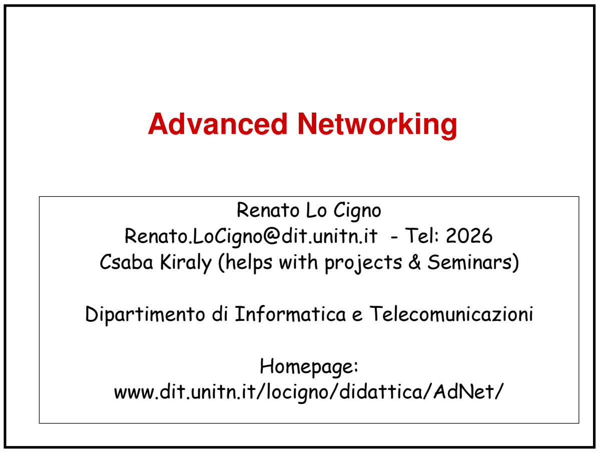
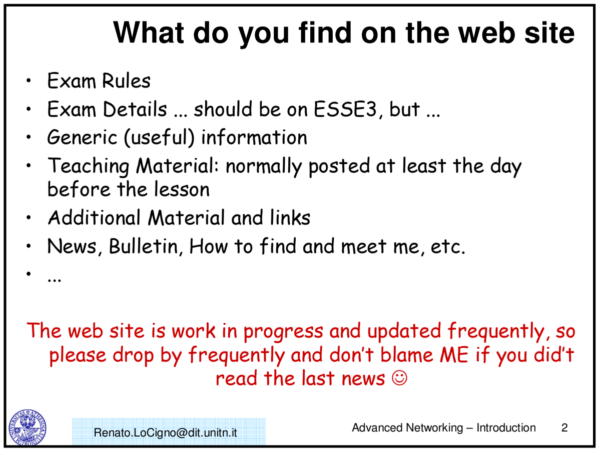


Advanced Networking

Renato Lo Cigno
Renato.LoCigno@dit.unitn.it - Tel: 2026
Csaba Kiraly (helps with projects & Seminars)

Dipartimento di Informatica e Telecomunicazioni

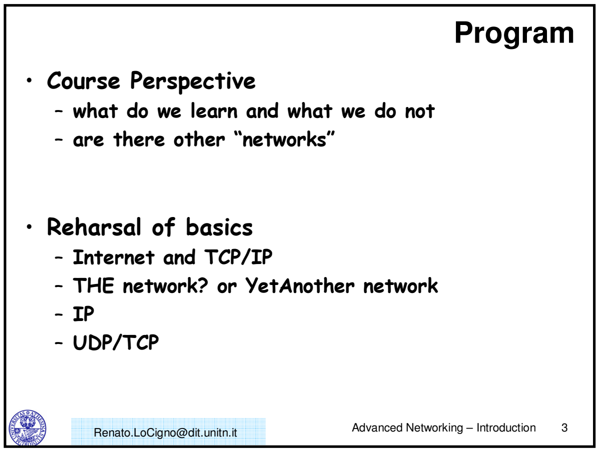
Homepage:
www.dit.unitn.it/locigno/didattica/AdNet/



What do you find on the web site

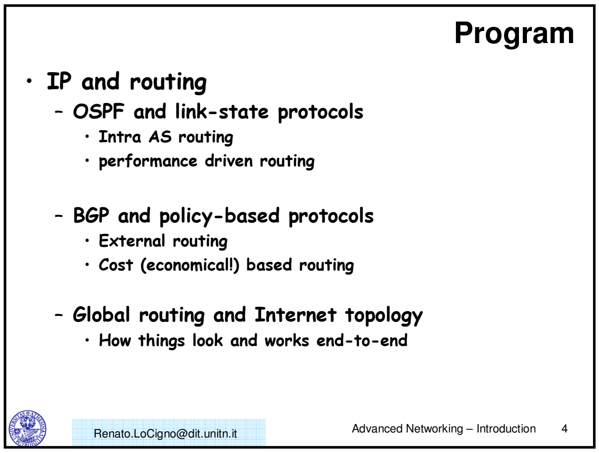
- Exam Rules
- Exam Details … should be on ESSE3, but …
- Generic (useful) information
- Teaching Material: normally posted at least the day before the lesson
- Additional Material and links
- News, Bulletin, How to find and meet me, etc.
- …

The web site is work in progress and updated frequently, so please drop by frequently and don’t blame ME if you did’t read the last news ☺



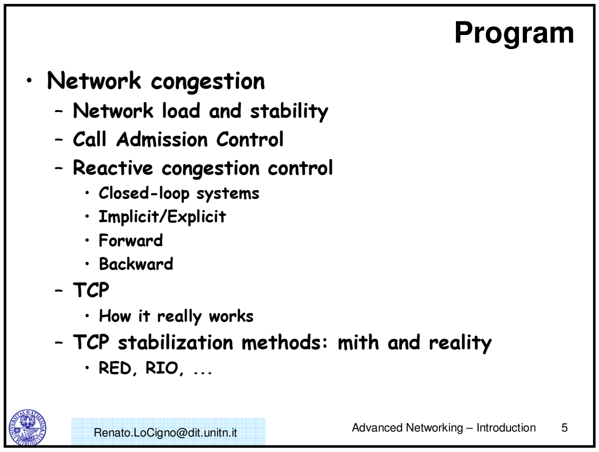
Program

- **Course Perspective**
  - **what do we learn and what we do not**
  - **are there other “networks”**
- **Reharsal of basics**
  - **Internet and TCP/IP**
  - **THE network? or YetAnother network**
  - **IP**
  - **UDP/TCP**



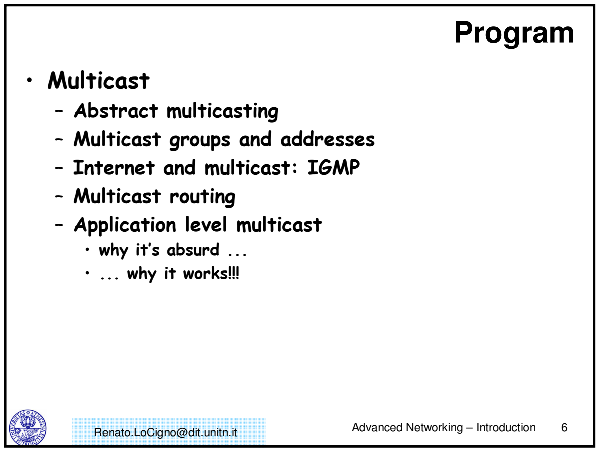
Program

- IP and routing
  - OSPF and link-state protocols
    - Intra AS routing
    - performance driven routing
  - BGP and policy-based protocols
    - External routing
    - Cost (economical!) based routing
  - Global routing and Internet topology
    - How things look and works end-to-end



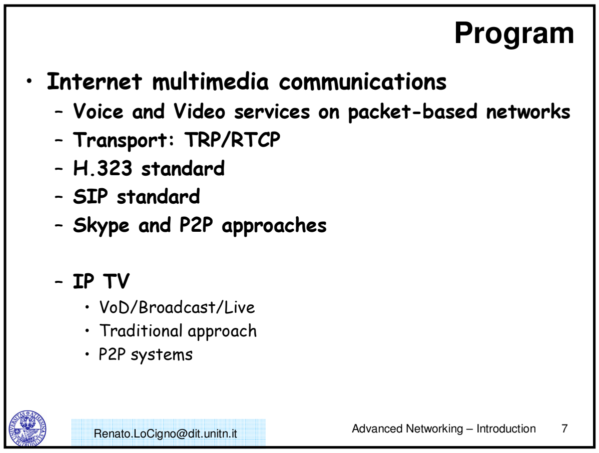
Program

- Network congestion
  - Network load and stability
  - Call Admission Control
  - Reactive congestion control
    - Closed-loop systems
    - Implicit/Explicit
    - Forward
    - Backward
  - TCP
    - How it really works
  - TCP stabilization methods: mith and reality
    - RED, RIO, ...



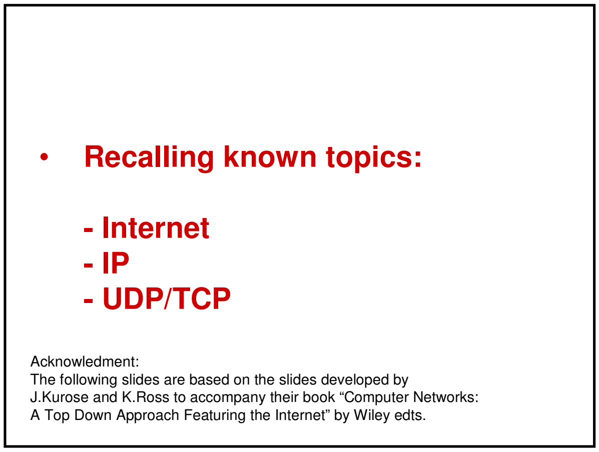
Program

- Multicast
  - Abstract multicasting
  - Multicast groups and addresses
  - Internet and multicast: IGMP
  - Multicast routing
  - Application level multicast
    - why it's absurd ...
    - ... why it works!!!



Program

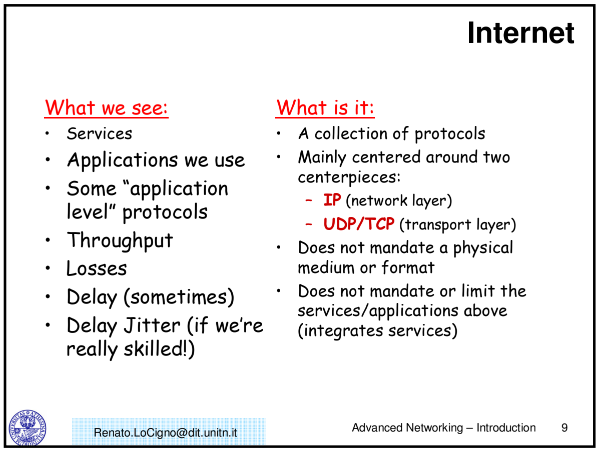
- Internet multimedia communications
  - Voice and Video services on packet-based networks
  - Transport: TRP/RTCP
  - H.323 standard
  - SIP standard
  - Skype and P2P approaches
  - IP TV
    - VoD/Broadcast/Live
    - Traditional approach
    - P2P systems



- **Recalling known topics:**

  - Internet
  - IP
  - UDP/TCP

Acknowledment:
The following slides are based on the slides developed by
J.Kurose and K.Ross to accompany their book “Computer Networks:
A Top Down Approach Featuring the Internet” by Wiley edts.



Internet

What we see:

- Services
- Applications we use
- Some “application level” protocols
- Throughput
- Losses
- Delay (sometimes)
- Delay Jitter (if we’re really skilled!)

What is it:

- A collection of protocols
- Mainly centered around two centerpieces:
  - IP (network layer)
  - UDP/TCP (transport layer)
- Does not mandate a physical medium or format
- Does not mandate or limit the services/applications above (integrates services)

IP: The Network Layer

Goals:

- recall principles behind network layer services:
  - routing (path selection)
  - dealing with scale
  - how a router works
- instantiation and implementation in the Internet

Overview:

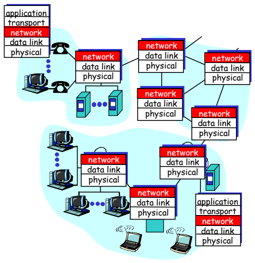
- network layer services
- routing principle: path selection
- IP
- Internet routing protocols reliable transfer
  - intra-domain
  - inter-domain
- what's inside a router?

Network layer functions

- transport packet from sending to receiving hosts
- network layer protocols in *every* host, router

three important functions:

- *path determination:* route taken by packets from source to dest. *Routing algorithms*
- *switching:* move packets from router's input to appropriate router output
- *call setup:* some network architectures require router call setup along path before data flows



Network service model

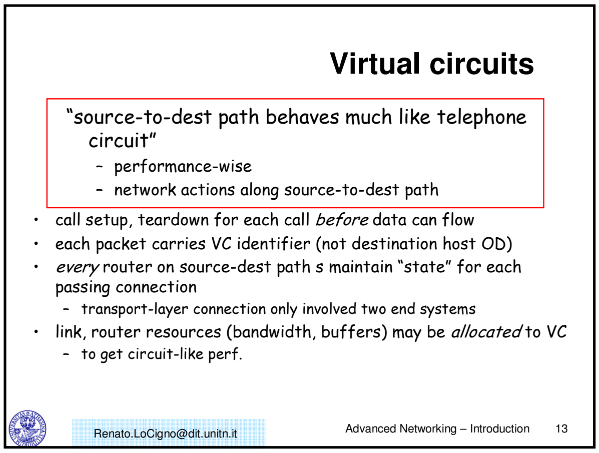
Q: What *service model* for "channel" transporting packets from sender to receiver?

service abstraction

- guaranteed bandwidth?
- preservation of inter-packet timing (no jitter)?
- loss-free delivery?
- in-order delivery?
- congestion feedback to sender?

The most important abstraction provided by network layer:

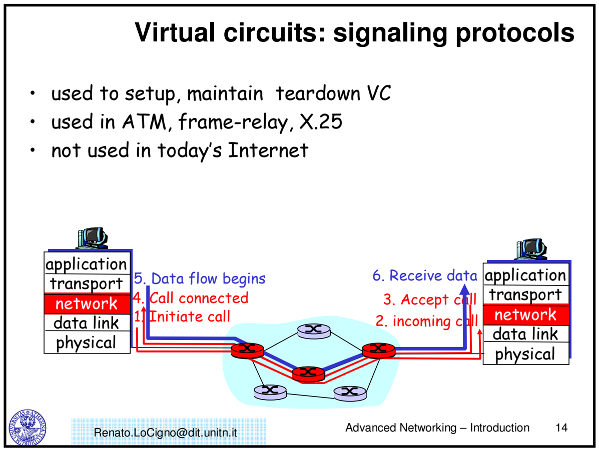
virtual circuit
or
datagram?



Virtual circuits

"source-to-dest path behaves much like telephone circuit"
- performance-wise
- network actions along source-to-dest path

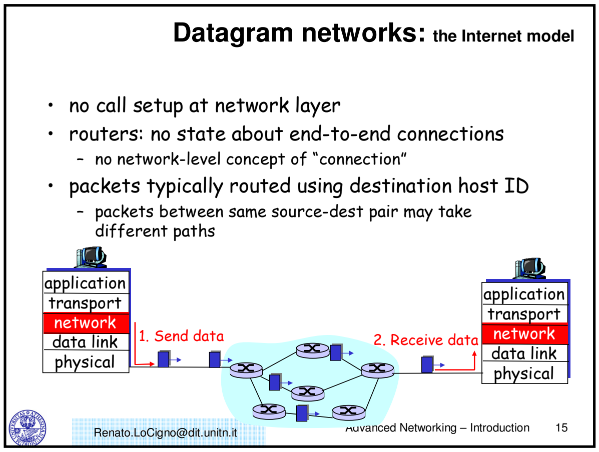
- call setup, teardown for each call *before* data can flow
- each packet carries VC identifier (not destination host OD)
- *every* router on source-dest path s maintain "state" for each passing connection
  - transport-layer connection only involved two end systems
- link, router resources (bandwidth, buffers) may be *allocated* to VC
  - to get circuit-like perf.



Virtual circuits: signaling protocols

- used to setup, maintain teardown VC
- used in ATM, frame-relay, X.25
- not used in today's Internet

application
transport
network
data link
physical
5. Data flow begins
4. Call connected
1. Initiate call
6. Receive data
3. Accept call
2. incoming call
application
transport
network
data link
physical



Datagram networks: the Internet model

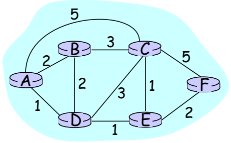
- no call setup at network layer
- routers: no state about end-to-end connections
  - no network-level concept of "connection"
- packets typically routed using destination host ID
  - packets between same source-dest pair may take different paths

application
transport
network
data link
physical
1. Send data
2. Receive data
application
transport
network
data link
physical

Routing

Routing protocol

Goal: determine "good" path (sequence of routers) thru network from source to dest.



Graph abstraction for routing algorithms:

- graph nodes are routers
- graph edges are physical links
  - link cost: delay, $ cost, or congestion level

- "good" path:
  - typically means minimum cost path
  - other def's possible

Routing Algorithm classification

Global or decentralized information?

Global:

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

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

A Link-State 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 routing table for that node
- iterative: after k iterations, know least cost path to k dest.'s

Notation:

- c(i,j): link cost from node i to j. cost infinite 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, that is next v
- N: set of nodes whose least cost path definitively known

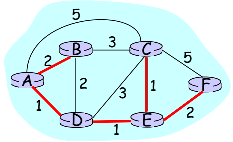
Dijsktra's Algorithm

```
***Initialization:***
  N = {A}
  for all nodes v
    if v adjacent to A
      then D(v) = c(A,v)
      else D(v) = infty

 ***Loop***
   find w not in N such that D(w) is a minimum
   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	start N	D(B),p(B)	D(C),p(C)	D(D),p(D)	D(E),p(E)	D(F),p(F)
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E
5	ADEBCF					



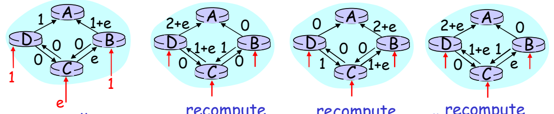
Dijkstra's algorithm, discussion

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: O(nlogn)

Oscillations possible:

- e.g., link cost = amount of carried traffic



initially … recompute routing … recompute … recompute

Distance Vector Routing Algorithm

iterative:

- continues until no nodes exchange info.
- *self-terminating*: no "signal" to stop

asynchronous:

- nodes need *not* exchange info/iterate in lock step!

distributed:

- each node communicates *only* with directly-attached neighbors

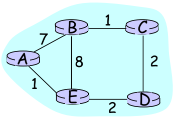
Distance Table data structure

- each node has its own
- row for each possible destination
- column for each directly-attached neighbor to node
- example: in node X, for dest. Y via neighbor Z:

$D^X(Y,Z)$ = distance *from* X *to* Y, *via* Z as next hop

= $c(X,Z) + \min_w\{D^Z(Y,w)\}$

Distance Table: example



$D^E(C,D) = c(E,D) + \min_w\{D^D(C,w)\}$
$= 2+2 = 4$

$D^E(A,D) = c(E,D) + \min_w\{D^D(A,w)\}$
$= 2+3 = 5$ loop!

$D^E(A,B) = c(E,B) + \min_w\{D^B(A,w)\}$
$= 8+6 = 14$ loop!

cost to destination via

$D^E()$	A	B	D
A	1	14	5
B	7	8	5
C	6	9	4
D	4	11	2

destination

Distance table gives routing table

cost to destination via

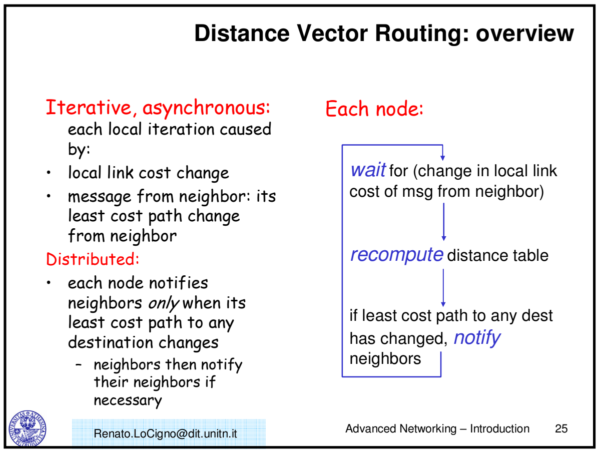
$D^E()$	A	B	D
A	1	14	5
B	7	8	5
C	6	9	4
D	4	11	2

destination

	Outgoing link to use, cost
A	A,1
B	D,5
C	D,4
D	D,4

destination

Distance table → Routing table



Distance Vector Routing: overview

Iterative, asynchronous:
each local iteration caused by:

- local link cost change
- message from neighbor: its least cost path change from neighbor

Distributed:

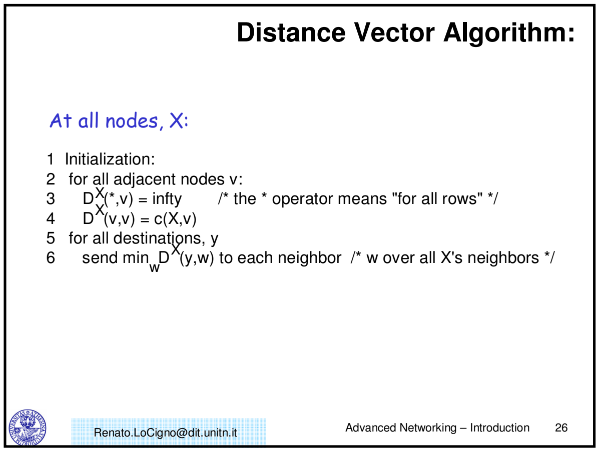
- each node notifies neighbors *only* when its least cost path to any destination changes
  - neighbors then notify their neighbors if necessary

Each node:

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

recompute distance table

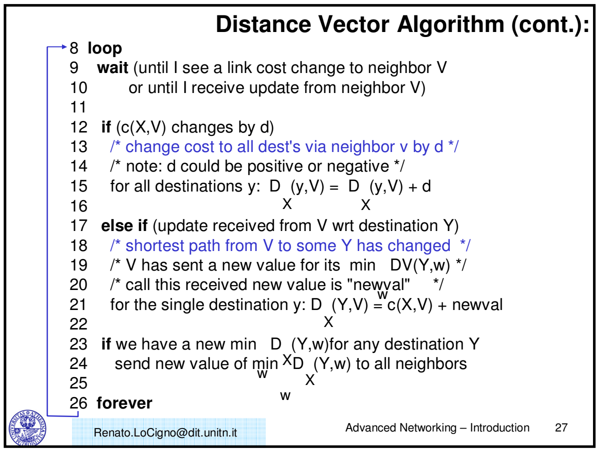
if least cost path to any dest has changed, *notify* neighbors



Distance Vector Algorithm:

At all nodes, X:

```
Initialization:
 for all adjacent nodes v:
    $D^X(*,v)$ = infty        /* the * operator means "for all rows" */
    $D^X(v,v)$ = c(X,v)
 for all destinations, y
    send $\min_w D^X(y,w)$ to each neighbor  /* w over all X's neighbors */
```



Distance Vector Algorithm (cont.):

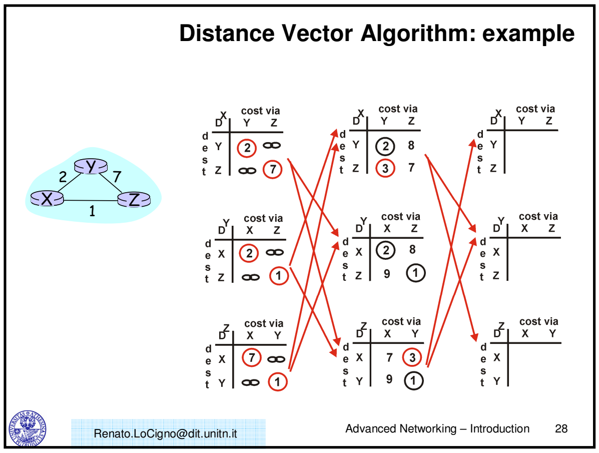
```
loop
  wait (until I see a link cost change to neighbor V
       or until I receive update from neighbor V)

  if (c(X,V) changes by d)
   /* change cost to all dest's via neighbor v by d */
   /* note: d could be positive or negative */
   for all destinations y:  $D_X(y,V) = D_X(y,V) + d$

  else if (update received from V wrt destination Y)
   /* shortest path from V to some Y has changed  */
   /* V has sent a new value for its  $\min_w$ DV(Y,w) */
   /* call this received new value is "newval"     */
   for the single destination y: $D_X(Y,V) = c(X,V) + newval$

  if we have a new $\min_w D_X(Y,w)$ for any destination Y
     send new value of $\min_w D_X(Y,w)$ to all neighbors

 forever
```



Distance Vector Algorithm: example

cost via

D^X	Y	Z
Y	2	∞
Z	∞	7

D^Y	X	Z
X	2	∞
Z	∞	1

D^Z	X	Y
X	7	∞
Y	∞	1

cost via

D^X	Y	Z
Y	2	8
Z	3	7

D^Y	X	Z
X	2	8
Z	9	1

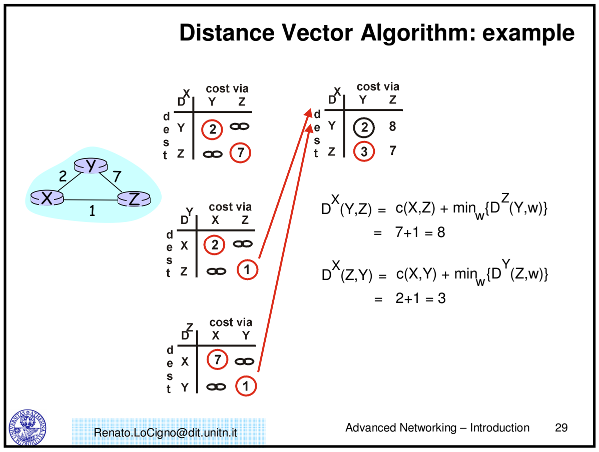
D^Z	X	Y
X	7	3
Y	9	1

cost via

D^X	Y	Z
Y		
Z		

D^Y	X	Z
X		
Z		

D^Z	X	Y
X		
Y		



Distance Vector Algorithm: example

cost via

D^X	Y	Z
Y	2	∞
Z	∞	7

D^Y	X	Z
X	2	∞
Z	∞	1

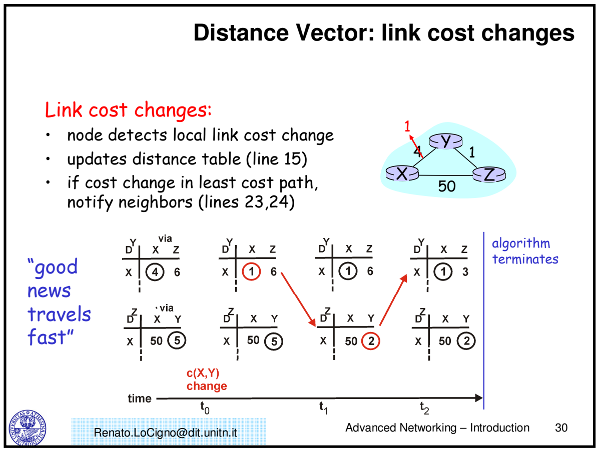
D^Z	X	Y
X	7	∞
Y	∞	1

cost via

D^X	Y	Z
Y	2	8
Z	3	7

$$D^X(Y,Z) = c(X,Z) + \min_w\{D^Z(Y,w)\} = 7+1 = 8$$

$$D^X(Z,Y) = c(X,Y) + \min_w\{D^Y(Z,w)\} = 2+1 = 3$$



Distance Vector: link cost changes

Link cost changes:

- node detects local link cost change
- updates distance table (line 15)
- if cost change in least cost path, notify neighbors (lines 23,24)

via

D^Y	X	Z
X	4	6

D^Z	X	Y
X	50	5

D^Y	X	Z
X	1	6

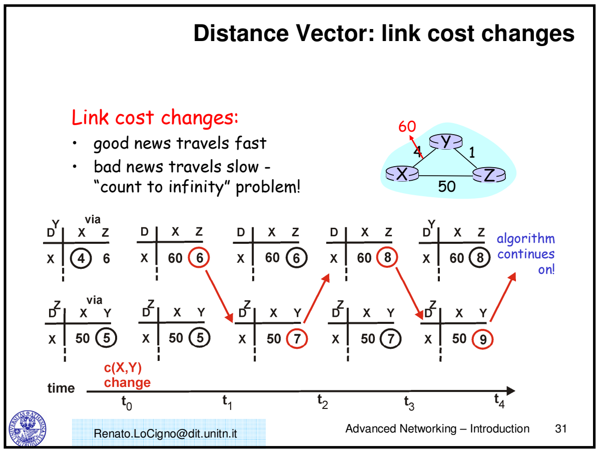
D^Z	X	Y
X	50	5

D^Y	X	Z
X	1	6

D^Z	X	Y
X	50	2

D^Y	X	Z
X	1	3

D^Z	X	Y
X	50	2

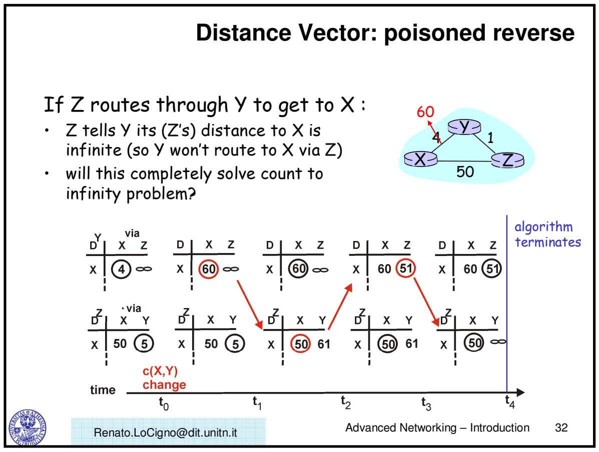


Distance Vector: link cost changes

Link cost changes:

- good news travels fast
- bad news travels slow - "count to infinity" problem!

time	D^Y via X	D^Y via Z	D^Z via X	D^Z via Y
t_0	4	6	50	5
t_1	60	6	50	5
t_2	60	6	50	7
t_3	60	8	50	7
t_4	60	8	50	9

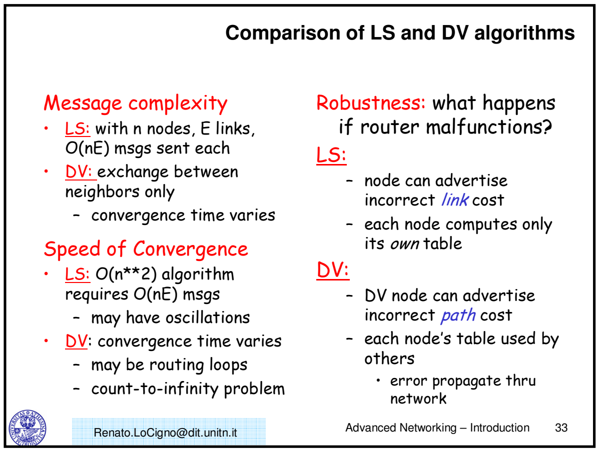


Distance Vector: poisoned reverse

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?

time	D^Y via X	D^Y via Z	D^Z via X	D^Z via Y
t_0	4	∞	50	5
t_1	60	∞	50	5
t_2	60	∞	50	61
t_3	60	51	50	61
t_4	60	51	50	∞



Comparison of LS and DV algorithms

Message complexity

- LS: with n nodes, E links, O(nE) msgs sent each
- 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

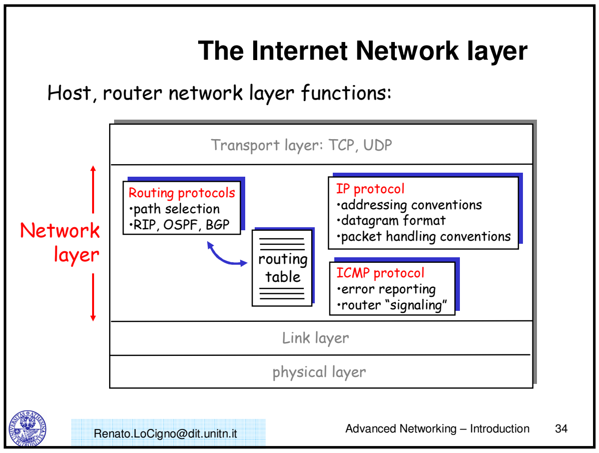
Robustness: what happens if router malfunctions?

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



The Internet Network layer

Host, router network layer functions:

Transport layer: TCP, UDP

Network layer

Routing protocols
•path selection
•RIP, OSPF, BGP

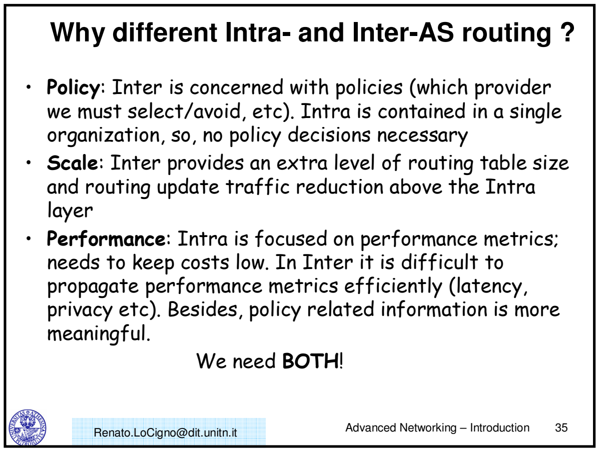
routing table

IP protocol
•addressing conventions
•datagram format
•packet handling conventions

ICMP protocol
•error reporting
•router "signaling"

Link layer

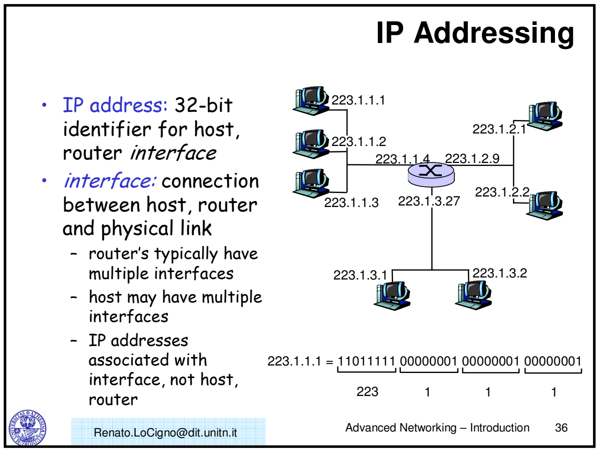
physical layer



Why different Intra- and Inter-AS routing ?

- **Policy**: Inter is concerned with policies (which provider we must select/avoid, etc). Intra is contained in a single organization, so, no policy decisions necessary
- **Scale**: Inter provides an extra level of routing table size and routing update traffic reduction above the Intra layer
- **Performance**: Intra is focused on performance metrics; needs to keep costs low. In Inter it is difficult to propagate performance metrics efficiently (latency, privacy etc). Besides, policy related information is more meaningful.

We need **BOTH**!



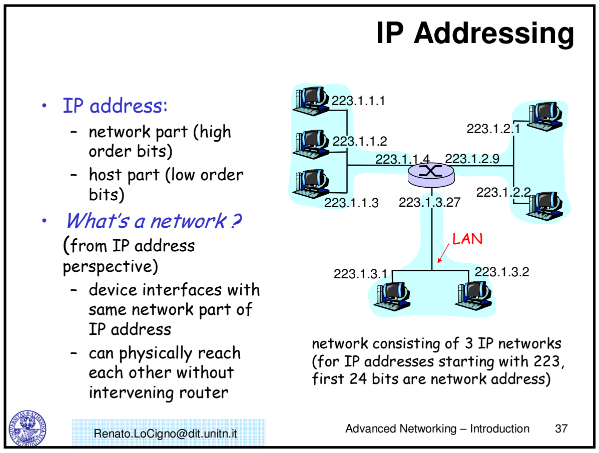
IP Addressing

- IP address: 32-bit identifier for host, router *interface*
- *interface:* connection between host, router and physical link
  - router's typically have multiple interfaces
  - host may have multiple interfaces
  - IP addresses associated with interface, not host, router

223.1.1.1
223.1.1.2
223.1.1.4
223.1.2.9
223.1.2.1
223.1.2.2
223.1.1.3
223.1.3.27
223.1.3.1
223.1.3.2

223.1.1.1 = 11011111 00000001 00000001 00000001

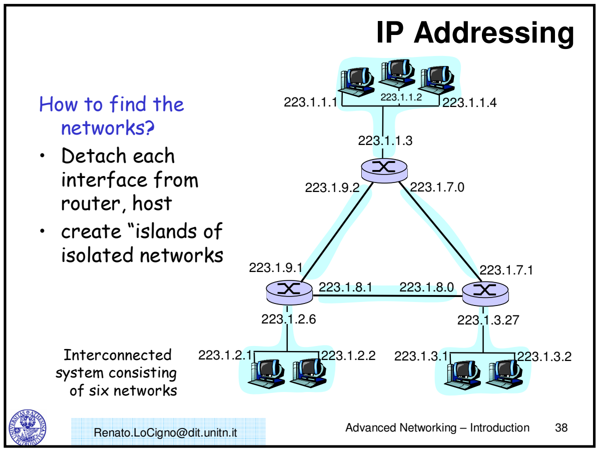
223 1 1 1



IP Addressing

- IP address:
  - network part (high order bits)
  - host part (low order bits)
- *What's a network ?* (from IP address perspective)
  - device interfaces with same network part of IP address
  - can physically reach each other without intervening router

network consisting of 3 IP networks (for IP addresses starting with 223, first 24 bits are network address)

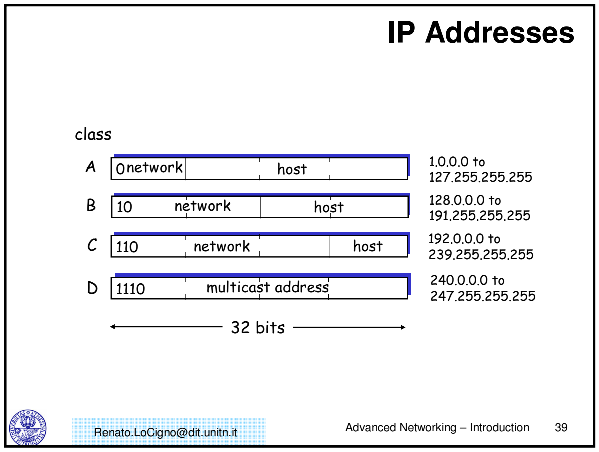


IP Addressing

How to find the networks?

- Detach each interface from router, host
- create "islands of isolated networks

Interconnected system consisting of six networks



IP Addresses

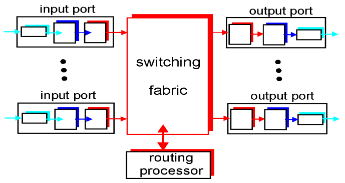
class				
A	0	network	host	1.0.0.0 to 127.255.255.255
B	10	network	host	128.0.0.0 to 191.255.255.255
C	110	network	host	192.0.0.0 to 239.255.255.255
D	1110	multicast address		240.0.0.0 to 247.255.255.255

Address Management

- As Internet grows, we run out of addresses
- Solution (a): **subnetting**. Eg, Class B Host field (16bits) is subdivided into <subnet;host> fields
- Solution (b): **CIDR** (Classless Inter Domain Routing): assign block of contiguous Class C addresses to the same organization; these addresses all share a common prefix
- repeated "aggregation" within same provider leads to shorter and shorter prefixes
- CIDR helps also routing table size and processing: Border Gwys keep only prefixes and find "longest prefix" match

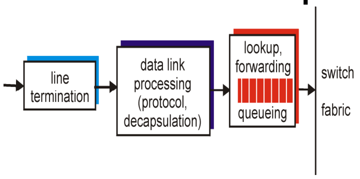
Router Architecture Overview

- Router main functions: *routing* algorithms and protocols processing, *switching* datagrams from an incoming link to an outgoing link



Router Components

Input Ports

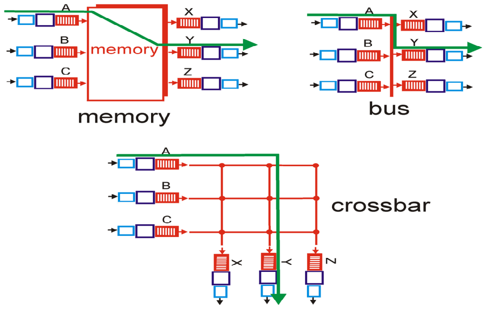


- **Decentralized switching***:* perform routing table lookup using a copy of the node routing table stored in the port memory
- Goal is to complete input port processing at 'line speed', ie processing time =< frame reception time (eg, with 2.5 Gbps line, 256 bytes long frame, router must perform about 1 million routing table lookups in a second)
- Queuing occurs if datagrams arrive at rate higher than can be forwarded on switching fabric

Speeding Up Routing Table Lookup

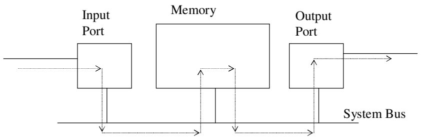
- Table is stored in a tree structure to facilitate binary search
- Content Addressable Memory (associative memory), eg Cisco 8500 series routers
- Caching of recently looked-up addresses
- Compression of routing tables

Switching Fabric



Switching Via Memory

- *First generation routers*: packet is copied under system's (single) CPU control; speed limited by Memory bandwidth. For Memory speed of B packet/sec or pps, throughput is B/2 pps



- *Modern routers*: input ports with CPUs that implement output port lookup, and store packets in appropriate locations (= switch) in a shared Memory; eg Cisco Catalyst 8500 switches

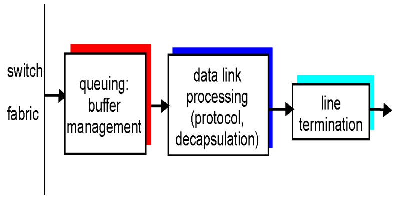
Switching Via Bus

- Input port processors transfer a datagram from input port memory to output port memory via a shared bus
- Main resource contention is over the bus; switching is limited by bus speed
- Sufficient speed for access and enterprise routers (not regional or backbone routers) is provided by a Gbps bus; eg Cisco 1900 which has a 1 Gbps bus

Switching Via An Interconnection Network

- Used to overcome bus bandwidth limitations
- Banyan networks and other interconnection networks were initially developed to connect processors in a multiprocessor computer system; used in Cisco 12000 switches provide up to 60 Gbps through the interconnection network
- Advanced design incorporates fragmenting a datagram into fixed length cells and switch the cells through the fabric; + better sharing of the switching fabric resulting in higher switching speed

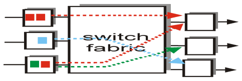
Output Ports


switch
fabric
queuing: buffer management
data link processing (protocol, decapsulation)
line termination

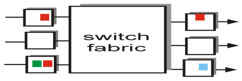
Buffering is required to hold datagrams whenever they arrive from the switching fabric at a rate faster than the transmission rate

Queuing At Input and Output Ports

- Queues build up whenever there is a rate mismatch or blocking. Consider the following scenarios:
  - Fabric speed is faster than all input ports combined; more datagrams are destined to an output port than other output ports; queuing occurs at output port
  - Fabric bandwidth is not as fast as all input ports combined; queuing may occur at input queues;
  - HOL blocking: fabric can deliver datagrams from input ports in parallel, except if datagrams are destined to same output port; in this case datagrams are queued at input queues; there may be queued datagrams that are held behind HOL conflict, even when their output port is available



output port contention at time t - only one red packet can be transferred



green packet experiences HOL blocking

Transport Layer: UDP & TCP

Goals:

- Recall principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet

Overview:

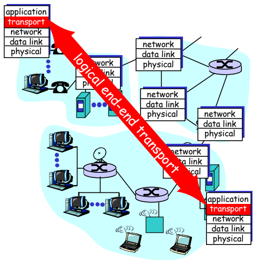
- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management

Transport services and protocols

- provide *logical communication* between app' processes running on different hosts
- transport protocols run in end systems (primarily)

transport vs network layer services:

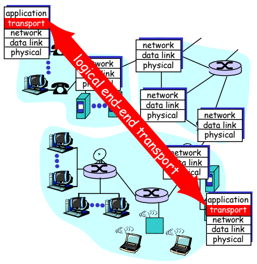
- *network layer:* data transfer between end systems
- *transport layer:* data transfer between processes
  - relies on, enhances, network layer services



Transport-layer protocols

Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast

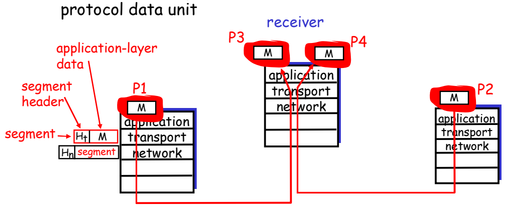


Multiplexing/demultiplexing

Recall: *segment* - unit of data exchanged between transport layer entities

- aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments (TPDUs)to correct app layer processes

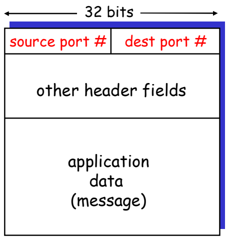


Multiplexing/demultiplexing

Multiplexing: gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

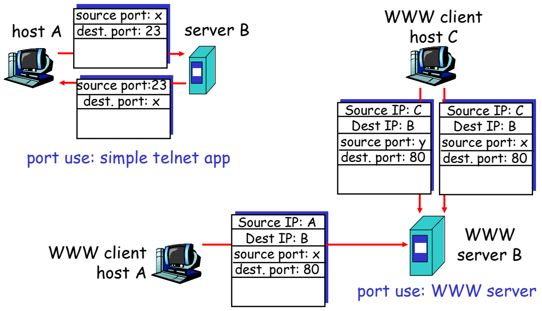
multiplexing/demultiplexing:

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications



TCP/UDP segment format

Multiplexing/demultiplexing: examples



UDP: User Datagram Protocol [RFC 768]

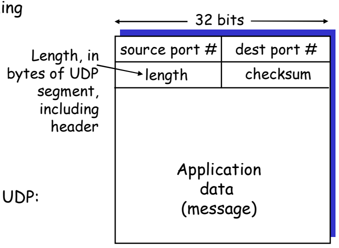
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

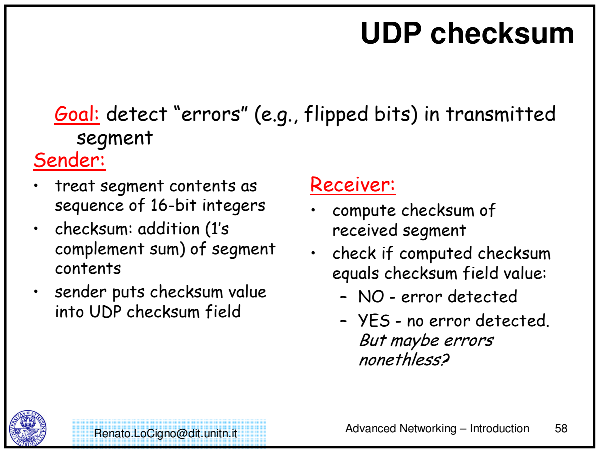
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses (why?):
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recover!


32 bits
source port #
dest port #
Length, in bytes of UDP segment, including header
length
checksum
Application data (message)

UDP segment format



UDP checksum

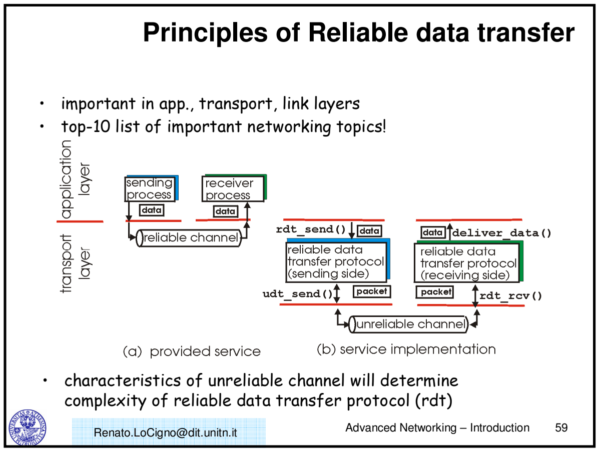
Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonethless?*



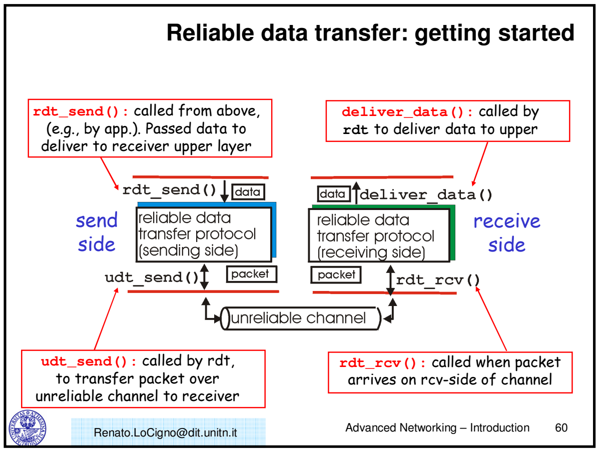
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

(a) provided service

(b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by rdt to deliver data to upper

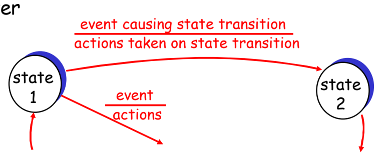
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver


event causing state transition
actions taken on state transition
state 1
state 2
event
actions

state: when in this "state" next state uniquely determined by next event

rdt: channels with errors *and* loss

Sssumption: underlying channel can lose packets (data or ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

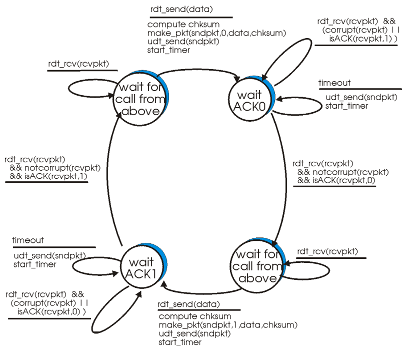
Q: how to deal with loss?

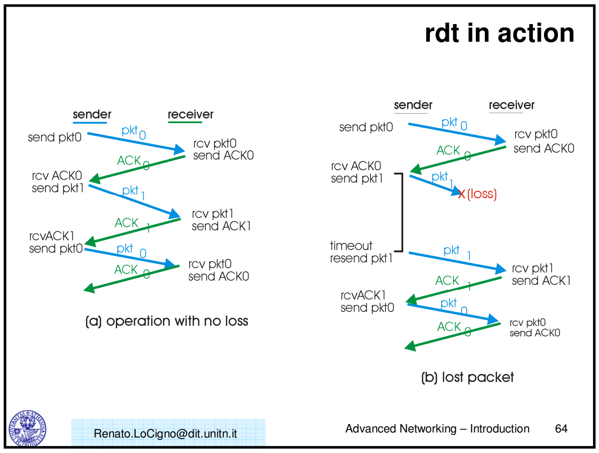
- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

Approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt: sender

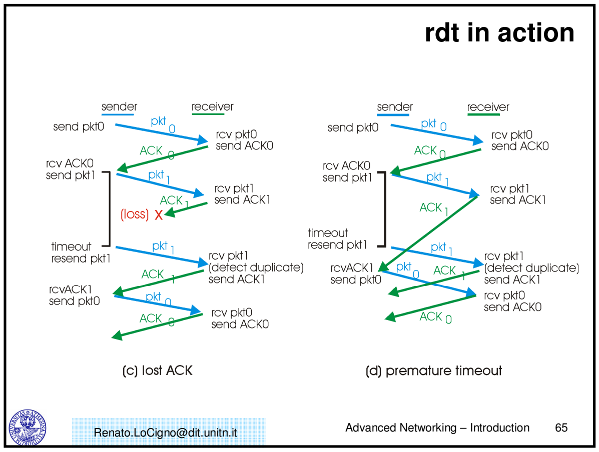

rdt_send(data)
compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) | | isACK(rcvpkt,1))
rdt_rcv(rcvpkt)
wait for call from above
wait ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
timeout
udt_send(sndpkt)
start_timer
wait ACK1
wait for call from above
rdt_rcv(rcvpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) | | isACK(rcvpkt,0))
rdt_send(data)
compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)
start_timer



rdt in action

(a) operation with no loss

(b) lost packet



rdt in action

(c) lost ACK

(d) premature timeout

Performance of rdt

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{8kb/pkt}{10^{**}9\ b/sec} = 8 \text{ microsec}$$

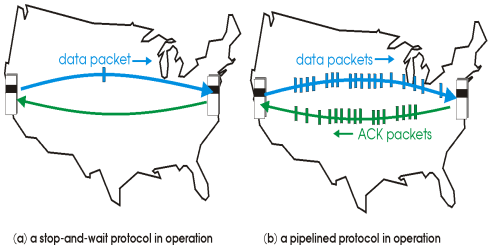
$$\text{Utilization} = U = \frac{\text{fraction of time}}{\text{sender busy sending}} = \frac{8 \text{ microsec}}{30.016 \text{ msec}} = 0.00015$$

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

Pipelined Protocols

- Channel utilization under a Stop&Wait protocol is not high when the propagation time is long relative to the transmission time
- Solution: pipelined protocols, where more than one packet can be sent without waiting for feedback, thus filling the 'pipeline'
- Two major versions (and lots of variations on the theme):
  - Go-Back-N
  - Selective Repeat
- New requirements:
  - Buffering more than one packet at sender, and possibly at receiver too
  - Larger sequence numbers for identifying packets in transit

Filling the Pipeline



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Stop&Wait Efficiency

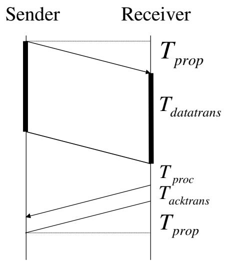
$$U = \frac{T_{datatrans}}{T_{datatrans} + 2*T_{prop} + T_{proc} + T_{acktrans}}$$

For relatively small T_{proc} and $T_{acktrans}$

$$U \approx \frac{T_{datatrans}}{T_{datatrans} + 2*T_{prop}}, \text{ or}$$

$$U \approx \frac{1}{1+2*a}, \text{ where } a = \frac{T_{prop}}{T_{datatrans}}$$

$T_{datatrans} = \frac{L}{C}$, where L is the Packet length and C is the Transmission Speed. For one bit of data, $T_{datatrans} = 1/C$; in this case $a = CT_{prop}$, which is called the "Bandwidth - Delay" product

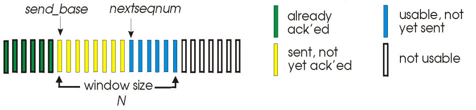


Go-Back-N

- Sender can go ahead and transmit packets without waiting for feedback up to some number of packets (for flow control reasons, details later)
- Definitions:

  N: maximum allowable number of transmission without feedback

  Base: lowest sequence number of unacked packets



Go-Back-N Window

- From definitions and figure above:

  [0, base-1] transmitted and acked

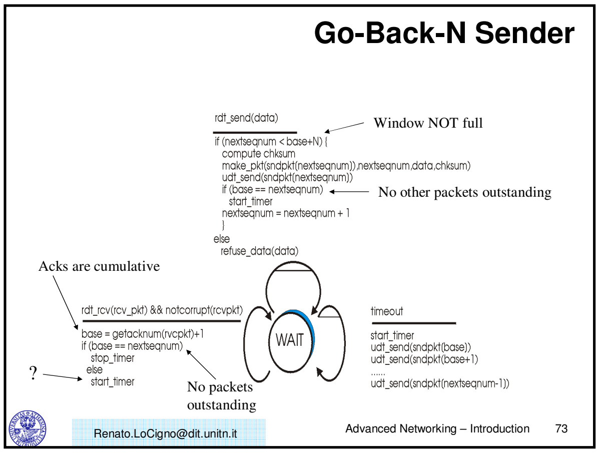
  [base, nextseqnum-1] transmitted and waiting for feedback, or 'outstanding'

  [nextseqnum, base+N-1] numbers that can be used when packets are provided by higher layer for transmission

  [base+N, maxseqnum] numbers that cannot be used until more packets are acked

Go-Back-N Window (Cont.)

- Because of the window metaphor, these protocols are also referred to as *sliding window protocols*
- Stop&Wait can be viewed as a sliding window protocol, with window size N = 1, and sequence space = [0,1]
- Sequence number is carried in a fixed length field in the packet header; with k bits in the Sequence number field, the sequence space is
- Since sequence numbers must wrap around, all sequence number arithmetic is modulo



Go-Back-N Sender

rdt_send(data)

```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
     start_timer
   nextseqnum = nextseqnum + 1
   }
else
   refuse_data(data)
```

Window NOT full

No other packets outstanding

Acks are cumulative

rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

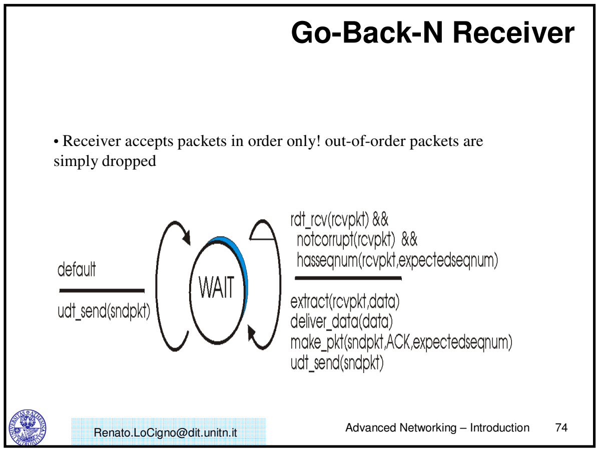
```
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
   stop_timer
 else
   start_timer
```

?

No packets outstanding

timeout

```
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
......
udt_send(sndpkt(nextseqnum-1))
```



Go-Back-N Receiver

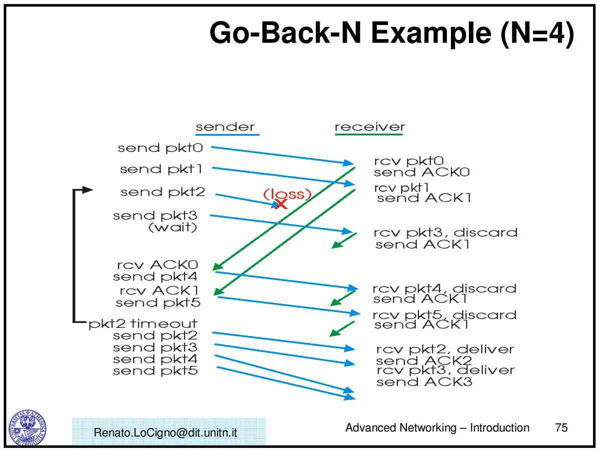
- Receiver accepts packets in order only! out-of-order packets are simply dropped

default

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
 notcorrupt(rcvpkt) &&
 hasseqnum(rcvpkt,expectedseqnum)

```
extract(rcvpkt,data)
deliver_data(data)
make_pkt(sndpkt,ACK,expectedseqnum)
udt_send(sndpkt)
```



Go-Back-N Example (N=4)

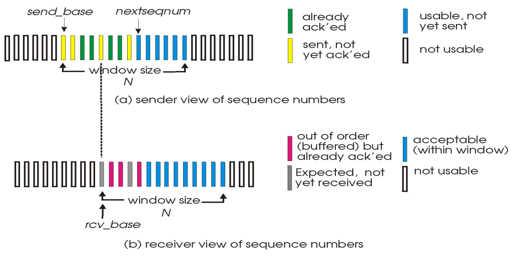
Go-Back-N Performance

- *Bandwidth-Delay Product* (ie “pipeline size”) is defined as the product of the channel transmission speed and the propagation delay
- As transmission speed or propagation delay increases, more packets can be transmitted to “fill the pipeline”
- For channels with high Bandwidth-Delay product, Go-Back-N performance may deteriorate: the number of outstanding packets may be large and all these packets will be unnecessarily retransmitted when an error occurs

Selective Repeat

- Selective Repeat addresses the performance limitation of Go-back-N mentioned above
- Receiver indicates to sender which packet needs to be retransmitted; sender retransmits only that packet
- Receiver accepts and buffers packets received out of order within a limit imposed by a *receiver window*
- Groups of packets with consecutive sequence numbers (or completed sequences) are delivered to the higher layer at the sender
- A timer must be associated with each packet (but we can use one hardware timer to implement multiple logical timers)

Selective Repeat Windows


send_base
nextseqnum
window size
N
(a) sender view of sequence numbers
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable
window size
N
rcv_base
(b) receiver view of sequence numbers

Selective Repeat Sender Event-Driven Algorithms

- Higher layer calls to transmit data:

```
if there are unused sequence numbers
    then packetize and transmit;
    else reject the data;
```

- Timeout occurs:

```
transmit the (single) packet which timed out;
```

- Ack is received:

```
mark packet acked;
if base can be moved
    then move it to the unacked packet  with the lowest sequence
      number;
```

Selective Repeat Receiver Event-Driven Algorithms

- Packet received, not corrupted, within current receive window:

```
Ack the received packet;
if not previously received
    then buffer the packet;
deliver consectively sequenced received packets to higher layer;
move window forward;
```

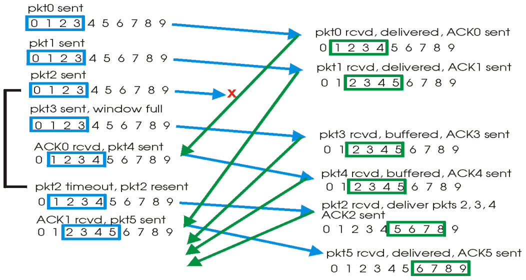
- Packet received, not corrupt, sequence number below window base:

```
Ack the received packet; /* packet previously acked and already
    delivered to higher layer*/
```

- Packet received, corrupt, or sequence number beyond window:

```
Ignore the packet
```

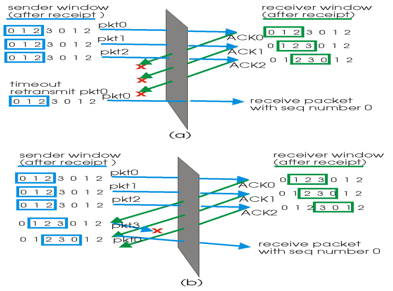
Selective Repeat Example



Setting The Window Size

- The window size N is an important parameter
- N should be large enough to allow filling the pipeline, thus making better utilization of the channel
- On the other hand, N is limited by the protocols (ensure receiver correctly identifies packets)
- It was found that N cannot be larger than half the sequence space length

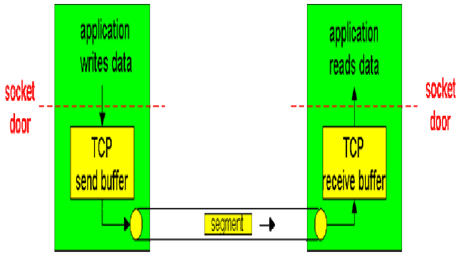
Misidentification Example



Reliable Transport Layer: TCP

- Full-duplex
- End-to-end protocol, transparent to network and lower layers in routers
- Connection-oriented, connection established through "three way handshake" protocol
- Byte Stream transfer, stream is divided into segments with a *maximum segment size* (MSS)
- Reliability through an ARQ type protocol
- Flow Control: receiver controls the amount of bytes a sender is allowed to send
- Point-to-point connection, no multicasting with TCP

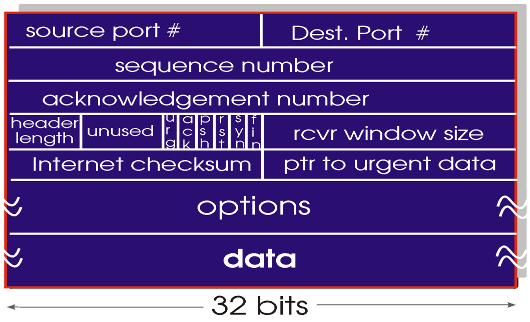
TCP Connection Model



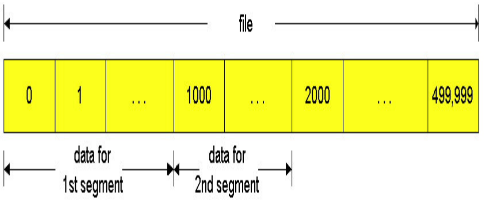
Segment Format

- Header contains:
  - Source and Destination Ports
  - Segment Sequence Number: that of the first byte in the segment (Byte Stream model)
  - Acknowledgment Number: sequence number of byte expected from the other side next
  - Header length: header as a fixed part of 20 bytes + optional fields
  - Receiver Window Size: the maximum number of bytes that the other side is allowed to send next
  - Header checksum: to ensure correctness of header field
  - Flags
  - 4 unused bits!

Segment Format (Cont.)



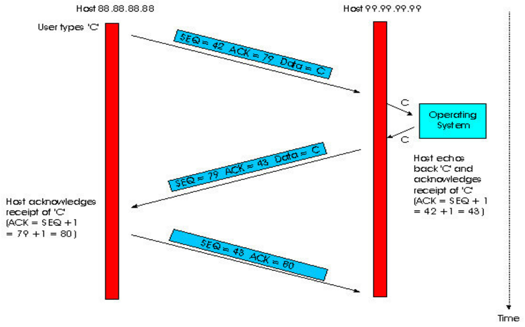
Segmented Byte Stream

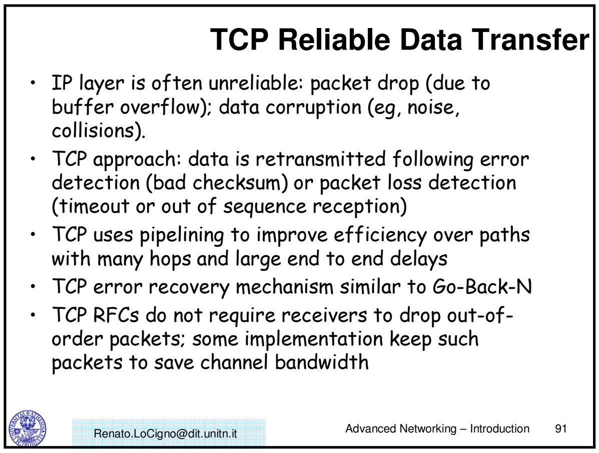


Telnet: A TCP ACK example

- Telnet: appl. level protocol for remote login
- Interactive mode; typed characters are "echoed back" by remote Host (each character traverses the network twice)
- Full duplex stream of characters provides opportunity for ACK piggybacking
- In simplex (one way) data transfer, explicit ACKs are required

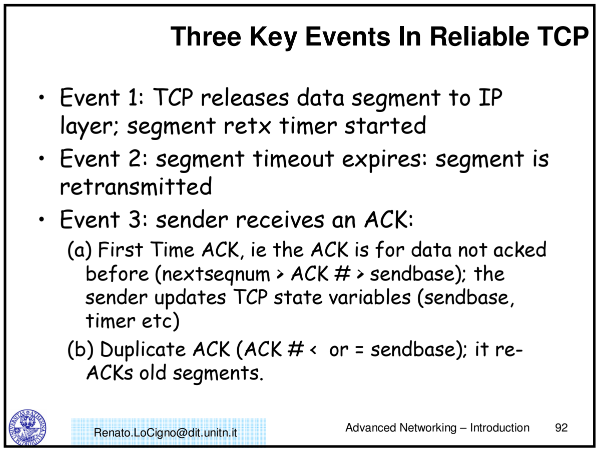
Telnet: ACK Example


Host 88.88.88.88
Host 99.99.99.99
User types 'C'
SEQ = 42 ACK = 79 Data = C
C
Operating System
C
Host echos back 'C' and acknowledges receipt of 'C' (ACK = SEQ + 1 = 42 +1 = 43)
SEQ = 79 ACK = 43 Data = C
Host acknowledges receipt of 'C' (ACK = SEQ +1 = 79 +1 = 80)
SEQ = 43 ACK = 80
Time



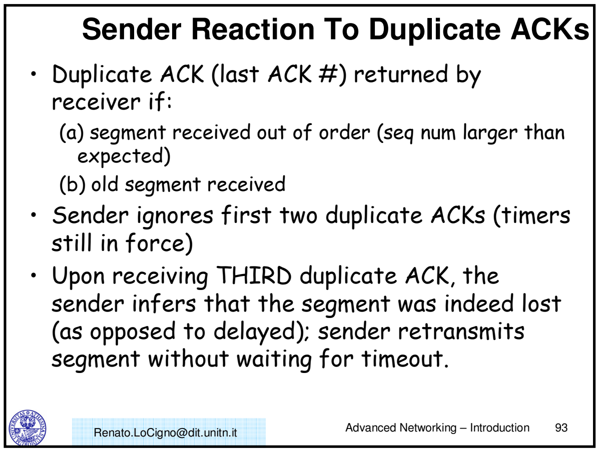
TCP Reliable Data Transfer

- IP layer is often unreliable: packet drop (due to buffer overflow); data corruption (eg, noise, collisions).
- TCP approach: data is retransmitted following error detection (bad checksum) or packet loss detection (timeout or out of sequence reception)
- TCP uses pipelining to improve efficiency over paths with many hops and large end to end delays
- TCP error recovery mechanism similar to Go-Back-N
- TCP RFCs do not require receivers to drop out-of-order packets; some implementation keep such packets to save channel bandwidth



Three Key Events In Reliable TCP

- Event 1: TCP releases data segment to IP layer; segment retx timer started
- Event 2: segment timeout expires: segment is retransmitted
- Event 3: sender receives an ACK:
  - (a) First Time ACK, ie the ACK is for data not acked before (nextseqnum > ACK # > sendbase); the sender updates TCP state variables (sendbase, timer etc)
  - (b) Duplicate ACK (ACK # < or = sendbase); it re-ACKs old segments.



Sender Reaction To Duplicate ACKs

- Duplicate ACK (last ACK #) returned by receiver if:
  - (a) segment received out of order (seq num larger than expected)
  - (b) old segment received
- Sender ignores first two duplicate ACKs (timers still in force)
- Upon receiving THIRD duplicate ACK, the sender infers that the segment was indeed lost (as opposed to delayed); sender retransmits segment without waiting for timeout.

TCP Receiver ACK Generation

EVENT:

Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged. No gaps in the received data.

ACTION:

Delayed ACK. Wait up to 500 ms for arrival of another in-order segment. If next in-order segment does not arrives in this interval, send an ACK

EVENT:

Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission. No gaps in the received data.

ACTION:

Immediately send single cumulative ACK, ACKing both in-order segments

TCP Receiver ACK Generation (Cont.)

EVENT:

Arrival of out-of-order segment with higher-than expected sequence number. Gap detected.

ACTION:

Immediately send duplicate ACK, indicating sequence number of next expected byte

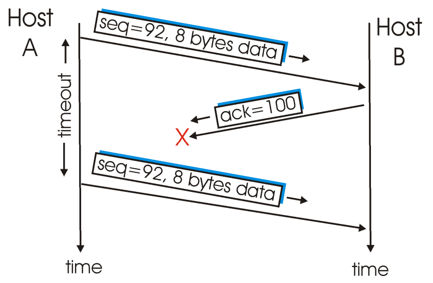
EVENT:

Arrival of segment that partially or completely fills in gap in received data

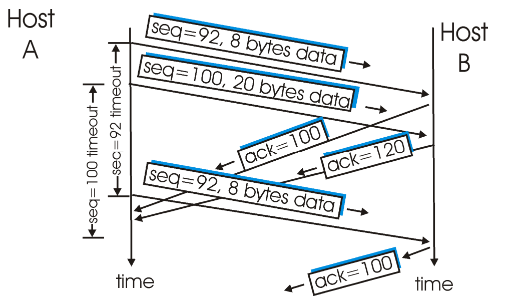
ACTION:

Immediately send ACK, provided that segment starts at the lower end of gap.

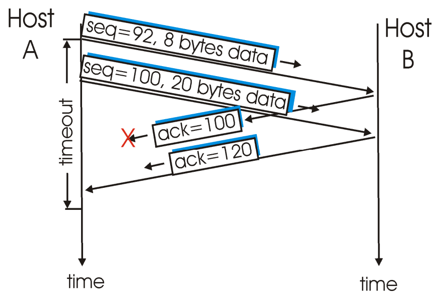
Example: TCP ACK loss



Example: TCP Timeout



Example: TCP ACK Loss



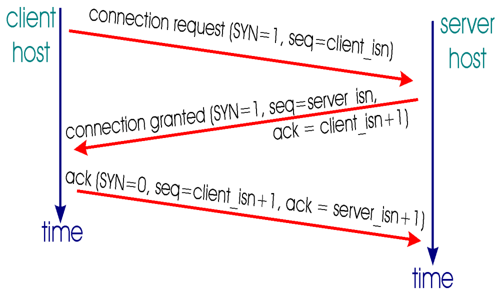
Flow/Congestion Control

- Flow Control (strict definition): regulate TCP flow so as to prevent receive buffer overflow at destination
- Flow Control (more general definition): regulate TCP flow so as to prevent buffer overflow anywhere along the path
- Congestion Control: regulate TCP flow(s) so as to avoid congestion in the entire network and to achieve efficient, fair sharing of resources.
- Key TCP flow/congestion mechanism: adjustable sender window

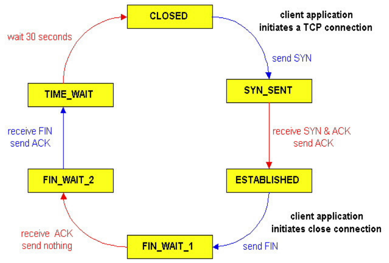
TCP Connection Management

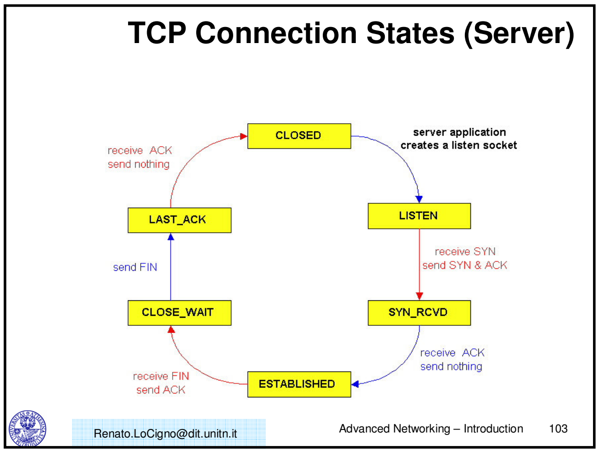
- TCP connection is set up using the *three way handshake* protocol
- Special segments (SYN segment, SYNACK segment) exchange initial client and server sequence numbers and allocate buffers
- Three Way Handshake protocol allows to detect and eliminate "old" connection requests (more robust than two separate handshakes)
- Another Three Way Handshake (with FIN flag turned on) is used to close the connection, releasing all resources

Three Way Handshake


client host
connection request (SYN=1, seq=client_isn)
server host
connection granted (SYN=1, seq=server_isn, ack = client_isn+1)
ack (SYN=0, seq=client_isn+1, ack = server_isn+1)
time
time

TCP Connection States (Client)





TCP Connection States (Server)

CLOSED

server application
creates a listen socket

LISTEN

receive SYN
send SYN & ACK

SYN_RCVD

receive ACK
send nothing

ESTABLISHED

receive FIN
send ACK

CLOSE_WAIT

send FIN

LAST_ACK

receive ACK
send nothing

Renato.LoCigno@dit.unitn.it

Advanced Networking – Introduction 103