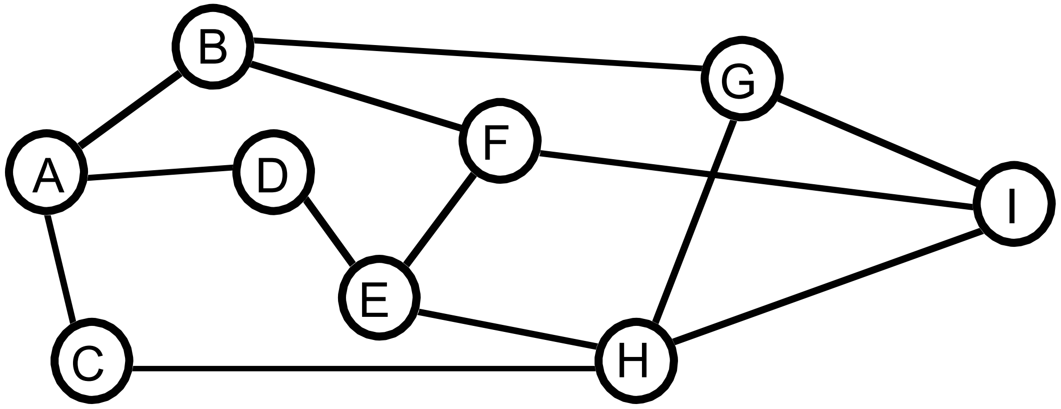
COMPSCI 314 S1 C

Wide Area Networks:
Routing Algorithms

Virtual Circuits

Wide Area Networks — Routing Tables



A Wide Area Network (WAN) is usually an arbitrary mesh of –

- The *stations*, or *nodes*, usually have several *connections* or *ports*, with each port connecting to exactly one other station (emphasising hardware) or node (emphasising graph theory aspects)
- The stations are connected by point-to-point *links*

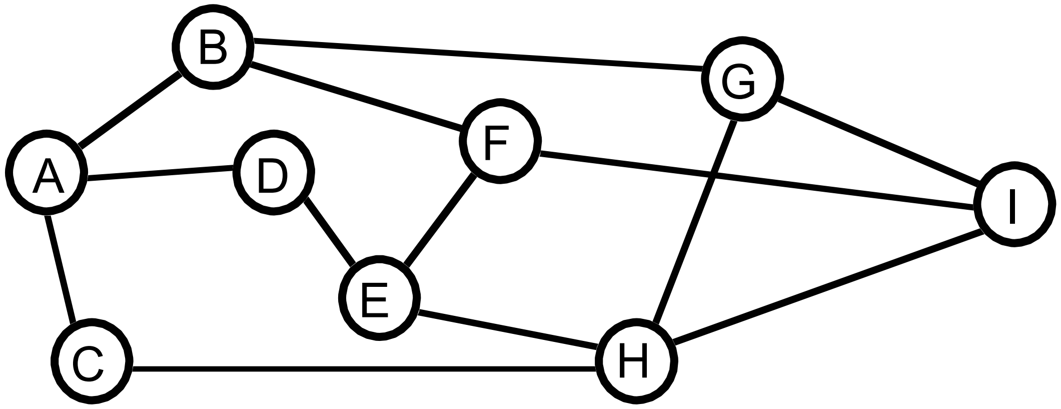
Wide Area Networks — Routing Tables

- A WAN node may be a complex of LANs (a university campus, etc.)
- A message arriving on one *input* port must usually be *forwarded*, *directed* or *routed* to some other *output* port on the same switch or router

Two types of routing —

1. Datagram routing has each packet independent and with the full destination address.
   Question —
   “Which is the best output port to use for this packet?”
2. Virtual Circuit routing assigns a temporary identifier (a ‘*virtual circuit*’) to each end-to-end circuit over each point to point link.
   Question —
   “If I receive VC = *x* over port *p*, what port and VC number should be assigned to forward it?”

Hop Counts



1. With datagram routing each station must know the cost of sending to each other station. The simplest cost is just the number of hops …

	A	B	C	D	E	F	G	H	I
A		1	1	1	2	2	2	2	3
B	1		2	2	2	1	1	2	2
C	1	2		2	3	3	2	1	2
D	1	2	2		1	2	3	2	3
E	2	2	3	1		1	2	1	2
F	2	1	3	2	1		2	2	1
G	2	1	2	3	2	2		1	1
H	2	2	1	2	1	2	1		1
I	3	2	2	3	2	1	1	1	

Dijkstra’s Algorithm

Dijkstra’s algorithm finds the shortest path from a *source* node to all other nodes. It partitions the nodes into two sets, those already *assigned* and those still *unassigned*. It is also called the *shortest path first* algorithm and is used for local IP routing.

It starts by determining the costs (or lengths) of all links between the nodes. Then, starting with only the source node in the assigned set, and all links inactive …

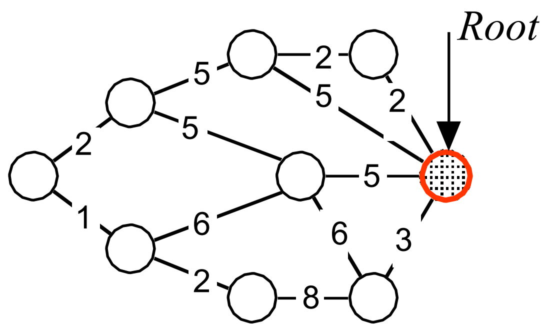
Dijkstra’s Algorithm (2)

1. Find the costs to the root node of all links which connect an unassigned node into the network.
2. Take the shortest or cheapest link, activate the link and add its node to the assigned set. (More than one link and node may be added if the link costs are equal.)
3. Repeat 1 and 2 until all the nodes have been added.

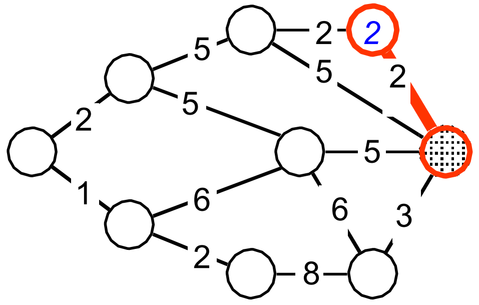
The diagram (next slide) shows the network with its link costs and shows how the paths develop as the algorithm proceeds.

An implementation of the algorithm will need tables of the link costs between nodes (some costs infinite) and continual searching of the costs between assigned and unassigned nodes; the graphical presentation is more obvious.

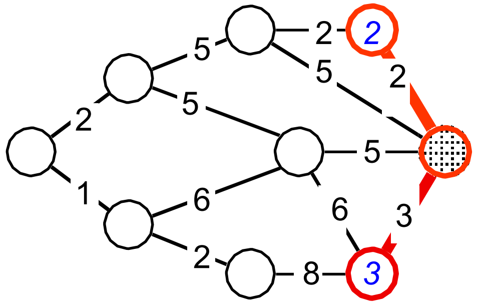
Dijkstra example: progressive evaluation



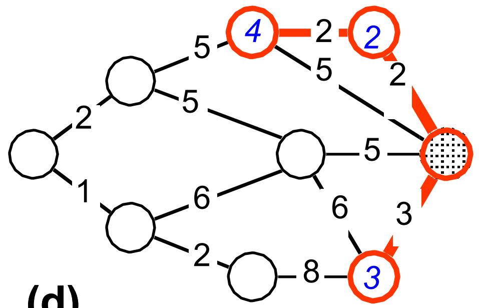
(a)



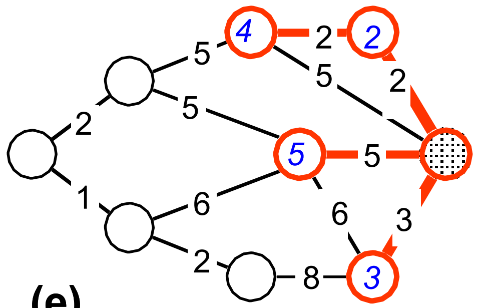
(b)



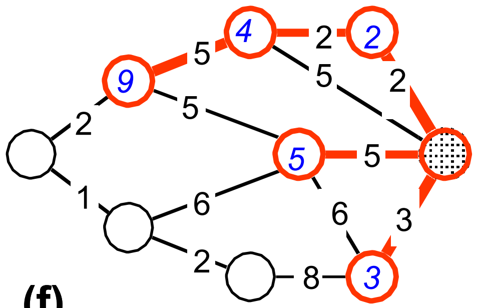
(c)



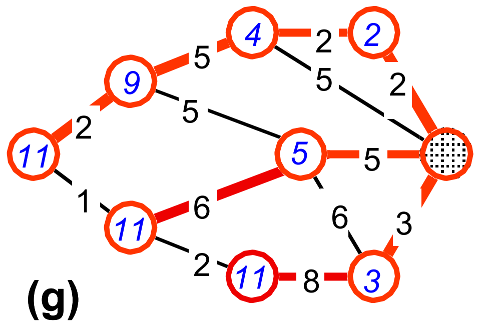
(d)



(e)



(f)

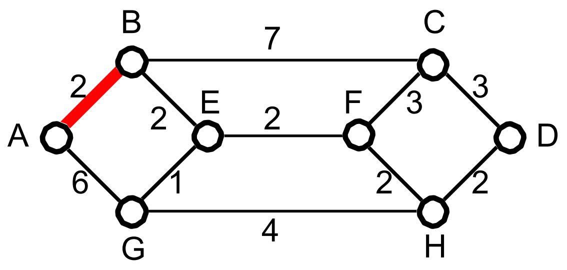


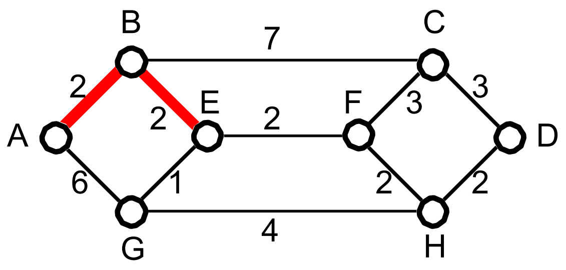
(g)

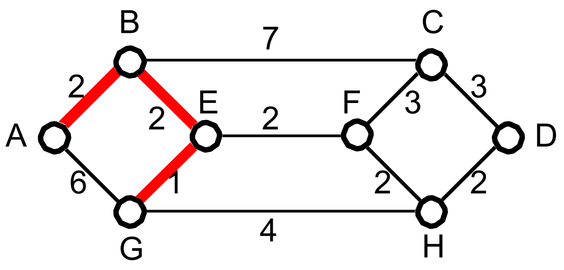
Step (g) brings in three nodes, all with a root cost of 11

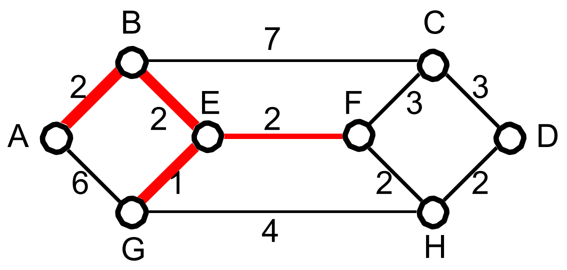
Dijkstra – second example

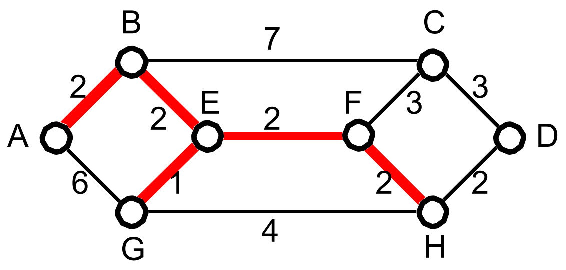
From 2002 exam; find route from A to D, with cost

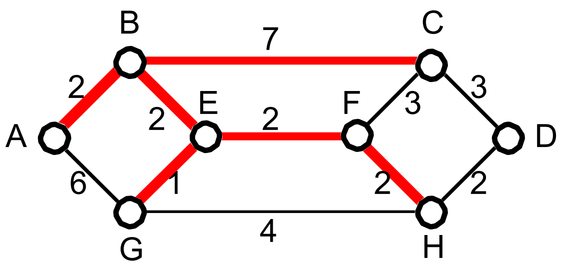


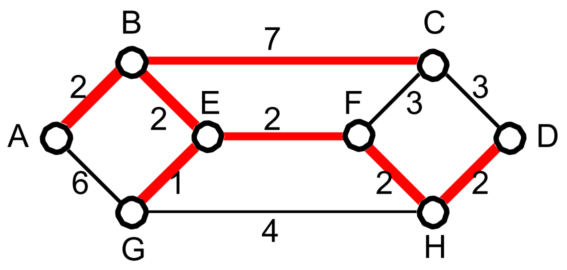












Comment: G and C are connected by 'off route' connections, or 'stubs.'

Note that costs *always* increase as nodes are connected

Bellman-Ford algorithm

The Bellman-Ford algorithm, also called the *distance-vector algorithm*, is a distributed algorithm which finds the shortest path to a destination from all other nodes. All costs are initially set to infinity.

- Each node keeps what it knows to be the shortest path from itself to the destination and informs its neighbours of this cost by sending the distance vector of {*dest*, *cost*} pairs
- A node i receives a distance vector from node j over a link with cost $d(i,j)$ and includes this in a table of all known costs to all destinations. If $L(i, k)$ is the cost from node i to the destination k, the cost from the current node i to the destination is the minimum over j of

$$L(i, k) = \min[L(i, k), d(i, j)+L(j, k)]$$

- If the new estimated cost is less than the current one, it is placed in the local table for transmission to neighbouring nodes

Bellman-Ford in action

(aka "Tux on the big OE")

Hong Kong onward destinations:

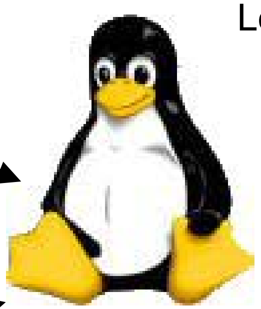
Destination	***Fare***	***via***
London	**$900**	**Singapore**
Singapore	$200	-
Beijing	$700	-
Tokyo	$400	-
...	...	...

Hong Kong: $600

Los Angeles: $900

Los Angeles onward destinations:

Destination	***Fare***	***via***
London	$400	-
Frankfurt	$500	-
Delhi	$1200	Frankfurt
Tokyo	$1000	-
...	...	...



Tux in Auckland

Sydney: $150

Destination	***Fare***	***via***
London	$1300	Singapore
Singapore	$700	-
Hong Kong	$700	-
Madrid	$1200	Dubai
...	...	...

Buenos Aires: $700

Buenos Aires onward destinations:

Destination	***Fare***	***via***
Helsinki	$1500	Frankfurt
Beijing	$1000	London
Madrid	$500	-
London	$800	-
...	...	...

Tux's routing table

Destination	***Fare***	***via***
London	$1300	Los Angeles
Singapore	$800	Hong Kong
Beijing	$1300	Hong Kong
Madrid	$1200	Buenos Aires
..	...	...

Melbourne: $180

Melbourne onward destinations:

Destination	***Fare***	***via***
London	$1350	Singapore
Frankfurt	$1300	Singapore
Delhi	$1000	Perth
Hong Kong	**$400**	**-**
...	...	...

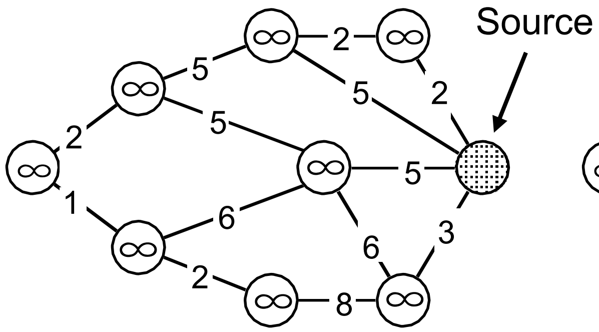
N.B.: Note that the Hong Kong and Melbourne tables may not be in synch!

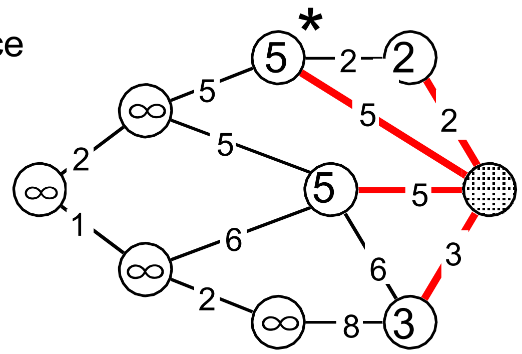
Bellman-Ford algorithm (2)

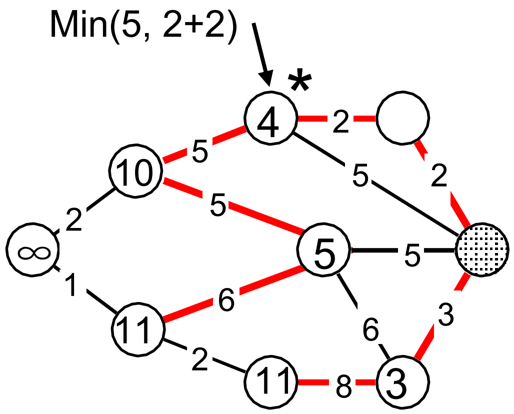
- The costs in the following diagram do not necessarily add up, because they travel out in ‘waves’ from the destination
- Less direct paths will arrive later, but may replace more direct paths if the new cost is less
- For example, the node marked with * receives a cost of 5 on the first message distribution
- At the next stage it receives a cost of (2+2=4) which replaces the original. Two other nodes also receive an initial estimate which is later reduced

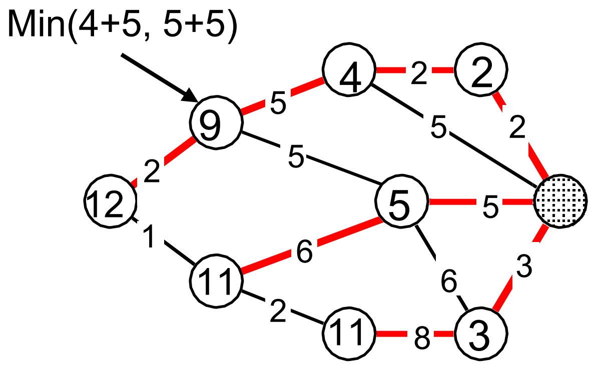
Unfortunately this diagram is no more than a crude approximation to the truth! You must handle Bellman-Ford by tables as in the text. It is tedious, but there is no alternative. The diagram here is a simple approximation for one node, without too much interaction between nodes. *See Shay, Chapter 7, pp. 459-466 (2nd edition)*

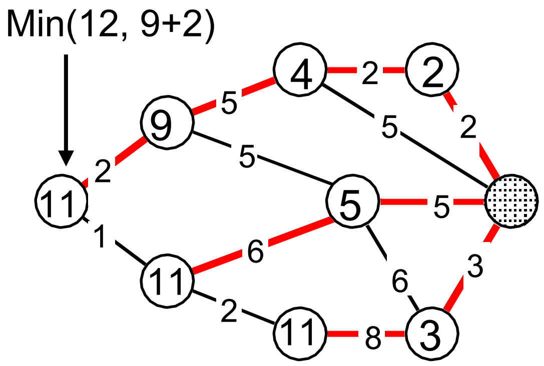
Bellman-Ford algorithm (3)





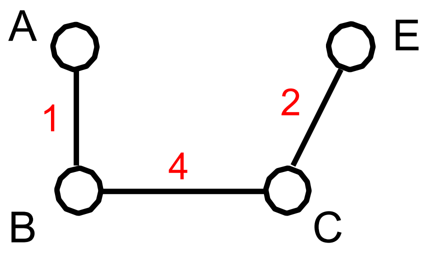






In reality the routing information spreads out simultaneously from all nodes, and then the different nodes start interacting in ways that just cannot be represented in a simple diagram.

Count-to-Infinity problem



Costs – C→E = 2
B→E = 6
A→E = 7

Consider the partial net above —

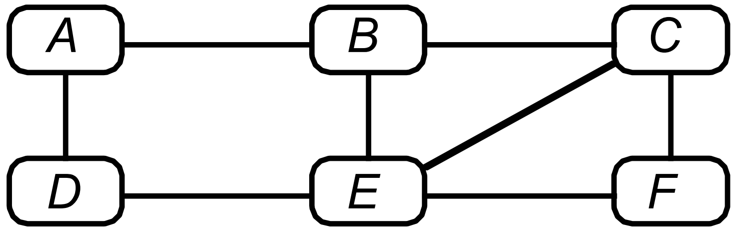
Then assume that link C → E fails (cost = ∞)

- C notes failure and sets C → E = ∞
- C passes this cost to B, which sets B → C → E = ∞
- but B hears from A of a route to E with cost = 7 and updates its best route to E to be 7+1 = 8
- now A hears of a route to E, via B, cost = 8 and sets its cost A→E = 9
- B hears from A and updates its cost B→E = 10
- loop continues as A→E and B→E both count upwards indefinitely
- Solve by setting an upper limit to link cost, at which cost = ∞

Bellman-Ford worked example

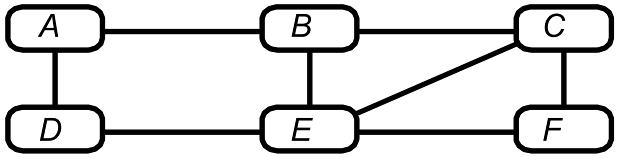
- In this network all costs are 1, making it a hop-count metric
- Consequently there is little change of route costs as better routes are found; the closest routes are the best ones
- If we had a few expensive links, we would find that some initial routes would be replaced by cheaper ones with more hops

All path costs are 1 (i.e. use hop-count metric)



Bellman-Ford worked example (2)

All path costs are 1 (i.e. use hop-count metric)



Iteration 1 Destination

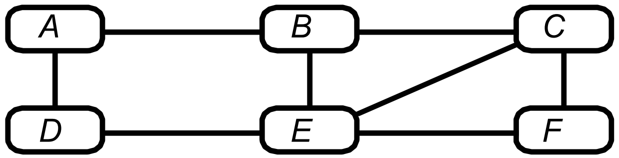
Source	A	B	C	D	E	F
A	—	[B,1]	?	[D,1]	?	?
B	[A,1]	—	[C,1]	?	[E,1]	?
C	?	[B,1]	—	?	[E,1]	[F,1]
D	[A,1]	?	?	—	[E,1]	?
E	?	[B,1]	[C,1]	[D,1]	—	[F,1]
F	?	?	[C,1]	?	[E,1]	—

Iteration 2 Destination

Source	A	B	C	D	E	F
A	—	[B,1]	[B,2]	[D,1]	[B,2]	?
B	[A,1]	—	[C,1]	[A,2]	[E,1]	[C,2]
C	[B,2]	[B,1]	—	[E,2]	[E,1]	[F,1]
D	[A,1]	[A,2]	[E,2]	—	[E,1]	[E,2]
E	[B,2]	[B,1]	[C,1]	[D,1]	—	[F,1]
F	?	[C,2]	[C,1]	[E,2]	[E,1]	—

Bellman-Ford worked example (3)

All path costs are 1 (i.e. use hop-count metric)



Iteration 3

Source	Destination A	B	C	D	E	F
A	—	[B,1]	[B,2]	[D,1]	[B,2]	[B,3]
B	[A,1]	—	[C,1]	[A,2]	[E,1]	[C,2]
C	[B,2]	[B,1]	—	[E,2]	[E,1]	[F,1]
D	[A,1]	[A,2]	[E,2]	—	[E,1]	[E,2]
E	[B,2]	[B,1]	[C,1]	[D,1]	—	[F,1]
F	[C,3]	[C,2]	[C,1]	[E,2]	[E,1]	—

Iteration 4
no change

Source	Destination A	B	C	D	E	F
A	—	[B,1]	[B,2]	[D,1]	[B,2]	[B,3]
B	[A,1]	—	[C,1]	[A,2]	[E,1]	[C,2]
C	[B,2]	[B,1]	—	[E,2]	[E,1]	[F,1]
D	[A,1]	[A,2]	[E,2]	—	[E,1]	[E,2]
E	[B,2]	[B,1]	[C,1]	[D,1]	—	[F,1]
F	[C,3]	[C,2]	[C,1]	[E,2]	[E,1]	—

Problems of Bellman-Ford routing

- Count to infinity, as above
- In a distributed algorithm, costs are updated as the routes are evaluated
- Routes may change during the calculation
- A low-cost route may suddenly become a preferred path and be overwhelmed as all traffic is directed to it; suddenly it is a very high cost route
- Traffic may oscillate between routes
- The routing process may become completely unstable
- In any case “bad news travels slowly” because congestion information travels out by only one hop per routing iteration. (If we use choke packets, etc., the congestion information could travel much faster)

Link State Routing

- When a node is initialised, it determines the link costs for all of its *interfaces* (connected links)
- Each node sends all of its costs to its neighbours, including all other costs which it knows
- Whenever any link costs change, or links become (unusable), the nodes connected to it advertise its new costs. We say that nodes use *reliable flooding* to propagate their link state
- Eventually each node knows all costs in the network and can execute a local algorithm, such as Dijkstra’s, so as to construct its own routing table

Hierarchical Routing

- Simple routing algorithms get overwhelmed by large networks
- Split networks into groups of nodes called *domains*
- Routing within domains is done per-node, using a standard protocol
- Each domain has one or more *border routers*, to connect to routers in other domains
- The border routers are effectively a network of routers
- There may be several levels of the hierarchy

We will revisit routing algorithms after looking at the IP protocol …

Circuit Types

- A *permanent circuit* is a hardwired connection between two end points. It is permanently assigned to that service, usually uses dedicated cables, and is unavailable to anybody else. Example is a private intercom.
- A *switched circuit* is *established* on demand by a *connection request*. It then resembles a permanent circuit and is dedicated to the user until a *disconnection request* asks for the connection to be *broken*.
  Example is a traditional telephone. The component connections are dedicated for the life of the connection, but then released for other users when the connection is broken.
- A *virtual circuit* has no links dedicated to any user. Users establish a virtual connection which looks like a switched circuit to the user. They send packets or messages which are directed through the network by *routing tables*, set up by the *connection request*.

Virtual Circuits

A *virtual circuit* appears to the user as equivalent to a dedicated point-point service but is maintained by computers. It will usually transport data at a guaranteed rate (bit/s) and with guaranteed reliability and error rate.

- Internally information is carried by many small packets, each with a short *virtual circuit number* which identifies its virtual circuit over that physical link
- The virtual circuit number changes as the packet is switched by a switch or router from one incoming link to an outgoing link, according to information held in *routing tables*
- It is the combination of entries in the routing tables of successive switches which defines the end-to-end virtual circuits

DO NOT confuse Virtual Circuits with Virtual LANs!

Virtual Circuit Routing

We discuss several terms before describing Virtual Circuit routing:

1. A *switch* and *router* are for our purposes very similar devices. Each router normally connects to several other routers and can divert messages coming from one link to any other link. A *switch* tends to examine less of a message (such as addresses in a bridge), while a *router* looks at protocols as well.

2. A *link* is a long distance connection between two routers, usually at least several kilometres long (perhaps many thousands of kilometres). It provides a point-to-point connection and may serve thousands of user circuits. (Remember that routing is not necessary in a fully-connected LAN because all stations can see all traffic.)

Virtual Circuit Routing (2)

3. A *circuit* is what the user sees as a connection to another user.

- The circuit is first *established* by a *call establishment* request (or *call set-up* request) which contains the full end-address of the called user (IP or similar, equivalent to a telephone number).
- This sets up a suitable set of entries in the switch *VC routing tables*. After establishing the circuit, the user may just send data over it according to any suitable protocol.
- Finally a *call disconnect* will break or tear down the connection by clearing its routing table entries.

Alternatively, if you are emphasising the hardware level, a circuit may be equivalent to a link as described above.

Virtual Circuit Routing (3)

4. A *port* is a physical connection by which a link connects into a switch or router:
   - It usually means a connector for the cable (the physical aspect of the link) and associated interfaces to encode and decode data and transfer it to and from the router memory and processing.
   - This is quite different from a *port* between protocol layers, such as from IP into TCP and other services.

5. A virtual circuit number (VC, or Virtual Circuit ID) is the number within the packet which identifies the virtual circuit:
   - It usually changes as the packet goes from router to router over the network.
   - Virtual Circuit numbers may be shared between different links to the same router, and even in different directions over the same link, but must be unique in one direction over the one link.

Virtual Circuit Number Generation

Virtual circuit numbers are usually quite arbitrary (except that some very small ones are often reserved for system work and communication between adjacent routers). Many systems have rules for allocating numbers.

- Here, the station or router making or forwarding the call request generates both VC numbers, using a small value for the ‘outward’ circuit and large ones for the ‘inward’ circuit.
- Sometimes the router receiving the request might generate both, or it might receive the outgoing number and reply with the incoming number.
- What must be avoided is the situation where a left-to-right call request and a right-to-left call request select the same number for say left-to right signalling on the same link.
- This is handled by using the different blocks of numbers.

Routing tables

There are two types of routing tables:

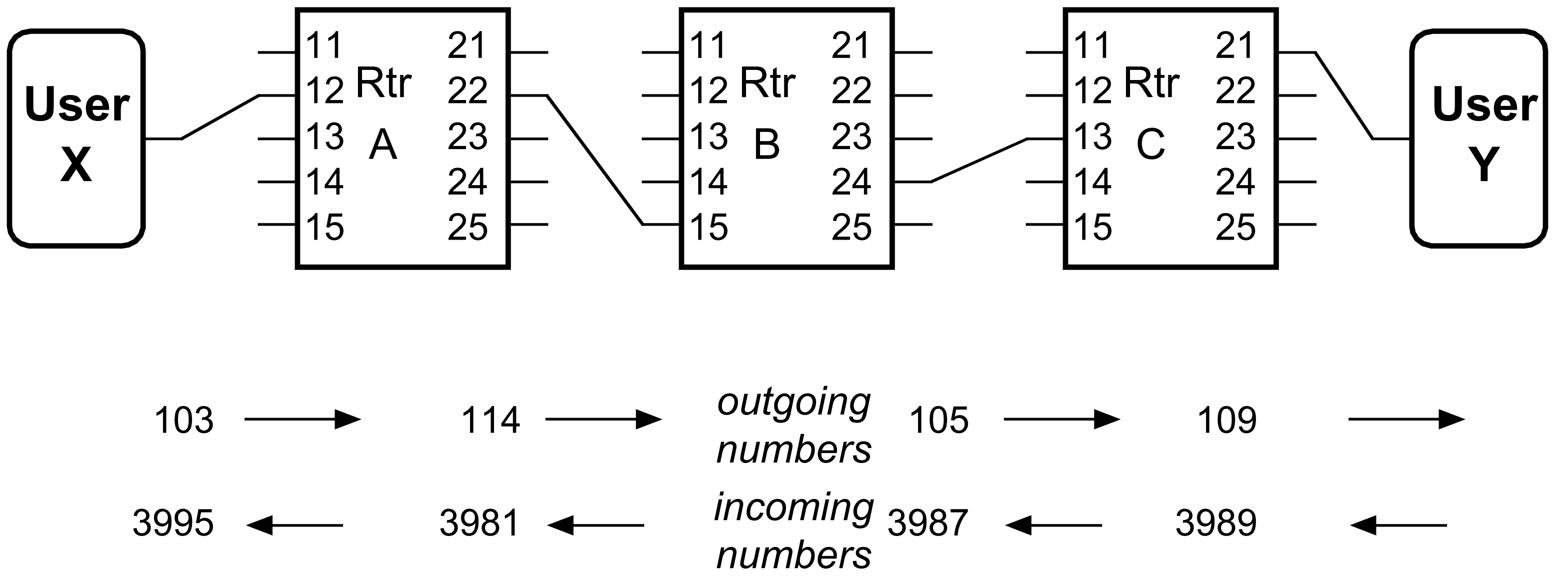
1. End-to-end routing tables are used during call establishment (and are identical to the routing tables used all the time for datagrams)
   - They are interrogated by the final address of the called user and give the best route toward that user, usually just specifying the output port to be used
2. Virtual Circuit routing tables are set up from the route discovered by call establishment
   - For each packet, they map from

     `{input_port, input_VC}` to `{output_port, output_VC}`

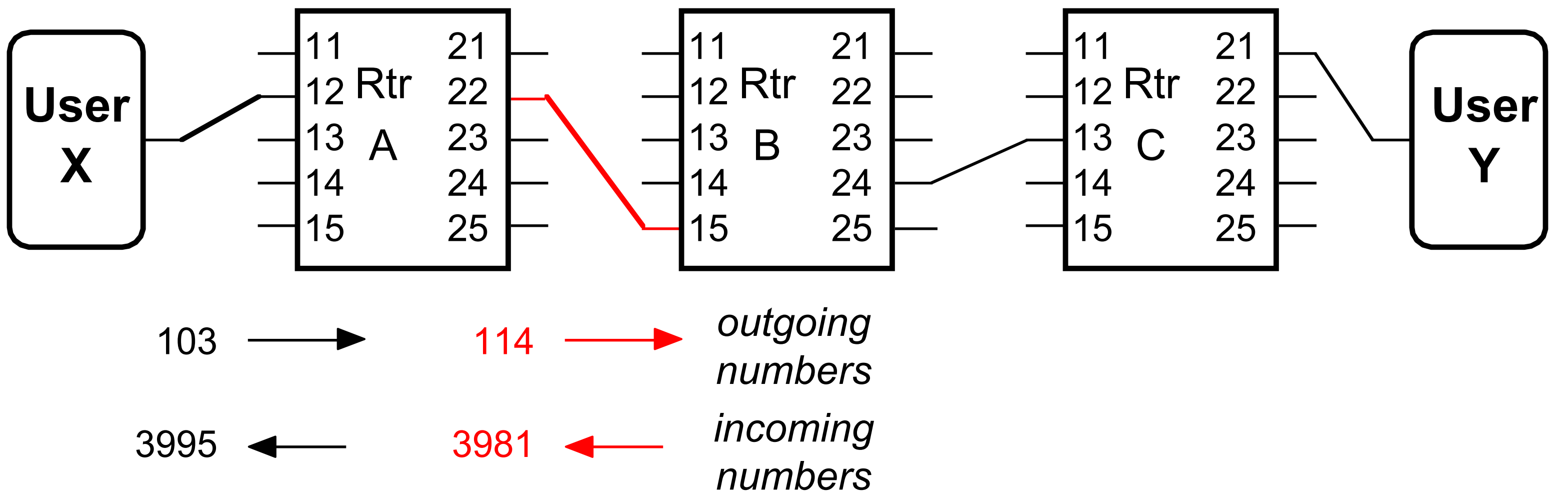
Setting up a virtual circuit

For this assume that User X wishes to connect User Y.

X has a connection to router A (via port 12, but this is irrelevant to X).

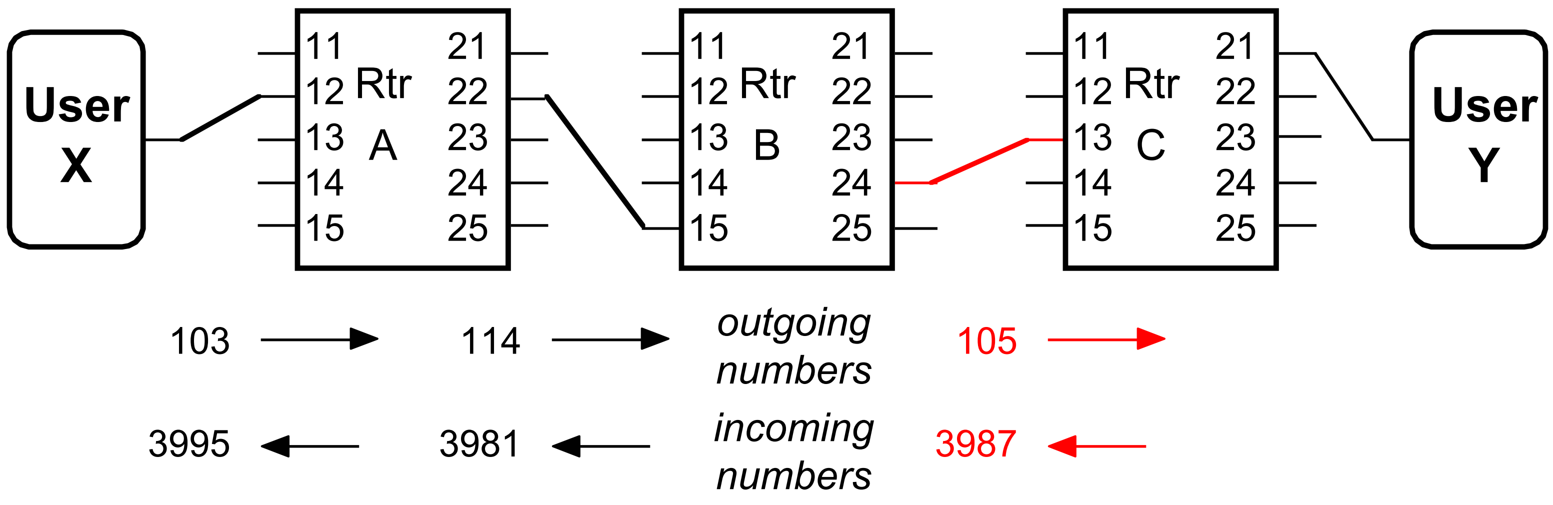


Setting up a virtual circuit (3)



- Router A finds that the best route to User Y is through port 22, forwards the call request over that port and sets up its VC routing table as below. Any packet with VC=103 on port 12 is given the VC=114 and sent over port 22. Similarly anything received with VC=3981 on port 22 is sent out on port 12 with VC=3995.

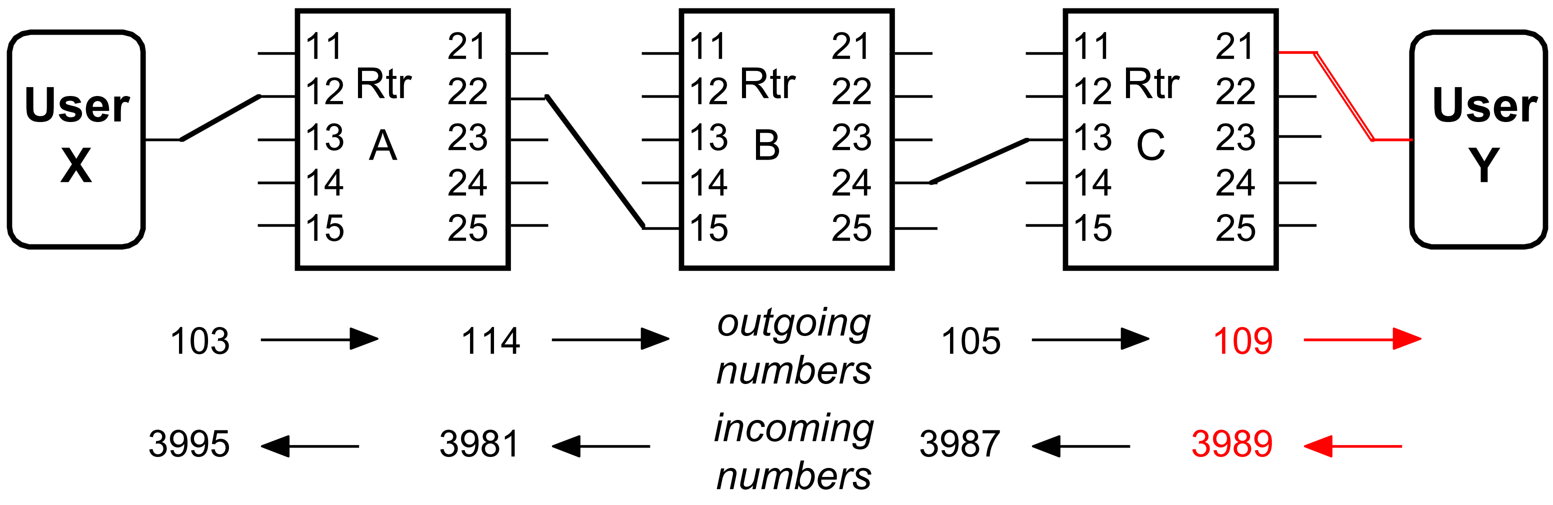
Setting up a virtual circuit (4)



User X
11 12 13 14 15 Rtr A 21 22 23 24 25
11 12 13 14 15 Rtr B 21 22 23 24 25
11 12 13 14 15 Rtr C 21 22 23 24 25
User Y
103
114
outgoing numbers
105
3995
3981
incoming numbers
3987

- Router B receives the call set-up request with VC=114 on port 15 (because this is its link to router A) and finds from the end-routing tables that the connection should be over port 24, to which it allocates VC=105. The randomly chosen VC number for the reverse circuit is 3987.

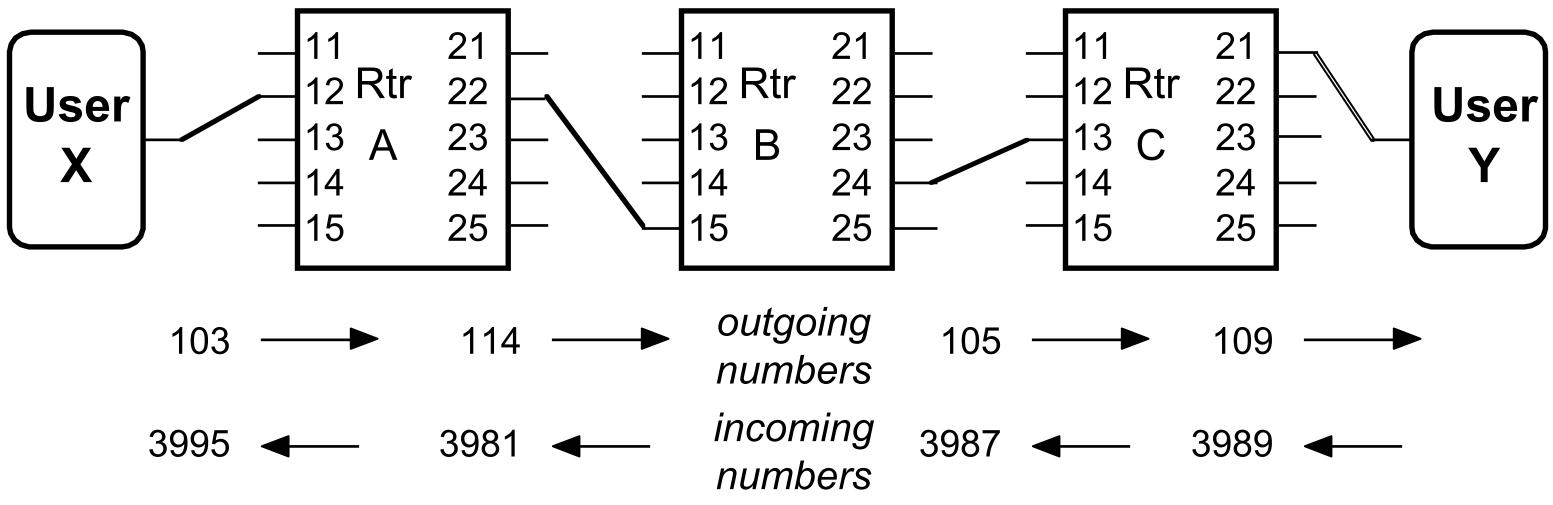
Setting up a virtual circuit (4)



User X
11 12 13 14 15 Rtr A 21 22 23 24 25
11 12 13 14 15 Rtr B 21 22 23 24 25
11 12 13 14 15 Rtr C 21 22 23 24 25
User Y
103
114
outgoing numbers
105
109
3995
3981
incoming numbers
3987
3989

- Router C receives the request with VC=105 on port 13 and finds that it should connect to port 21, for which it chooses a circuit of 109, with 3989 in the reverse direction.

Setting up a virtual circuit (5)


User X
11 12 13 14 15 Rtr A 21 22 23 24 25
11 12 13 14 15 Rtr B 21 22 23 24 25
11 12 13 14 15 Rtr C 21 22 23 24 25
User Y
103
114
outgoing numbers
105
109
3995
3981
incoming numbers
3987
3989

- User Y receives the request and accepts it, noting that for this circuit it will receive with VC=109 and send with VC=3989.

Setting up a virtual circuit (6)

The final routing tables are what are needed for message transmission, but the table also shows a ‘reverse VC’ entry. This is seldom shown in texts but is needed if the router must reply to a message. For example if Router B wants to reply to VC=114 on port=15, it will reply with VC=3981 (also of course on port 15).

	In_Port	In_VC	Out_Port	Out_VC	*ReverseVC*
Router A	12	103	22	114	3995
	22	3981	12	3995	114
Router B	15	114	24	105	3981
	24	3997	15	3981	105
Router C	12	105	21	109	3997
	22	3989	13	3997	109

Virtual Circuits: Summary

- Virtual Circuit:
  - Appears to the user as a point-to-point circuit
- VC numbers
  - Must be unique within each link of the circuit
- Routing tables
  - End-to-end: very similar to those for *datagram routing*
  - VC routing: set up for each switch during call setup
- Networks using VCs:
  - ATM
  - Frame Relay

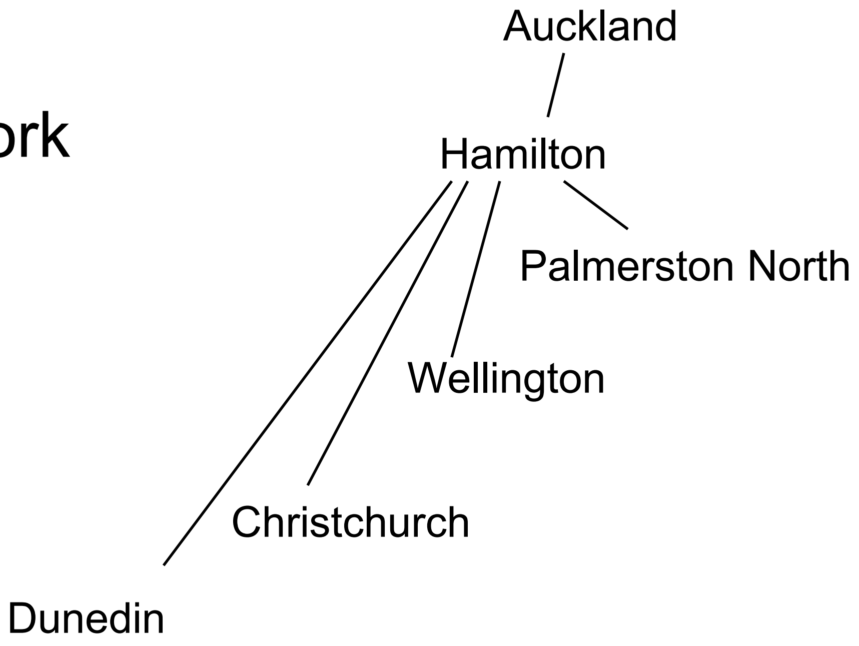
Frame Relay

- A simplified version of X.25
  [Shay (2nd edtion) section 7.3]
- Provides links to users as VCs,
  with specified link capacities and QoS behaviour

Kawaihiko network

(1989 – 1998)

All links 48 kb/s



Congestion and Deadlock

- Congestion occurs when a node (or switch, or router, …) cannot forward all the traffic which it receives, usually because the output links are overwhelmed
- Congestion does not occur just because the output link is fully loaded. It really happens because the node *buffers* are *overloaded* and cannot accept more traffic
- The switch’s buffers fill with unsent traffic — eventually its own buffers reject traffic and congestion spreads
- *Deadlock* is an extreme case of congestion, where every node is waiting for another node to do something

Handling Congestion

In general, control congestion by reducing the traffic into the network, or into the congested area.

If congestion does occur, the main methods of congestion control are —

- Packet discard. Just throw away some packets, perhaps selected at random. This is simple and may be quite effective. A reasonable protocol can recover from lost packets – TCP assumes that all packet is loss is from discards. It is a preferred method in ATM (Asynchronous Transfer Mode) networks, where some 'cells' can be marked as 'discardable'.

Handling Congestion (2)

- *Congestion notification*. When the packet passes through a congested node, that node sets a *congestion indication* bit in the packet header, to inform the receiver that there is congestion.
  The receiver must then ask the sender to reduce traffic.

- *Buffer allocation*. Congestion occurs because a node exhausts its buffers. If we make sure that all the nodes along the path have enough buffers to hold all unacknowledged traffic, we will never get congestion.
  This requires Virtual Circuits, operating with a window end-to-end protocol, so we know exactly how many buffers are needed at each node. It may be extravagant if circuits seldom need all their buffers.

Handling Congestion (3)

- *Choke Packets*. When a node detects approaching congestion, it sends back a choke packet, asking the sender to reduce its input traffic. Eventually the sending node will try increasing the traffic to the previous value, hoping that congestion will have cleared.

- *Explicit Congestion Notification (ECN)*. When the packet passes through a congested node, that node sets a *congestion indication* bit in the packet header to inform the receiver that there is congestion.
  The receiver must then ask the sender to reduce traffic.

Congestion Collapse

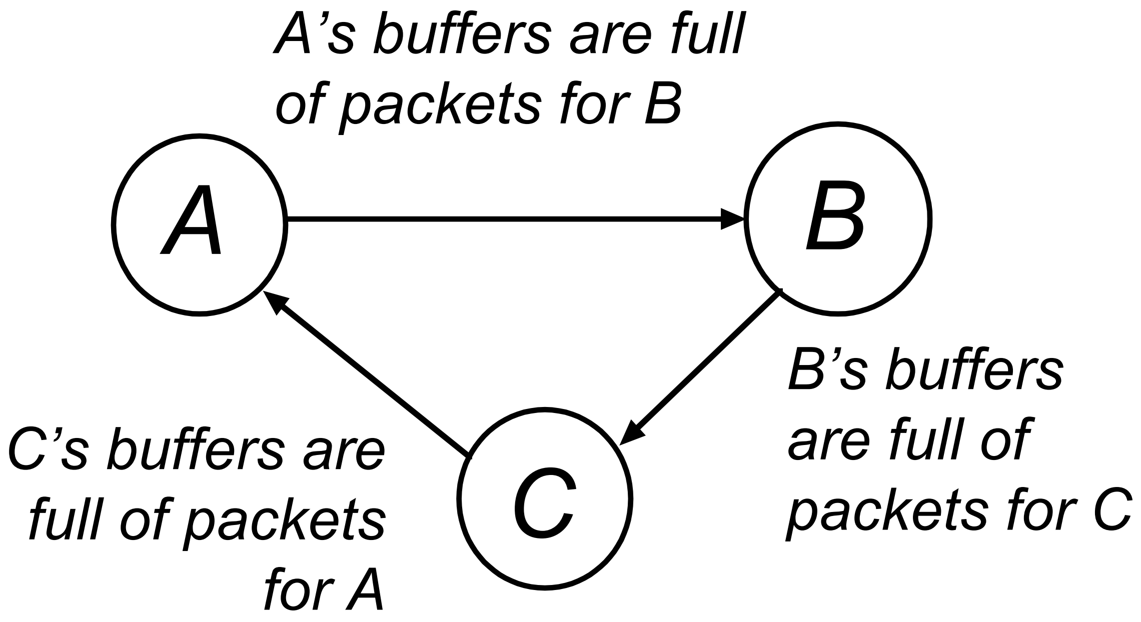
- *Congestion collapse* can occur with a badly-designed protocol, especially if congestion is handled by discarding packets. (We met it earlier in the simple Aloha protocol.)

  It can also occur if congestion causes packet delays to be so large that the sender times-out and retransmits. The network then contains the original data (which caused the original congestion) *plus* the retried traffic (which will surely cause even more congestion).

In general, avoid congestion collapse by sending data at a lower rate if congestion is suspected, such as increasing the time between retransmissions. Ethernet/IEEE802.3 uses binary exponential backoff for just this reason.

Deadlock (or deadly embrace, or lock-up)

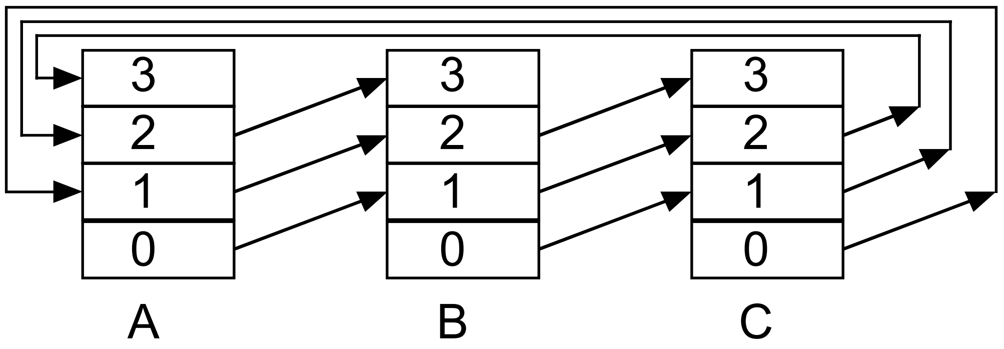
- *Store and forward deadlock*
  Each node's buffers are full of packets for some node which it cannot reach directly. Usually a node detecting store and forward deadlock will discard a randomly selected packet.


A's buffers are full of packets for B
A
B
B's buffers are full of packets for C
C
C's buffers are full of packets for A

Deadlock (2)

Can solve with either

- Use a hierarchy of buffers, according to number of hops — a group for 1 hop traffic, a group for 2 hops and so on. A packet waits if there is no room in its member of the hierarchy. No packets wait for a 'lower' buffer and the circular wait must be broken at some stage.

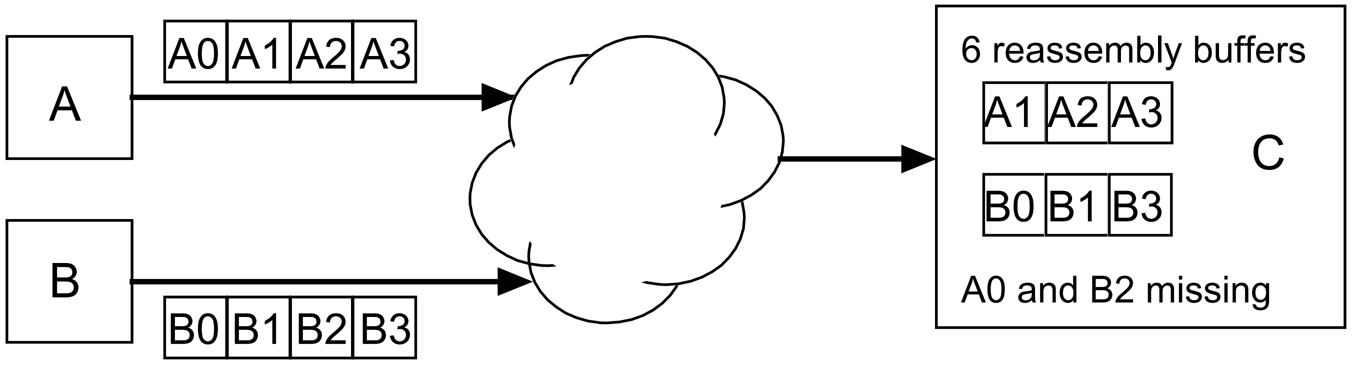


In a possible deadlock loop, messages from A are in a different cycle of buffers from B and C's messages, so they can never interfere.

- Reserve a few 'overflow' buffers, used only if a deadlock is detected. Then move a random packet into the overflow to release space for a packet which may be forwarded.

Deadlock (3)

- *Reassembly deadlock*



- Two stations, A and B, send messages to a third station C. Each message is divided into 4 packets, but C cannot hold complete packet windows from both A and B
- All of C’s buffers are full, but some packets are missing or late
- C cannot accept the packets needed to complete a message and allow buffers to clear — we have deadlock
- Can discard buffers in C until one message can reassemble
- Nodes A and B should negotiate space in C to allow complete reassembly