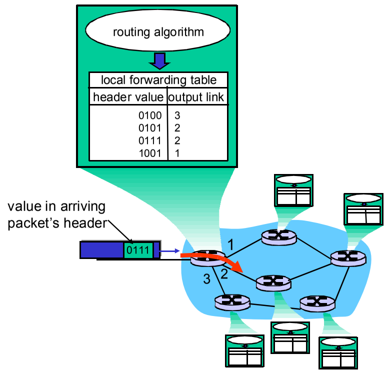
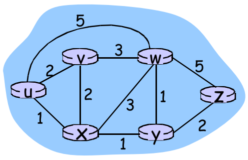
Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

Interplay between routing, forwarding



Graph abstraction



Graph: G = (N,E)

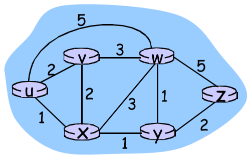
N = set of routers (**n**odes) = { u, v, w, x, y, z }

E = set of links (**e**dges) ={ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

Remark: Graph abstraction is useful in other network contexts

Example: P2P, where N is set of peers and E is set of TCP connections

Graph abstraction: costs



- $c(x,x')$ = cost of link (x,x')
  - e.g., $c(w,z) = 5$
- **cost** could always be 1, or inversely related to bandwidth, or inversely related to congestion

Cost of path $(x_1, x_2, x_3,\ldots, x_p) = c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$

Question: What's the least-cost path between u and z ?

Routing algorithm: algorithm that finds **least-cost path**

Routing Algorithm classification

Global or local (decentralized) information?

Global:
- all routers have complete topology, link cost info
- "link state" algorithms

Local/Decentralized:
- router knows physically-connected neighbors, link costs to neighbors
- iterative process of computation, exchange of info with neighbors
- "distance vector" algorithms

Static or dynamic?

Static:
- routes change slowly over time

Dynamic:
- routes change more quickly
  - periodic update
  - in response to link cost changes

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

A Link-State (Global) Routing Algorithm

Dijkstra's algorithm
- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ('source") to all other nodes
  - gives forwarding table for that node
- iterative: after k iterations, know least cost path to k dest.'s

Notation:
- $c(x,y)$: link cost from node x to y; $= \infty$ if not direct neighbors
- $D(v)$: current value of cost of path from source to dest. v
- $p(v)$: predecessor node along path from source to v
- N': set of nodes whose least cost path definitively known

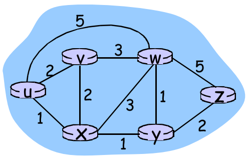
Dijsktra's Algorithm: greedy

```
Initialization:
  N' = {u} // starting from source node
  for all nodes v
    if v adjacent to u
       then D(v) = c(u,v) // neighbor connected
    else D(v) = ∞ // not connected

Loop
  find w not in N' such that D(w) is a minimum // greedy!
  add w to N'
  update D(v) for all v adjacent to w and not in N' :
     D(v) = min( D(v), D(w) + c(w,v) )
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes in N'
```

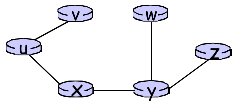
Dijkstra's algorithm: example

Step	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					



Dijkstra's algorithm: example (2)

Resulting shortest-path tree from u:



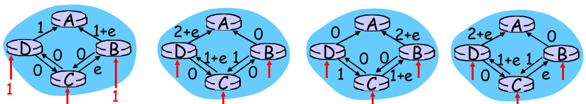
Resulting forwarding table in u:

destination	link output port
v	(u,v)
x	(u,x)
y	(u,x)
w	(u,x)
z	(u,x)

Dijkstra's algorithm, discussion

Algorithm complexity: n nodes
- each iteration: need to check all nodes, w, not in N
- n(n+1)/2 comparisons: $O(n^2)$
- more efficient implementations possible (using a heap): O(nlogn)

Oscillations possible (dynamically changing cost):
- e.g., link cost = amount of carried traffic
- **Link costs are not symmetric**



initially … recompute routing … recompute … recompute

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

Distance Vector Algorithm

Bellman-Ford Equation (dynamic programming)

Define

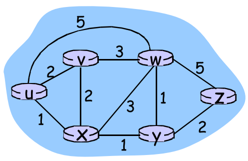
$d_x(y)$:= cost of least-cost path from x to y

Then

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

where min is taken over all neighbors v of x

Bellman-Ford example



B-F equation says:

$d_u(z) = \min \{ c(u,v) + d_v(z),$
$c(u,x) + d_x(z),$
$c(u,w) + d_w(z) \}$
$= \min \{2 + 5,$
$1 + 3,$
$5 + 3\} = 4$

Clearly, $d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$

Node that achieves minimum is next hop in shortest path → forwarding table

Distance Vector Algorithm

- $D_x(y)$ = estimate of least cost from x to y
- Node x knows cost to each neighbor v: $c(x,v)$
- Node x maintains distance vector $\mathbf{D}_x = [D_x(y): y \in N]$
- Node x also maintains its neighbors' distance vectors
  - For each neighbor v, x maintains $\mathbf{D}_v = [D_v(y): y \in N]$

Distance vector algorithm (4)

Basic idea:
- From time-to-time, each node sends its own distance vector estimate to neighbors
- Asynchronous **异步的**
- When a node x receives new DV estimate from neighbor, it updates its own DV using B-F equation:

  $D_x(y) \leftarrow \min_v\{c(x,v) + D_v(y)\}$ for each node $y \in N$

- Under minor, natural conditions, the estimate *$D_x(y)$* ***converge to*** *the actual least cost* $d_x(y)$
  - **最终会逼近实际距离**

Distance Vector Algorithm (5)

Iterative, asynchronous: each local iteration caused by:
- local link cost change
- DV update message from neighbor

Distributed:
- 邻居信息，无需全局信息
- each node notifies neighbors *only* when its DV changes
  - neighbors then notify their neighbors if necessary

Each node:

wait for (change in local link cost or msg from neighbor)

recompute estimates

if DV to any dest has changed, *notify* neighbors

$D_x(y) = \min\{c(x,y) + D_y(y), c(x,z) + D_z(y)\}$
$= \min\{2+0 , 7+1\} = 2$

$D_x(z) = \min\{c(x,y) + D_y(z), c(x,z) + D_z(z)\}$
$= \min\{2+1 , 7+0\} = 3$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

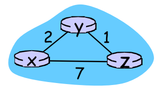
node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

time



$D_x(y) = \min\{c(x,y) + D_y(y), c(x,z) + D_z(y)\}$
$= \min\{2+0 , 7+1\} = 2$

$D_x(z) = \min\{c(x,y) + D_y(z), c(x,z) + D_z(z)\}$
$= \min\{2+1 , 7+0\} = 3$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	7	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

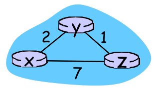
node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	3	1	0

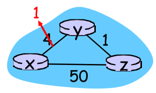
from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

time



Distance Vector: link cost changes

Link cost changes:
- node detects local link cost change
- updates routing info, recalculates distance vector
- if DV changes, notify neighbors



"good news travels fast"

At time t_0, y detects the link-cost change, updates its DV, and informs its neighbors.

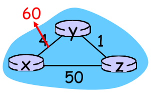
At time t_1, z receives the update from y and updates its table. It computes a new least cost to x and sends its neighbors its DV.

At time t_2, y receives z's update and updates its distance table. y's least costs do not change and hence y does *not* send any message to z.

Distance Vector: link cost changes

Link cost changes:
- good news travels fast
- **bad news travels slow - "count to infinity" problem!**
- 44 iterations before algorithm stabilizes: see text

Poisoned reverse:
- If Z routes through Y to get to X :
  - Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z)
- will this completely solve count to infinity problem?



4→60, but z does not know;
z→x is still 5
Then, Y→x becomes 5, via z; y tells z than d(y,x)=5
Then, z→x becomes 6, via y; z tells y than d(z,x)=6
This repeats until the path cost larger than 50

Comparison of LS and DV algorithms

Message complexity
- LS: with n nodes, E links, O(nE) msgs sent
- DV: exchange between neighbors only
  - **convergence time varies**

Speed of Convergence
- LS: O(n^2) algorithm requires O(nE) msgs
  - may have oscillations
- DV: convergence time varies
  - may be routing loops
  - count-to-infinity problem

Robustness: what happens if router malfunctions?

LS:
- node can advertise incorrect *link* cost
- each node computes only its *own* table

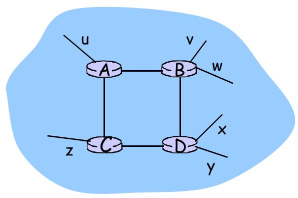
DV:
- DV node can advertise incorrect *path* cost
- each node's table used by others
  - **error propagate thru network**

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

RIP (Routing Information Protocol)

- (Local, decentralized) distance vector (**DV**) algorithm
- distance metric: # of hops (max = 15 hops) 距离



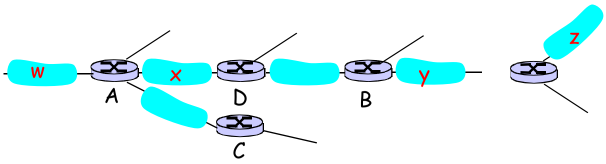
From router A to subnets:

destination	hops
subnet u	1
subnet v	2
subnet w	2
subnet x	3
subnet y	3
subnet z	2

RIP advertisements

- *distance vectors:* exchanged among neighbors every 30 sec via Response Message (also called advertisement)
- 每 30 秒更新一次，邻居路由器之间的信息交换
- each advertisement: list of up to 25 destination subnets

RIP: Example



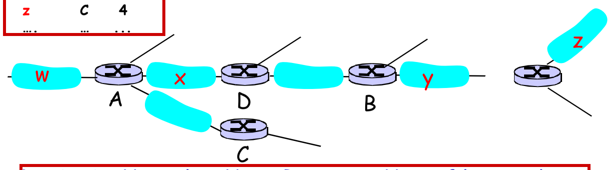
Destination Network	Next Router	Num. of hops to dest.
w	A	2
y	B	2
z	B	7
x	--	1
….	….	….

Routing/Forwarding table in D

RIP: Example

Dest	Next	hops
w	-	1
x	-	1
z	C	4
….	…	…

Advertisement from A to D



Destination Network	Next Router	Num. of hops to dest.
w	A	2
y	B	2
z	~~B~~ A	~~7~~ 5
x	--	1
….	….	….

Routing/Forwarding table in D

OSPF (Open Shortest Path First)

- "open": publicly available
- uses **Link State algorithm**
  - LS packet dissemination
  - topology map at each node
  - route computation using Dijkstra's algorithm
- OSPF advertisement carries one entry per neighbor router
- advertisements disseminated to entire AS via **flooding** **开销太大**
  - carried in OSPF messages directly over IP (rather than TCP or UDP

OSPF "advanced" features (not in RIP)

- security: all OSPF messages authenticated (to prevent malicious intrusion, **net layer auth. using MD5 hash**)
- multiple same-cost paths allowed (only one path in RIP)
- Integrated uni- and multicast support:
  - Multicast OSPF (MOSPF) uses same topology data base as OSPF
- hierarchical OSPF in large domains.

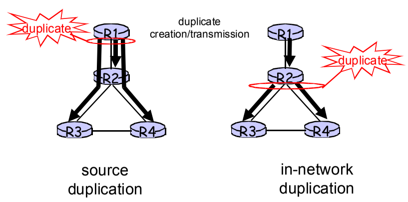
Internet inter-AS routing: BGP

- BGP (Border Gateway Protocol): *the* de facto standard **事实上在用的标准**
- BGP provides each AS a means to:
  1. Obtain subnet reachability information from neighboring ASs.
  2. Propagate reachability information to all AS-internal routers.
  3. Determine "good" routes to subnets based on reachability information and policy.
- allows subnet to advertise its existence to rest of Internet: *"I am here"*

Chapter 4: Network Layer

Broadcast (广播) Routing

- deliver packets from source to all other nodes
- source duplication is inefficient:



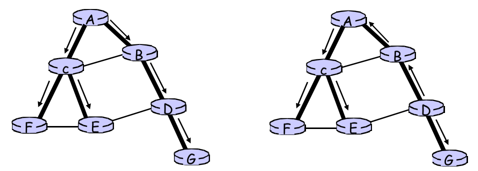
- source duplication: how does source determine recipient addresses?

In-network duplication

- flooding: when node receives brdcst pckt, sends copy to all neighbors (**infinite cascading**)
  - Problems: cycles & broadcast storm
- controlled flooding: node only brdcsts pkt if it hasn't brdcst same packet before (只发给没收到的)
  - Node keeps track of pckt ids already brdcsted
  - Or reverse path forwarding (RPF): only forward pckt if it arrived on shortest path between node and source
- spanning tree (最优解)
  - No redundant packets received by any node

Spanning Tree

- First construct a spanning tree
- Nodes forward copies only along spanning tree **沿着 tree 发送以避免重复**

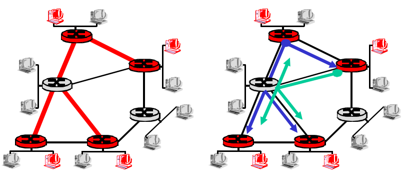


(a) Broadcast initiated at A (b) Broadcast initiated at D

B 和 C 之间的 broadcast pkt 无需再转发

Multicast Routing: Problem Statement

- ***Goal:*** find a tree (or trees) connecting routers having local mcast group members
  - *tree:* not all paths between routers used
  - *source-based:* different tree from each sender to rcvrs
  - *shared-tree:* same tree used by all group members



Shared tree Source-based trees

所有红色 hosts 是一个 mcast group

如何完成群发?

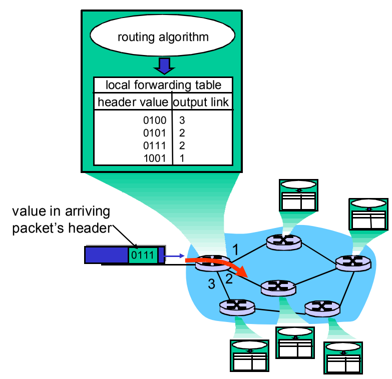
Chapter 4: summary

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 **What's inside a router**
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - **Link state**
  - **Distance Vector**
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 **Broadcast and multicast routing**

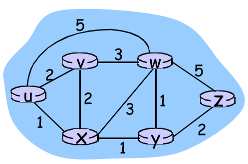
Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

Interplay between routing, forwarding



Graph abstraction



Graph: G = (N,E)

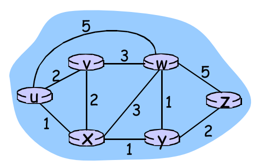
N = set of routers (nodes) = { u, v, w, x, y, z }

E = set of links (edges) ={ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

Remark: Graph abstraction is useful in other network contexts

Example: P2P, where N is set of peers and E is set of TCP connections

Graph abstraction: costs



- $c(x,x')$ = cost of link (x,x')
  - e.g., $c(w,z) = 5$
- **cost** could always be 1, or inversely related to bandwidth, or inversely related to congestion

Cost of path $(x_1, x_2, x_3,\ldots, x_p) = c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$

Question: What's the least-cost path between u and z ?

Routing algorithm: algorithm that finds **least-cost path**

Routing Algorithm classification

Global or local (decentralized) information?

Global:
- all routers have complete topology, link cost info
- "link state" algorithms

Local/Decentralized:
- router knows physically-connected neighbors, link costs to neighbors
- iterative process of computation, exchange of info with neighbors
- "distance vector" algorithms

Static or dynamic?

Static:
- routes change slowly over time

Dynamic:
- routes change more quickly
  - periodic update
  - in response to link cost changes

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

A Link-State (Global) Routing Algorithm

Dijkstra's algorithm
- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ('source") to all other nodes
  - gives forwarding table for that node
- iterative: after k iterations, know least cost path to k dest.'s

Notation:
- c(x,y): link cost from node x to y; = ∞ if not direct neighbors
- D(v): current value of cost of path from source to dest. v
- p(v): predecessor node along path from source to v
- N': set of nodes whose least cost path definitively known

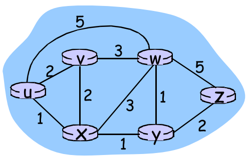
Dijsktra's Algorithm: greedy

```
Initialization:
  N' = {u} // starting from source node
  for all nodes v
    if v adjacent to u
       then D(v) = c(u,v) // neighbor connected
    else D(v) = ∞ // not connected

Loop
  find w not in N' such that D(w) is a minimum // greedy!
  add w to N'
  update D(v) for all v adjacent to w and not in N' :
     D(v) = min( D(v), D(w) + c(w,v) )
  /* new cost to v is either old cost to v or known
   shortest path cost to w plus cost from w to v */
until all nodes in N'
```

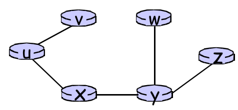

Dijkstra's algorithm: example

Step	N'	D(v),p(v)	D(w),p(w)	D(x),p(x)	D(y),p(y)	D(z),p(z)
0	u	2,u	5,u	1,u	∞	∞
1	ux	2,u	4,x		2,x	∞
2	uxy	2,u	3,y			4,y
3	uxyv		3,y			4,y
4	uxyvw					4,y
5	uxyvwz					



Dijkstra's algorithm: example (2)

Resulting shortest-path tree from u:



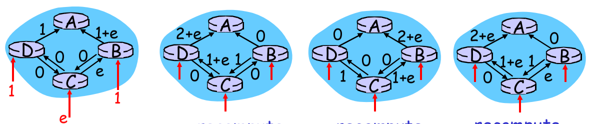
Resulting forwarding table in u:

destination	link output port
v	(u,v)
x	(u,x)
y	(u,x)
w	(u,x)
z	(u,x)

Dijkstra's algorithm, discussion

Algorithm complexity: n nodes
- each iteration: need to check all nodes, w, not in N
- n(n+1)/2 comparisons: $O(n^2)$
- more efficient implementations possible (using a heap): O(nlogn)

Oscillations possible (dynamically changing cost):
- e.g., link cost = amount of carried traffic
- Link costs are not symmetric



initially ... recompute routing ... recompute ... recompute

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

Distance Vector Algorithm

Bellman-Ford Equation (dynamic programming)

Define

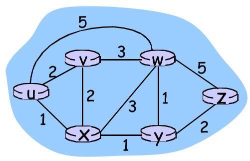
$d_x(y)$:= cost of least-cost path from x to y

Then

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

where min is taken over all neighbors v of x

Bellman-Ford example



B-F equation says:

$$d_u(z) = \min \{ c(u,v) + d_v(z),\ c(u,x) + d_x(z),\ c(u,w) + d_w(z) \}$$
$$= \min \{2 + 5,\ 1 + 3,\ 5 + 3\} = 4$$

Clearly, $d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$

Node that achieves minimum is next hop in shortest path → forwarding table

Distance Vector Algorithm

- $D_x(y)$ = estimate of least cost from x to y
- Node x knows cost to each neighbor v: $c(x,v)$
- Node x maintains distance vector $\mathbf{D}_x = [D_x(y): y \in N]$
- Node x also maintains its neighbors' distance vectors
  - For each neighbor v, x maintains $\mathbf{D}_v = [D_v(y): y \in N]$

Distance vector algorithm (4)

Basic idea:

- From time-to-time, each node sends its own distance vector estimate to neighbors
- Asynchronous **异步的**
- When a node x receives new DV estimate from neighbor, it updates its own DV using B-F equation:

  $D_x(y) \leftarrow \min_v\{c(x,v) + D_v(y)\}$ for each node $y \in N$

- Under minor, natural conditions, the estimate $D_x(y)$ ***converge to*** *the actual least cost* $d_x(y)$
  - **最终会逼近实际距离**

Distance Vector Algorithm (5)

Iterative, asynchronous: each local iteration caused by:
- local link cost change
- DV update message from neighbor

Distributed:
- **邻居信息，无需全局信息**
- each node notifies neighbors *only* when its DV changes
  - neighbors then notify their neighbors if necessary

Each node:

wait for (change in local link cost or msg from neighbor)

↓

recompute estimates

↓

if DV to any dest has changed, *notify* neighbors

$D_x(y) = \min\{c(x,y) + D_y(y),\ c(x,z) + D_z(y)\} = \min\{2+0,\ 7+1\} = 2$

$D_x(z) = \min\{c(x,y) + D_y(z),\ c(x,z) + D_z(z)\} = \min\{2+1,\ 7+0\} = 3$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

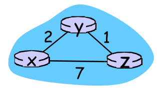
node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

time



$D_x(y) = \min\{c(x,y) + D_y(y),\ c(x,z) + D_z(y)\} = \min\{2+0,\ 7+1\} = 2$

$D_x(z) = \min\{c(x,y) + D_y(z),\ c(x,z) + D_z(z)\} = \min\{2+1,\ 7+0\} = 3$

node x table

from \ cost to	x	y	z
x	0	2	7
y	∞	∞	∞
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	7	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

node y table

from \ cost to	x	y	z
x	∞	∞	∞
y	2	0	1
z	∞	∞	∞

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	7	1	0

from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

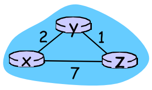
node z table

from \ cost to	x	y	z
x	∞	∞	∞
y	∞	∞	∞
z	7	1	0

from \ cost to	x	y	z
x	0	2	7
y	2	0	1
z	3	1	0

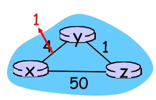
from \ cost to	x	y	z
x	0	2	3
y	2	0	1
z	3	1	0

time



Distance Vector: link cost changes

Link cost changes:
- node detects local link cost change
- updates routing info, recalculates distance vector
- if DV changes, notify neighbors



"good news travels fast"

At time t_0, *y* detects the link-cost change, updates its DV, and informs its neighbors.

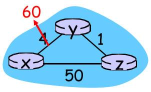
At time t_1, *z* receives the update from *y* and updates its table. It computes a new least cost to *x* and sends its neighbors its DV.

At time t_2, *y* receives *z*'s update and updates its distance table. *y*'s least costs do not change and hence *y* does *not* send any message to *z*.

Distance Vector: link cost changes

Link cost changes:
- good news travels fast
- **bad news travels slow - "count to infinity" problem!**
- 44 iterations before algorithm stabilizes: see text

Poisoned reverse:
- If Z routes through Y to get to X :
  - Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z)
- will this completely solve count to infinity problem?



4→60, but z does not know;
z→x is still 5
Then, Y→x becomes 5, via z; y tells z than d(y,x)=5
Then, z→x becomes 6, via y; z tells y than d(z,x)=6
This repeats until the path cost larger than 50

Comparison of LS and DV algorithms

Message complexity
- LS: with n nodes, E links, O(nE) msgs sent
- DV: exchange between neighbors only
  - **convergence time varies**

Speed of Convergence
- LS: $O(n^2)$ algorithm requires O(nE) msgs
  - may have oscillations
- DV: convergence time varies
  - may be routing loops
  - count-to-infinity problem

Robustness: what happens if router malfunctions?

LS:
- node can advertise incorrect *link* cost
- each node computes only its *own* table

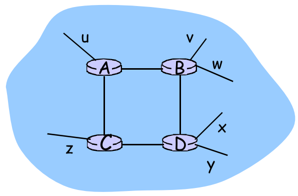
DV:
- DV node can advertise incorrect *path* cost
- each node's table used by others
  - **error propagate thru network**

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

RIP (Routing Information Protocol)

- (Local, decentralized) distance vector (DV) algorithm
- distance metric: # of hops (max = 15 hops) **距离**



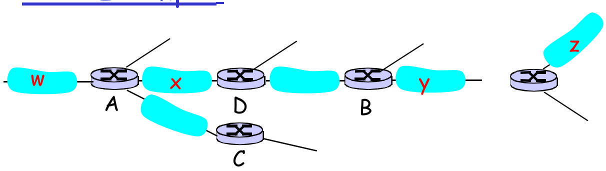
From router A to subnets:

destination	hops
subnet u	1
subnet v	2
subnet w	2
subnet x	3
subnet y	3
subnet z	2

RIP advertisements

- *distance vectors:* exchanged among neighbors every 30 sec via Response Message (also called advertisement)
- 每 30 秒更新一次，邻居路由器之间的信息交换
- each advertisement: list of up to 25 destination subnets

RIP: Example



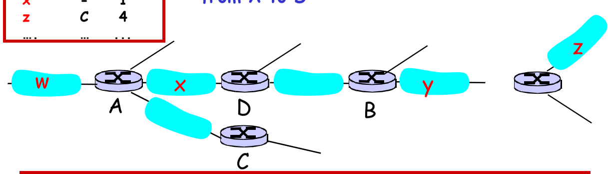
Destination Network	Next Router	Num. of hops to dest.
w	A	2
y	B	2
z	B	7
x	--	1
….	….	….

Routing/Forwarding table in D

RIP: Example

Dest	Next	hops
w	-	1
x	-	1
z	C	4
….	…	...

Advertisement from A to D



Destination Network	Next Router	Num. of hops to dest.
w	A	2
y	B	2
z	~~B~~ A	~~7~~ 5
x	--	1
….	….	….

Routing/Forwarding table in D

OSPF (Open Shortest Path First)

- "open": publicly available
- uses **Link State algorithm**
  - LS packet dissemination
  - topology map at each node
  - route computation using **Dijkstra**'s algorithm
- OSPF advertisement carries one entry per neighbor router
- advertisements disseminated to entire AS via **flooding 开销太大**
  - carried in OSPF messages directly over IP (rather than TCP or UDP

OSPF "advanced" features (not in RIP)

- security: all OSPF messages authenticated (to prevent malicious intrusion, **net layer auth. using MD5 hash**)
- multiple same-cost paths allowed (only one path in RIP)
- Integrated uni- and multicast support:
  - Multicast OSPF (MOSPF) uses same topology data base as OSPF
- hierarchical OSPF in large domains.

Internet inter-AS routing: BGP

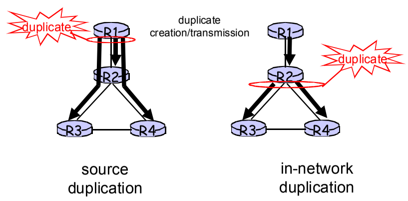
- BGP (Border Gateway Protocol): *the* de facto standard **事实上在用的标准**
- BGP provides each AS a means to:
  1. Obtain subnet reachability information from neighboring ASs.
  2. Propagate reachability information to all AS-internal routers.
  3. Determine "good" routes to subnets based on reachability information and policy.
- allows subnet to advertise its existence to rest of Internet: *"I am here"*

Chapter 4: Network Layer

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 What's inside a router
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - Link state
  - Distance Vector
  - Hierarchical routing
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 Broadcast and multicast routing

Broadcast (广播) Routing

- deliver packets from source to all other nodes
- source duplication is inefficient:



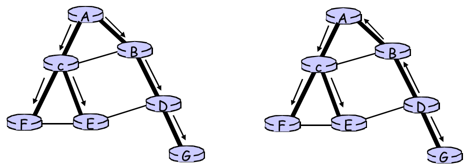
- source duplication: how does source determine recipient addresses?

In-network duplication

- flooding: when node receives brdcst pckt, sends copy to all neighbors (**infinite cascading**)
  - Problems: cycles & broadcast storm
- controlled flooding: node only brdcsts pkt if it hasn't brdcst same packet before (只发给没收到的)
  - Node keeps track of pckt ids already brdcsted
  - Or reverse path forwarding (RPF): only forward pckt if it arrived on shortest path between node and source
- spanning tree (最优解)
  - No redundant packets received by any node

Spanning Tree

- First construct a spanning tree
- Nodes forward copies only along spanning tree **沿着 tree 发送以避免重复**

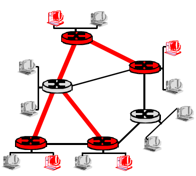


(a) Broadcast initiated at A (b) Broadcast initiated at D

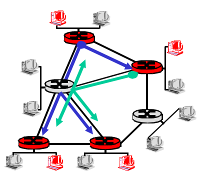
B 和 C 之间的 broadcast pkt 无需再转发

Multicast Routing: Problem Statement

- ***Goal:*** find a tree (or trees) connecting routers having local mcast group members
  - *tree:* not all paths between routers used
  - *source-based:* different tree from each sender to rcvrs
  - *shared-tree:* same tree used by all group members



Shared tree



Source-based trees

所有红色 hosts 是一个 mcast group

如何完成群发?

Chapter 4: summary

- 4. 1 Introduction
- 4.2 Virtual circuit and datagram networks
- 4.3 **What's inside a router**
- 4.4 IP: Internet Protocol
  - Datagram format
  - IPv4 addressing
  - ICMP
  - IPv6
- 4.5 Routing algorithms
  - **Link state**
  - **Distance Vector**
- 4.6 Routing in the Internet
  - RIP
  - OSPF
  - BGP
- 4.7 **Broadcast and multicast routing**

File into TCP segments

- **rdt: a pkt stream; each pkt has a seq. #**
- **TCP: a byte stream; each byte has seq. #**

Q: how a file (500,000 bytes) divided into TCP segments? MSS = 1000 bytes.

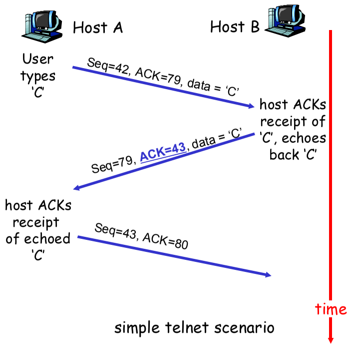
- 500, 000 bytes can create 500 segments.
- 1st byte seq. # is 0, the last byte seq. # is 499,999
- Segment's seq. # is its 1st byte's seq. #
  - Ex. 1st segment seq # = 0; 2nd segment seq # = 1000

Telnet: TCP seq. #'s and ACKs

Full duplex data

Seq. #'s:
- "number" of first byte in segment's data

ACKs:
- seq # of next byte expected from other side (ACK=43)
- cumulative ACK (ACK up to last rcved byte seq. #)



simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

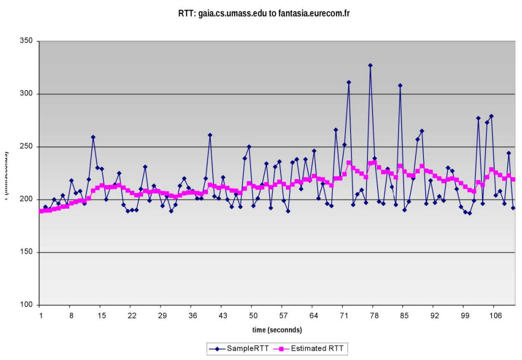
Q: how to estimate RTT?
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current `SampleRTT`

TCP Round Trip Time and Timeout

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125
- **Heuristic: May not be the best**

Example RTT estimation:



Est. RTT can roughly reflect the real RTT value.

TCP Round Trip Time and Timeout

Setting the timeout
- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT-EstimatedRTT}|$$

(typically, β = 0.25)

统计学概念：指数加权移动平均 EWMA (Exponentially Weighted Moving Average)

Then set timeout interval:

Weighted approach

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

rdt = reliable data transfer

Reliable

- Data + **reply** (ACK/NACK)

- Data pkt, or ACK may be
  - Corrupted (Bit error) → checksum
  - Lost → seq. # to prepvent dulicate receptions
  - Delayed → countdown timer

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Connectionless transport: UDP
- 3.3 Connection-oriented transport: TCP
  - segment structure
  - TCP's reliable data transfer
  - flow control
  - connection management
- 3.4 Principles of congestion control
- 3.5 TCP congestion control

TCP reliable data transfer

Features inherited from rdt's principles

- **TCP's rdt** service on top of IP's unreliable service
- **Pipelined** segments
- **Cumulative** acks
  - **Ack up to the last received byte**
- TCP uses single retransmission **timer**

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider **simplified/ simplest** TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of **timer as for oldest un-acked segment**)
- expiration interval: `TimeOutInterval`

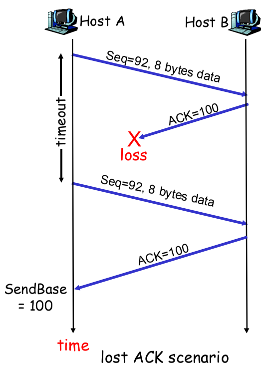
timeout:

- retransmit segment that caused timeout
- restart timer

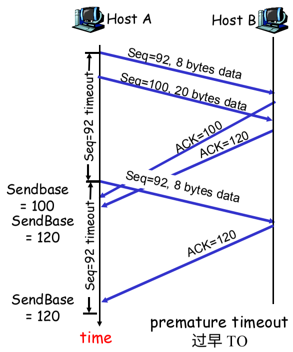
Ack rcvd:

- If Rx acknowledges previously unacked segments
  - update what is known to be acked
  - **start a new timer** for a new outstanding segments

TCP: retransmission scenarios

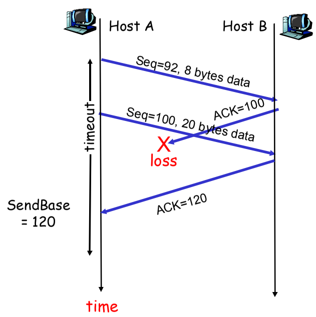


lost ACK scenario



premature timeout
过早 TO

TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK (**网络通畅下一个也许马上来**)
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments (**正要发 ACK，来了一个新 segment，一次两个**)
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send ***duplicate ACK***, indicating seq. # of next expected byte (**顺序错乱，中间缺一段**)
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap (**顺序错乱，中间缺一段，从最底部开始补**)

Fast Retransmit=Re-Tx b4 TO

- **Re-transmit before timeout**
- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back (sender 一个接一个，背靠背地发送 segment)
  - If segment is lost, there will likely be many duplicate ACKs.
- Trigger event for fast re-Tx: If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - **fast retransmit: resend segment before timer expires**

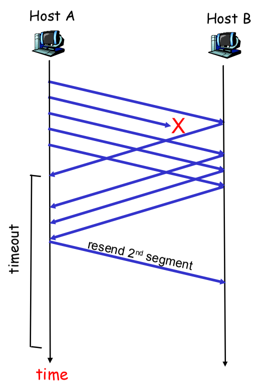


Figure 3.37 Resending a segment after triple duplicate ACK

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
               SendBase = y
               if (there are currently not-yet-acknowledged segments)
                    start timer
          }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
               }
```

a duplicate ACK for already ACKed segment

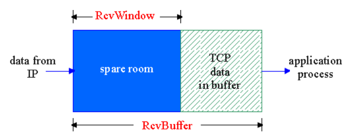
fast retransmit

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:

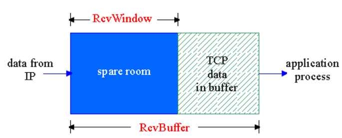


- app process may be slow at reading from buffer

flow control
sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate
- 使发送端的发送速度，尽量匹配，接收端的读取速度

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of `RcvWindow` in segments
- Sender limits unACKed data to `RcvWindow`
  - guarantees receive buffer doesn't overflow
- 发送端将要发送的数据，少于，接收端空余容量
  - 接收端将空余容量告知发送端

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator
  `Socket clientSocket = new Socket("hostname","port number");`
- *server:* contacted by client
  `Socket connectionSocket = welcomeSocket.accept();`

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

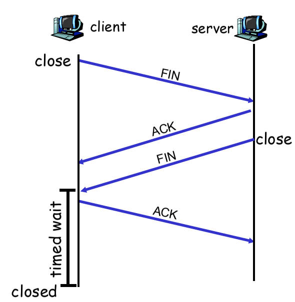
TCP Connection Management (cont.)

Closing a connection:

client closes socket: `clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

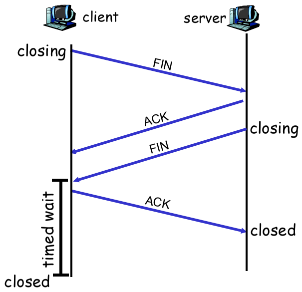


TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.
- Enters "timed wait"
- Respond with ACK to received FINs

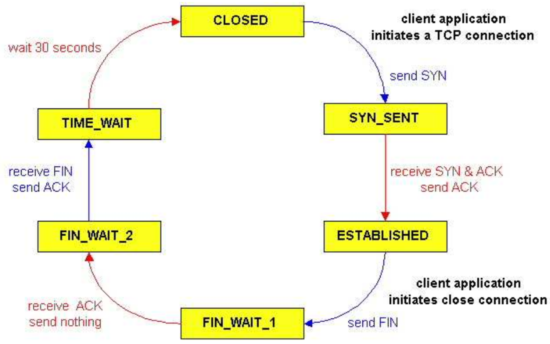
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



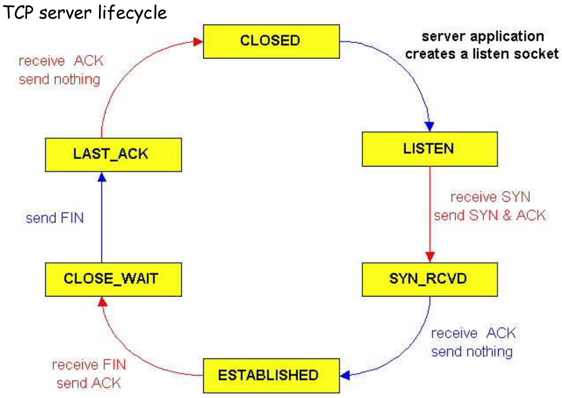
TCP Connection Management (cont)

TCP client lifecycle



TCP Connection Management (cont)

TCP server lifecycle



Chapter 3 outline

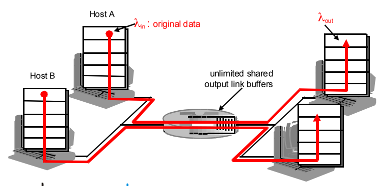
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

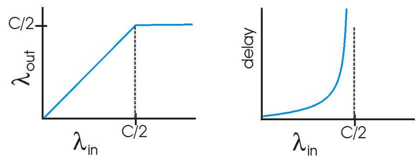
Principles of Congestion Control

Congestion:
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
  - **Flow control：接收端能力有限，忙不过来**
  - **Congestion control: 网络太拥挤，发不过来**
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- **a top-10 problem!**

Causes/costs of congestion: scenario 1

- Case 1： router has **infinite buffers**
- two senders, two receivers
- no retransmission

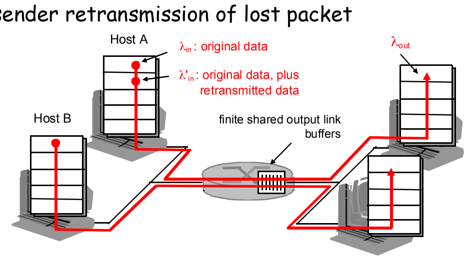




Throughput and delay

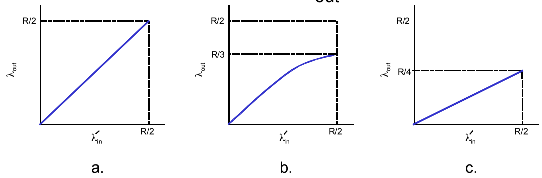
Causes/costs of congestion: scenario 2

- Case 2: one router, ***finite*** buffers
- Pkt loss at router
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}

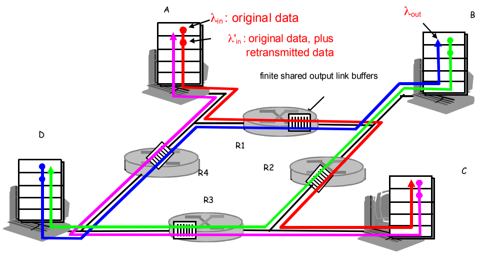


"costs" of congestion:　　**Goodput = 实际吞吐量，有效吞吐量**
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: scenario 3

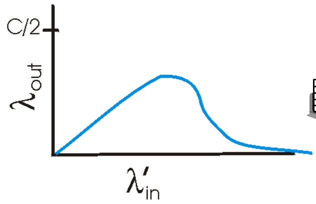
- four senders
- multihop paths
- timeout/retransmit

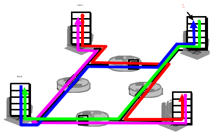
Q: what happens as λ_{in} and λ'_{in} increase ?



Flow B-D outcompetes flow A-C at R2!!!

Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Flow A-C: if pkt loss at R2, then resource forwarding this pkt at R1 is wasted!!!

Approaches towards congestion control

Two broad approaches towards congestion control:

Network-assisted congestion control:
- 网络辅助拥塞控制
- 网络路由器发送信息辅助
- routers provide feedback to end systems
  - single bit indicating congestion
  - explicit rate sender should send at

End-end congestion control:
- 端到端拥塞控制
- 接收端 ACK 辅助，发送端猜测
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

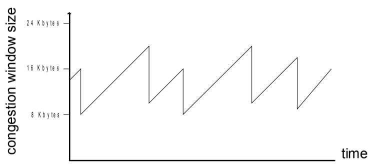
TCP

- Rdt
  - ACK
  - Checksum
  - Seq. #
  - Timer
  - Flow control
- TCP
  - Flow control
  - TCP Congestion control
    - Additive increase, multiplicative decrease (AIMD 线性增加、成倍递减)
    - Congestion window (CongWin)
    - Slow start
    - Collision avoidance
    - Threshold
    - Fast re-tx and fast recovery

TCP congestion control: additive increase, multiplicative decrease (AIMD)

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **AI**, *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - **MD**, *multiplicative decrease:* cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



Congestion window (CongWin) = # of bytes that are allowed to be sent in next RTT

TCP Congestion Control: details

- Window: sender limits transmission such as
  `LastByteSent-LastByteAcked ≤ CongWin`
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

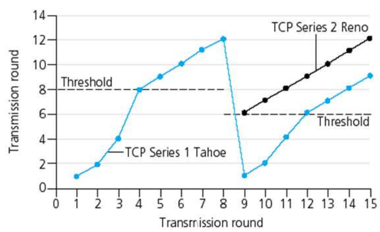
- `CongWin` is dynamic, function of perceived network congestion

How does sender perceive congestion/loss?
- Timeout *or* 3 duplicate acks
- TCP sender reduces rate (`CongWin`) after loss event

three mechanisms:
- AIMD
- slow start
- conservative after timeout events

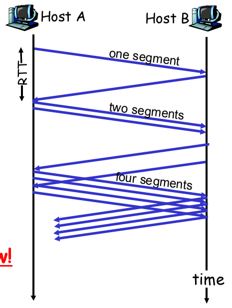
TCP Slow Start

- When connection begins, `CongWin` = 1 MSS
  - If MSS = 500 bytes, and RTT = 200 msec
  - initial rate = 500/0.2*8 = 20 kb/s
- Available bandwidth much greater than MSS/RT
- Thus, quickly increase the **CongWin** by 1 MSS for each received ACK
  - 1MSS→ 2MSS→ 4MSS。。。



TCP Slow Start (more)

- When connection begins, increase rate exponentially
  - by incrementing CongWin for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

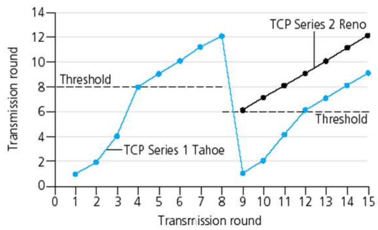


"Slow start" = starting rate is slow!
The increase speed is fast!

Collision avoidance: linear increase

Slow start to linear increase at a threshold

Q: **What is threshold value?**
A: **Half of CongWin before timeout.**



Implementation:

- At loss event, Threshold is set to 1/2 of CongWin just before loss event

Fast re-tx and fast recovery

- Fast re-tx and fast recovery after 3 dup ACKs (pkt loss):
  - CongWin is cut to half
  - window then grows linearly
- But after timeout event:
  - CongWin instead set to 1 MSS (to initial start);
  - Slow start: window then grows exponentially
  - Congestions avoidance: Linear growth to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in **slow-start** phase, window grows exponentially.
- When `CongWin` is above `Threshold`, sender is in **congestion-avoidance** phase, window grows **linearly**.
- When a triple duplicate ACK occurs, re-Tx the segment **(fast re-Tx)**. `Threshold` set to `CongWin/2` **(fast recovery)** and `CongWin` set to `Threshold`.
- When **timeout** occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS. Enter slow-start phase again.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	**Additive increase**, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by **triple duplicate ACK**	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	**Fast recovery**, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	**Timeout**	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP average throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let *W* be the window size when loss occurs.
  - When window is *W*, throughput is *W/RTT*
  - Just after loss, window drops to *W/2*, throughput to *W/2RTT*.
- Average throughout: .75 W/RTT

TCP Futures: TCP over "long, fat pipes"

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

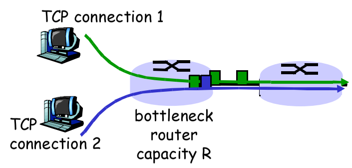
$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- → $L = 2 \cdot 10^{-10}$ *difficult to achieve*
- New versions of TCP for high-speed

TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

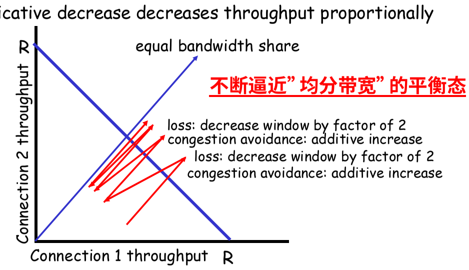
Absolute fairness



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- **Multimedia** apps often do not use TCP
  - **do not want rate throttled by congestion control**
- Instead use UDP:
  - pump audio/video at constant rate, tolerate packet loss
- Research area: TCP friendly

在线视频音频可以有错误但是不能突然中断服务

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts.
- **Web browsers** do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets **more than R/2** !

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing (socket and port #)
  - reliable data transfer
  - flow control（发送端与接收端之间协调控制）
  - congestion control(基于对网络拥挤程度猜想的控制）
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"