LSU, regardless of sequence number, replaces old

The slide outlines an early approach used for flooding link-state updates on the ARPAnet. The "age" of an LSU is used to cope with the restart of a router: it starts with sequence numbers of zero. However, before it starts sending new LSUs, it waits long enough (90 seconds) so that all of the copies of its old LSUs on other routers have an age of zero. Thus all routers will accept the new LSUs.

Sequence-Number Wraparound



This slide (taken from Perlman, *op. cit.*) illustrates how comparison is done, taking wraparound into account. If the range of the sequence number is [0, n-1], then half of the range is greater than any particular value of a sequence number a and half is less than it. So, in particular, given any two sequence numbers a and b, if $a < b$ and $b-a < n/2$, or $a > b$ and $a-b > n/2$, then a is considered less than b.



An unfortunate result of our rules for sequence-number comparison is that we can get something like what's shown in the slide. Clearly, this "can't happen" in practice—correctly functioning routers won't produce consecutive LSUs with sequence numbers as shown in the slide. However, we have no guarantees that routers will always function correctly. One (malfunctioning) router on the ARPAnet actually did produce consecutive LSUs with sequence numbers as shown on the slide. The result was a complete collapse of the ARPAnet: a router holding an LSU with sequence number *a* received one with sequence number *b*, which it then forwarded (flooded to everyone). It then received an LSU with sequence number *c*, which, being more recent than the one with *b*, was again forwarded. At this point it received an LSU with sequence number *a*, resulting from its flood of the ARPAnet with that LSU. But now this appeared to be the most recent, so it forwarded (flooded) this one as well, ad infinitum. With all routers doing the same thing, all ARPAnet bandwidth was filled with these floods. (See Perlman, *op. cit.*, for more details.)

(Why didn't ageing take care of things? The LSUs didn't reside at each router long enough to be aged.)

Solution

- Sequence numbers not allowed to easily wraparound:
  - when max value is reached, no wraparound allowed until LSU "ages out"
  - thus after receiving LSUs with sequence numbers *a*, *b*, and *c*, by the time sequence numbers are allowed to wrap, a repeat occurrence of *a* will have "aged out"
- Sequence-number space made large enough so that wraparound occurs only on misbehaving routers

The solution is as sketched in the slide. In addition, further enhancements were made to use the bandwidth more efficiently.

Open Shortest Path First (OSPF)

- Link-state protocol
- Predecessors developed starting in late ’70s
- Implemented on Unix as *gated* (which also does RIP)
- In use since 1990
- Described in RFC 2328

OSPF (open shortest path first; the “open” means that it’s an open, as opposed to a closed or proprietary protocol) is in widespread use on the internet. It is a link-state protocol, but has a number of additional features that add to its usefulness.

The OSPF Approach

- All communication layered directly on top of IP
  - OSPF is a transport-layer protocol
- "Discover" neighbors
  - "hello" messages sent periodically on point-to-point links, multicast on LANs
- Measure or obtain cost of transmission to each neighbor
- Transmit routing information to all other routers
  - use reliable "flooding" algorithm
- Compute shortest path based upon current link-state information (use Dijkstra's algorithm)
  - modify Dijkstra's algorithm to obtain all shortest paths