

# **COMP9121: Design of Networks and Distributed Systems**

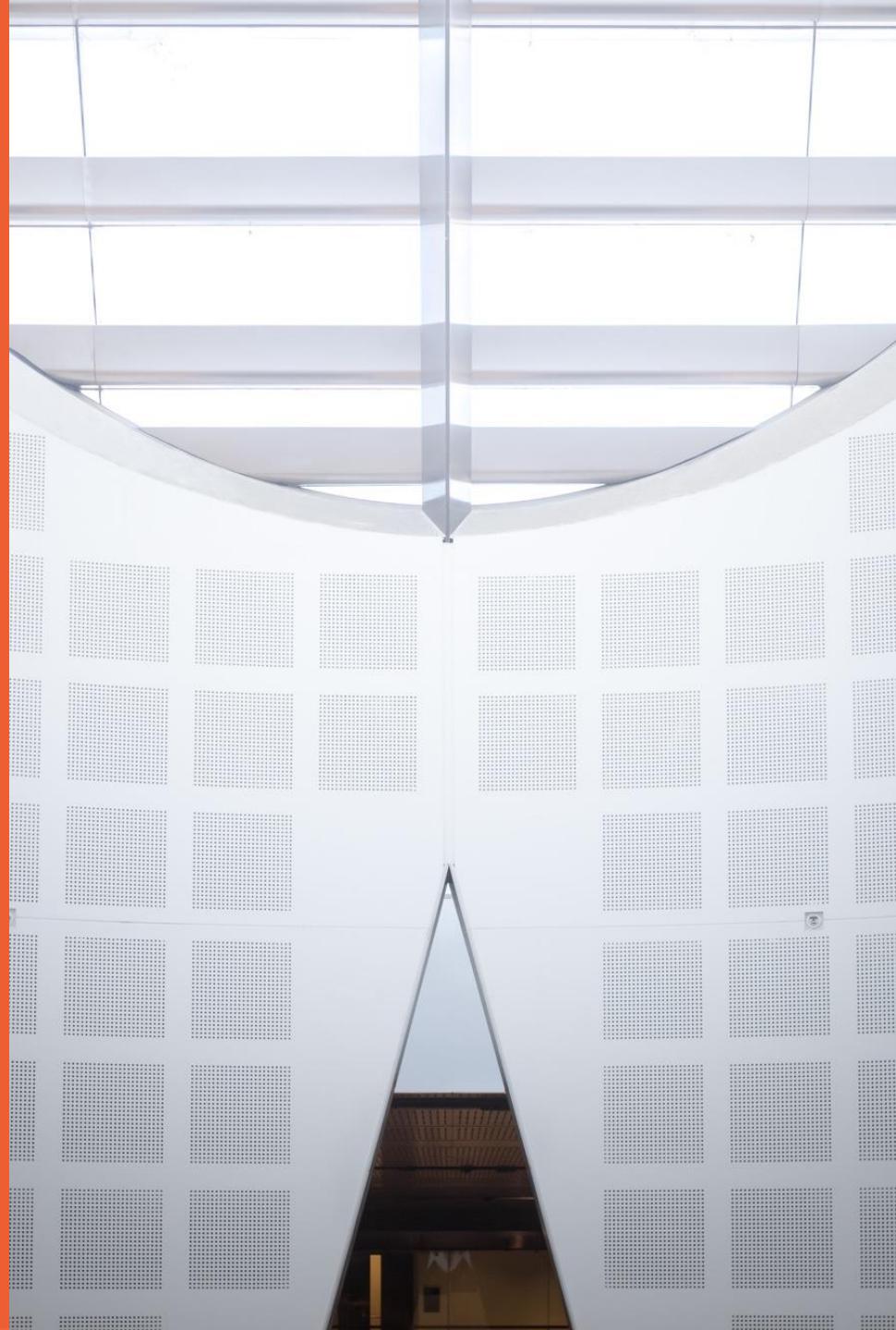
**Week 7: Network Layer 4 and  
Transport Layer 1**

**Wei Bao**

**School of Computer Science**



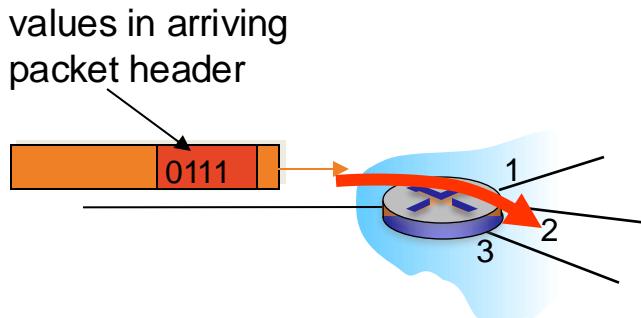
THE UNIVERSITY OF  
**SYDNEY**



## Network layer: data plane, control plane

### Data plane

- local, per-router function
- determines how datagram arriving on router input port is forwarded to router output port
- forwarding function

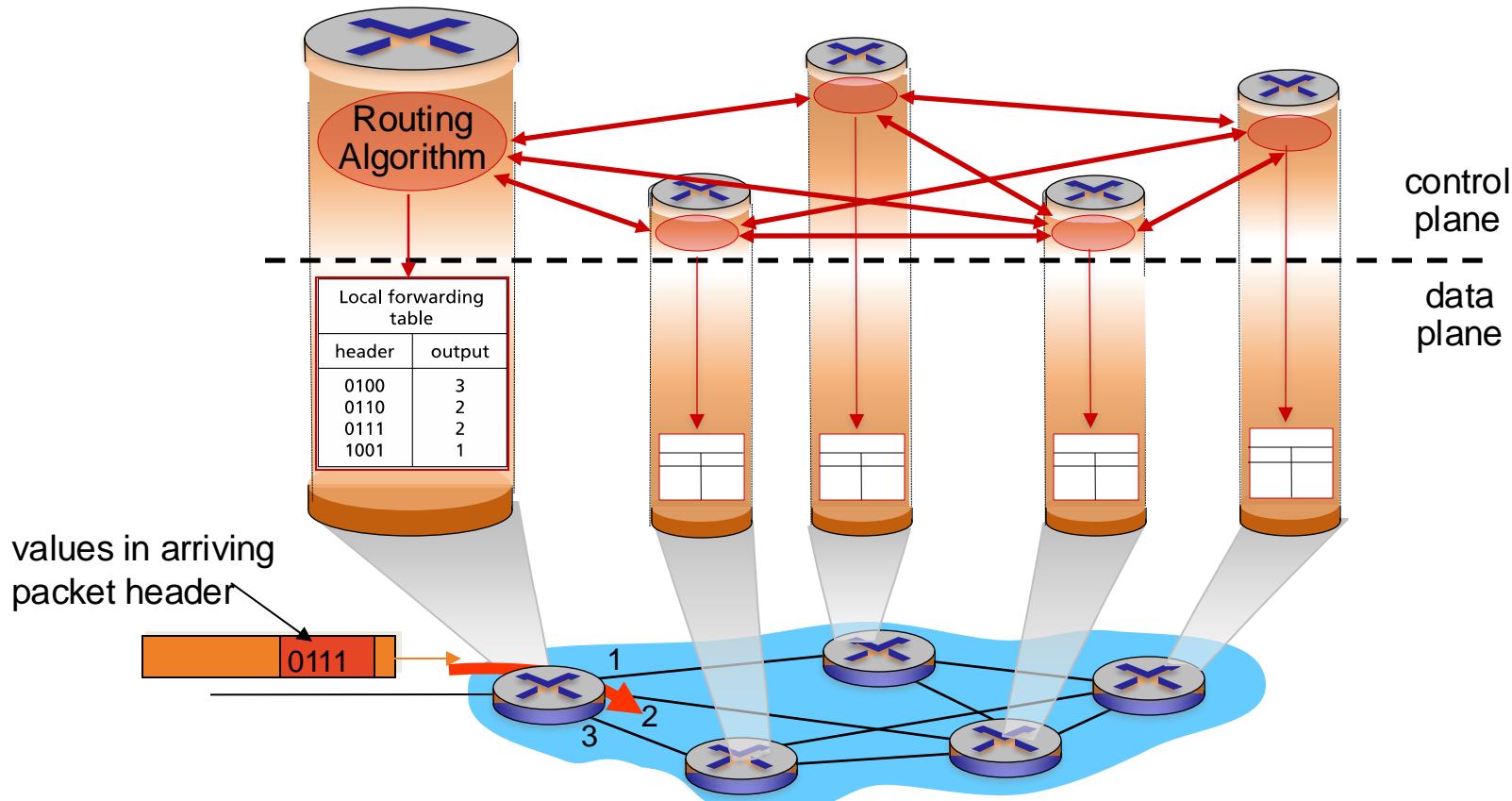


### Control plane

- network-wide logic
- determines how datagram is routed among routers along end-end path from source host to destination host
- two control-plane approaches:
  - *traditional routing algorithms*: implemented in routers
  - *software-defined networking (SDN)*: implemented in (remote) servers

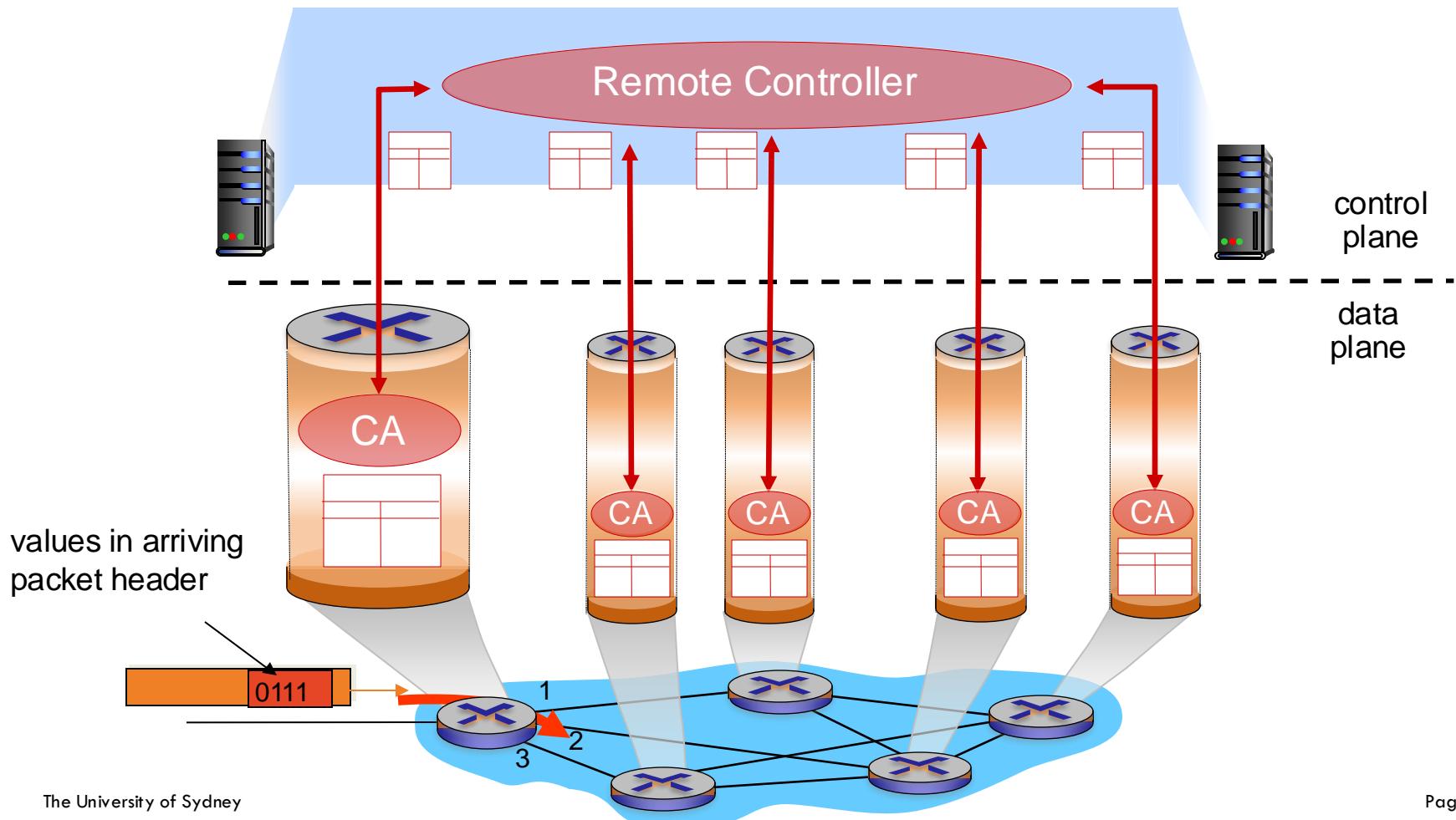
# Traditional: Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



# SDN: Logically centralized control plane

A distinct (typically remote) controller interacts with local control agents (CAs)



每個 router 要怎樣处理 data gram 呢？  
會有 data gram ？

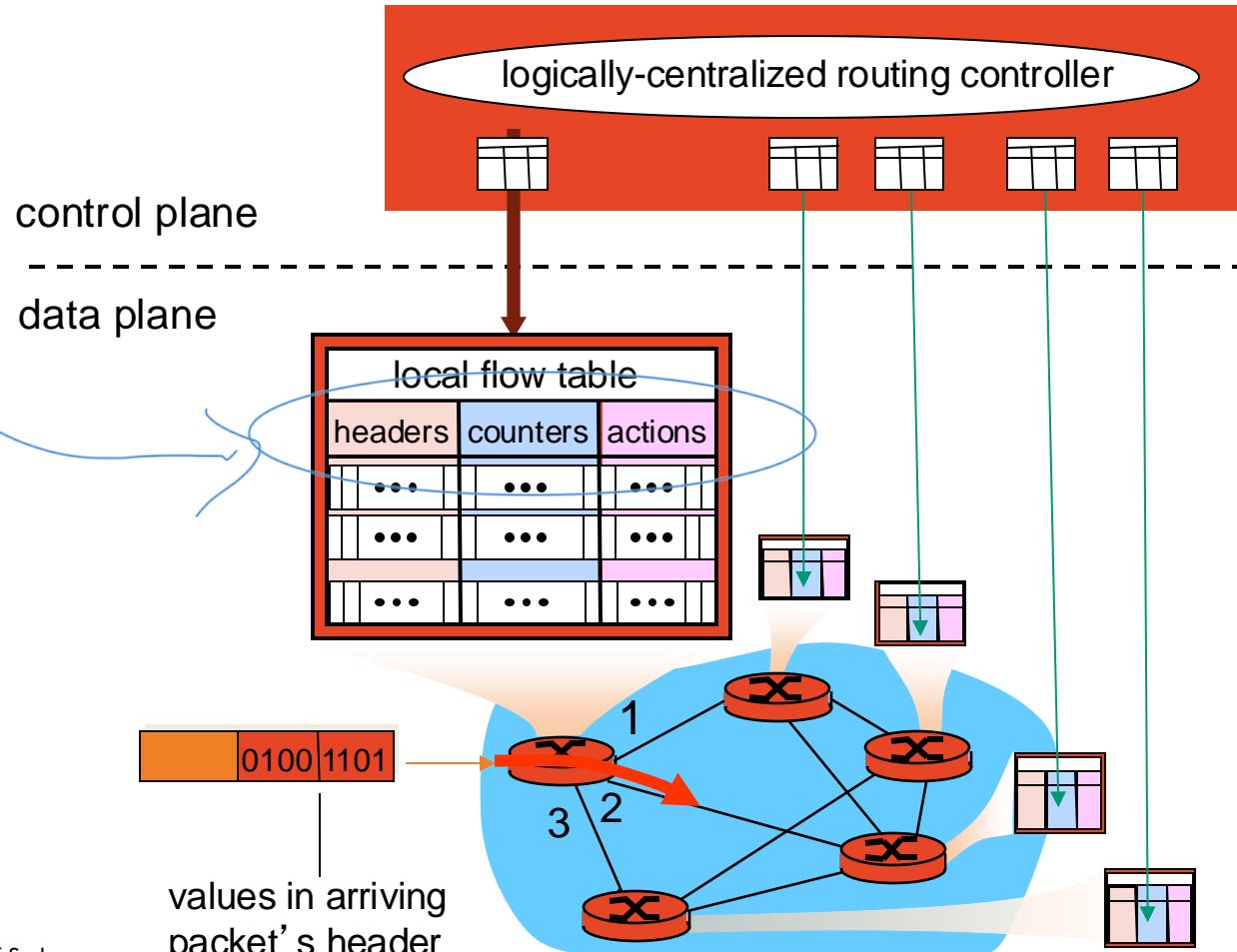
## Software defined networking (SDN)

- Internet network layer: historically has been implemented via distributed, per-router approach
  - *monolithic* router contains switching hardware, runs proprietary implementation of Internet standard protocols (IP, RIP, IS-IS, OSPF, BGP) in proprietary router OS (e.g., Cisco IOS)
  - different “middleboxes” for different network layer functions: firewalls, load balancers, NAT boxes, ..
- ~2005: renewed interest in rethinking network control plane

# **Data Plane**

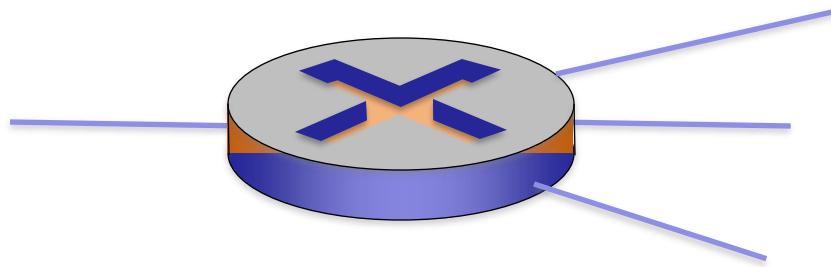
# Data Plane: Generalized Forwarding

Each router contains a **flow table** that is computed and distributed by a *logically centralized* routing controller



# OpenFlow data plane abstraction

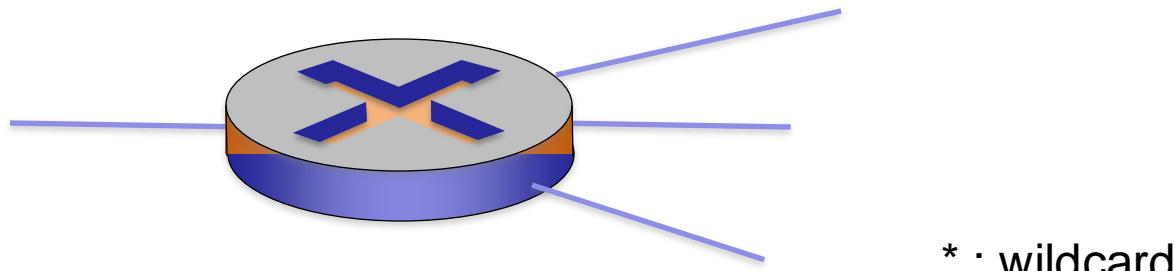
- *flow*: defined by header fields
- generalized forwarding: simple packet-handling rules
  - *Pattern*: match values in packet header fields
  - *Actions*: for matched packet: drop, forward, modify, matched packet or send matched packet to controller
  - *Priority*: disambiguate overlapping patterns
  - *Counters*: #bytes and #packets



*Flow table in a router (computed and distributed by controller) define router's match+action rules*

# OpenFlow data plane abstraction

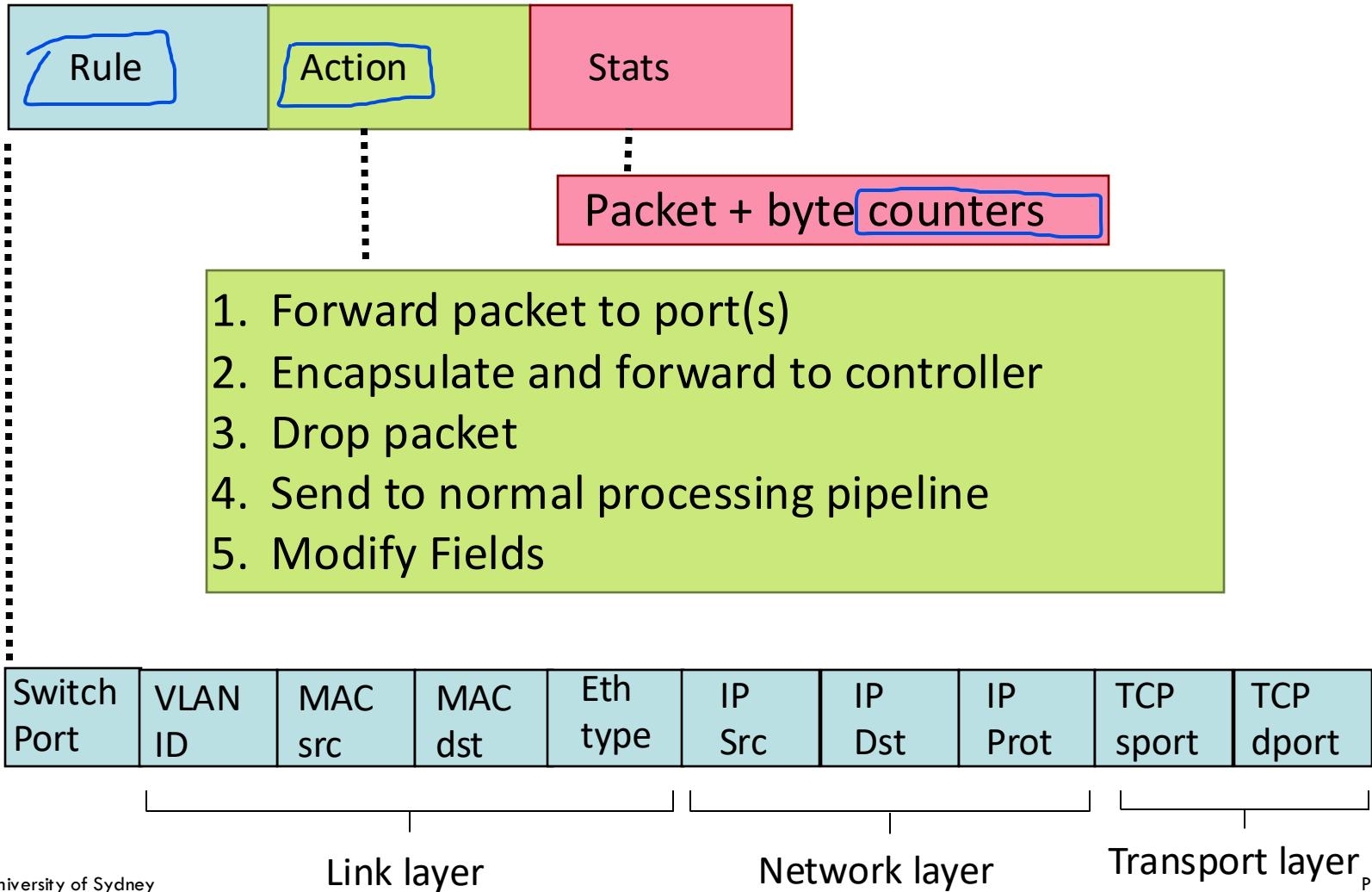
- *flow*: defined by header fields
- generalized forwarding: simple packet-handling rules
  - *Pattern*: match values in packet header fields
  - *Actions*: for matched packet: drop, forward, modify, matched packet or send matched packet to controller
  - *Priority*: disambiguate overlapping patterns
  - *Counters*: #bytes and #packets



\* : wildcard

1.  $\text{src}=1.2.*.*$ ,  $\text{dest}=3.4.5.* \rightarrow \text{drop}$
2.  $\text{src} = *.*.*.*$ ,  $\text{dest}=3.4.*.* \rightarrow \text{forward}(2)$
3.  $\text{src}=10.1.2.3$ ,  $\text{dest} = *.*.*.* \rightarrow \text{send to controller}$

# OpenFlow: Flow Table Entries



# Examples

## Destination-based forwarding:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	*	51.6.0.8	*	*	*	port6

*IP datagrams destined to IP address 51.6.0.8 should be forwarded to router output port 6*

## Firewall:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	*	*	*	*	22	drop

*do not forward (block) all datagrams destined to TCP port 22*

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	128.119.1.1	*	*	*	*	drop

*do not forward (block) all datagrams sent by host 128.119.1.1*

# Examples

## Destination-based layer 2 (switch) forwarding:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	22:A7:23: 11:E1:02	*	*	*	*	*	*	*	*	port3

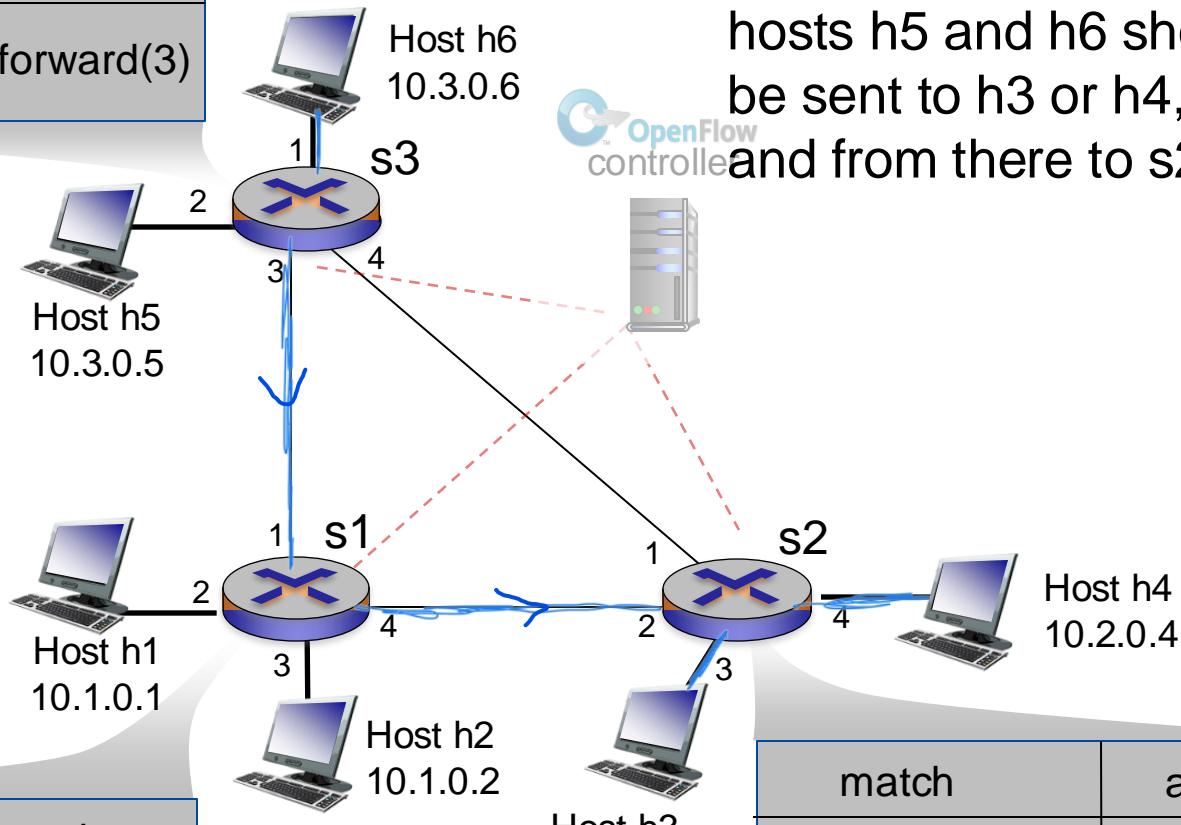
*layer 2 frames from MAC address 22:A7:23:11:E1:02  
should be forwarded to output port 3*

# OpenFlow abstraction

- *match+action*: unifies different kinds of devices
- Router
  - *match*: longest destination IP prefix
  - *action*: forward out a link
- Switch
  - *match*: destination MAC address
  - *action*: forward or flood
- Firewall
  - *match*: IP addresses and TCP/UDP port numbers
  - *action*: permit or deny
- NAT
  - *match*: IP address and port
  - *action*: rewrite address and port

# OpenFlow example

match	action
IP Src = 10.3.*.* IP Dst = 10.2.*.*	forward(3)



**Example:** datagrams from hosts h5 and h6 should be sent to h3 or h4, via s1 and from there to s2

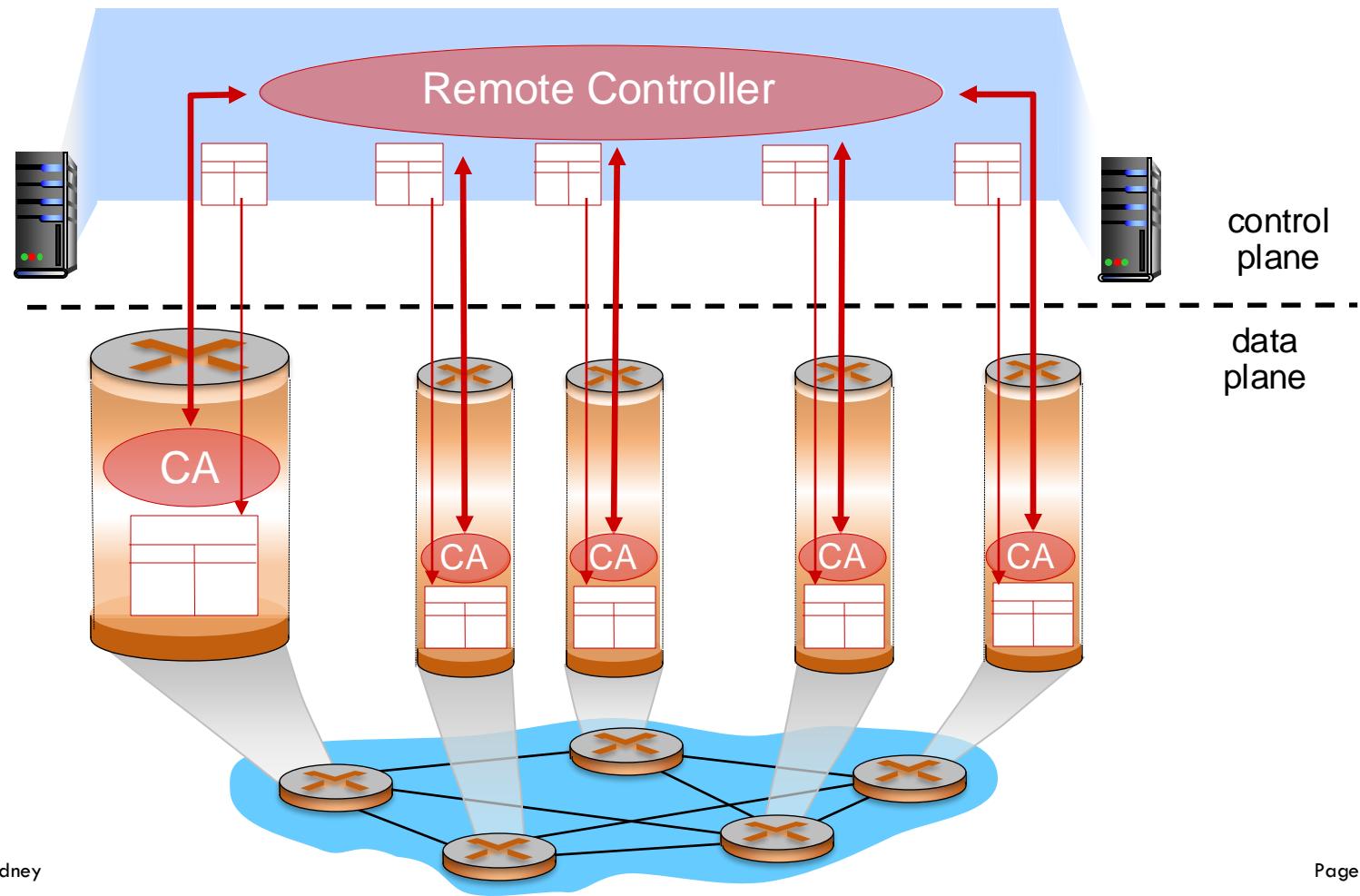
match	action
ingress port = 1 IP Src = 10.3.*.* IP Dst = 10.2.*.*	forward(4)

match	action
ingress port = 2 IP Dst = 10.2.0.3	forward(3)
ingress port = 2 IP Dst = 10.2.0.4	forward(4)

# **Control Plane**

## Recall: logically centralized control plane

A distinct (typically remote) controller interacts with local control agents (CAs) in routers to compute forwarding tables

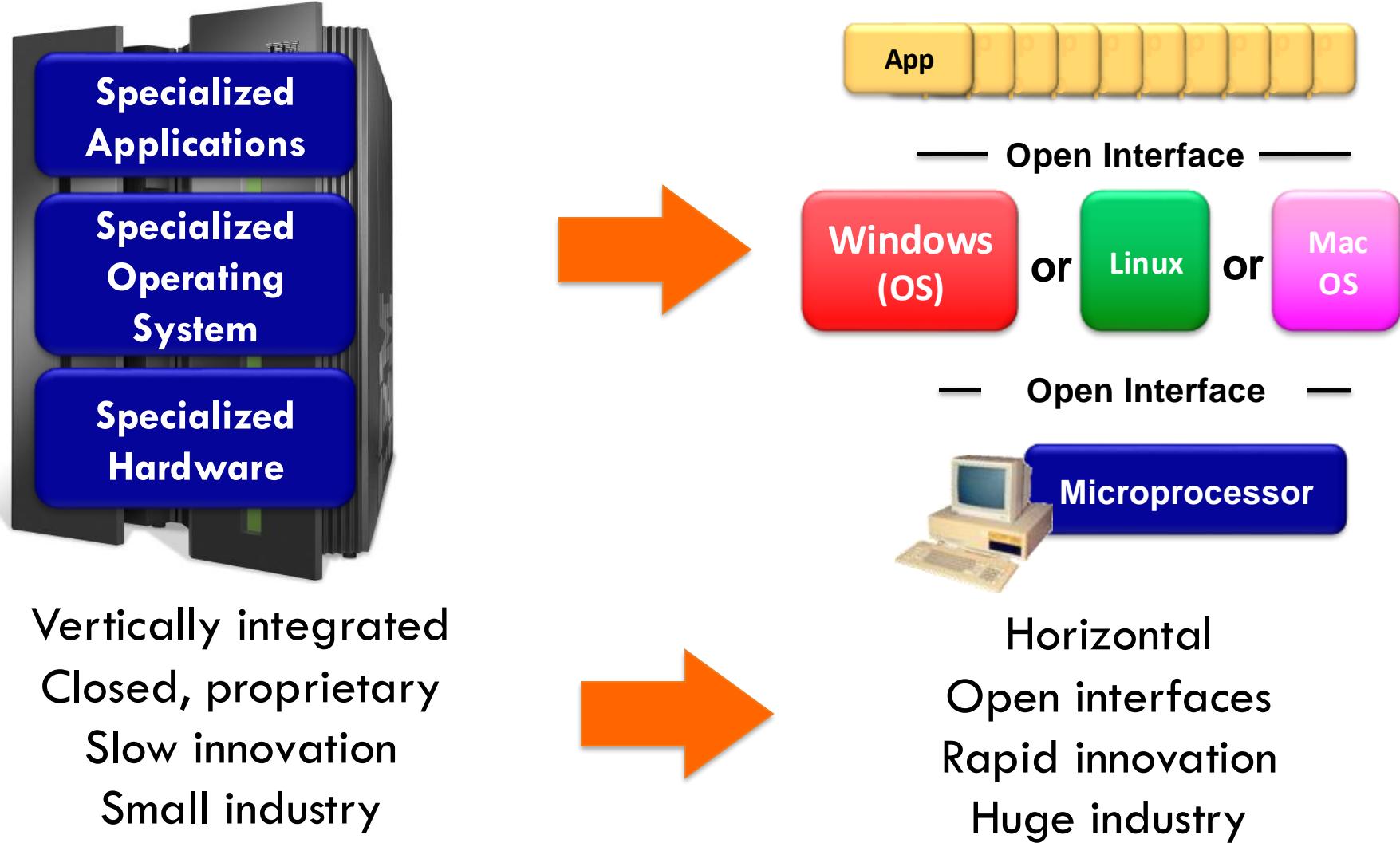


# Software defined networking (SDN)

## *Why a logically centralized control plane?*

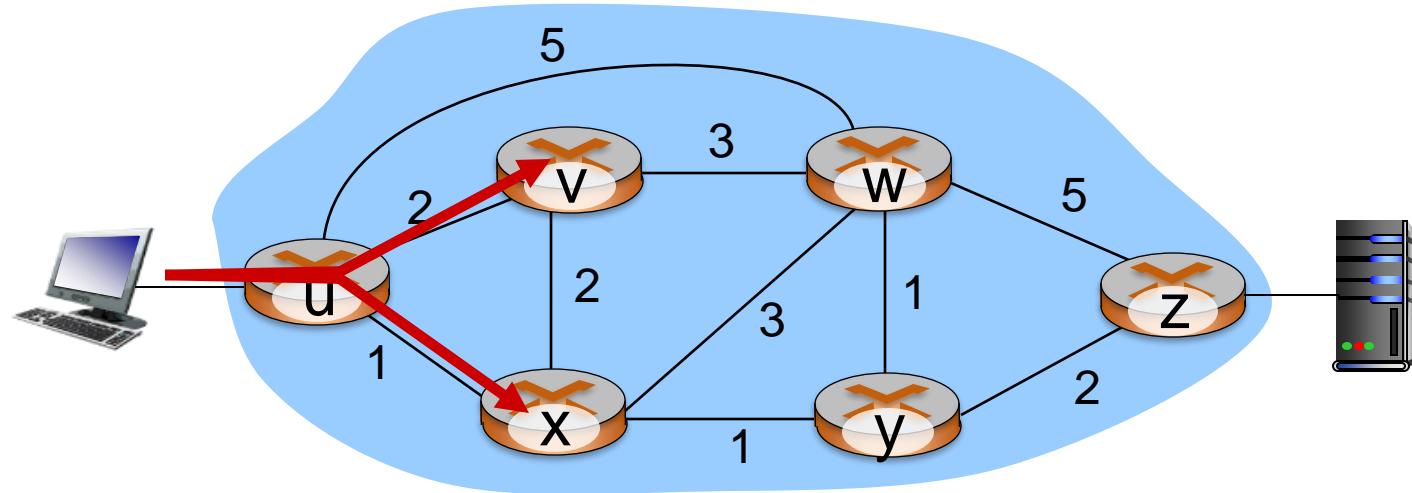
- easier network management: avoid router misconfigurations, greater flexibility of traffic flows
- table-based forwarding allows “programming” routers
- centralized programming easier: compute tables centrally and distribute
- distributed programming: more difficult: compute tables as result of distributed algorithm (protocol) implemented in each and every router
- open (non-proprietary) implementation of control plane

# Analogy: mainframe to PC evolution



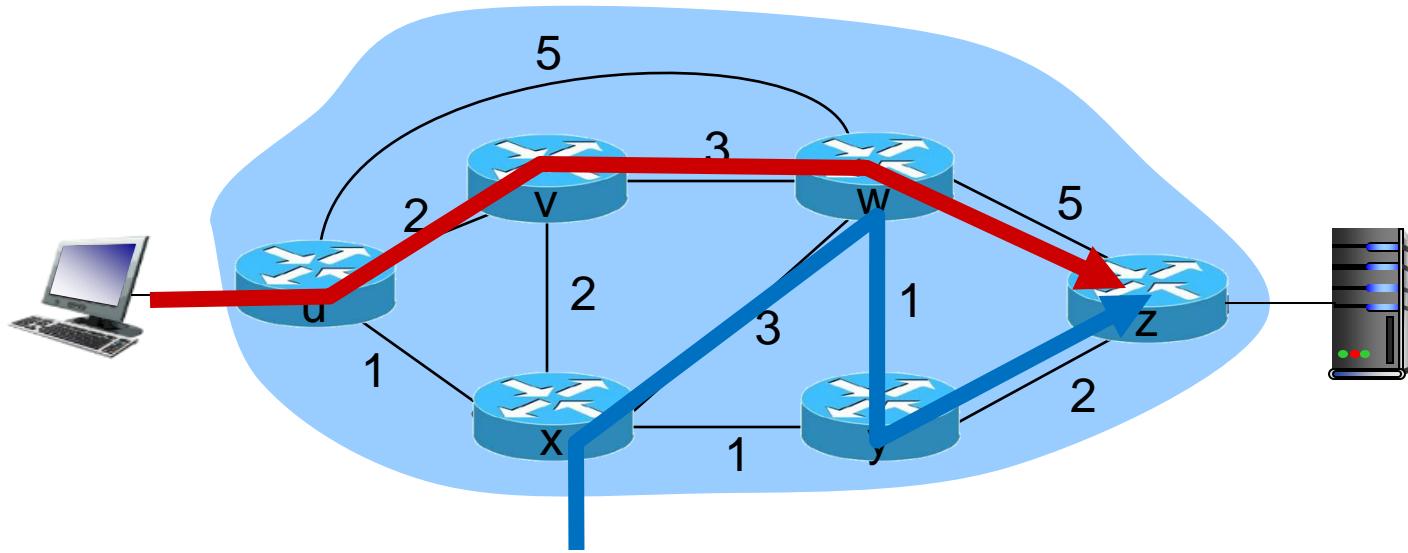
\* Slide courtesy: N. McKeown

# Traffic engineering: difficult traditional routing



Q: what if network operator wants to split u-to-z traffic along uvwz *and* uxzy (load balancing)?  
A: can't do it (or need a new routing algorithm)

# Traffic engineering: difficult traditional routing



Q: what if w wants to route blue and red traffic differently?

A: can't do it (with destination based forwarding, and LS, DV routing)

# Software defined networking (SDN)

4. programmable control applications

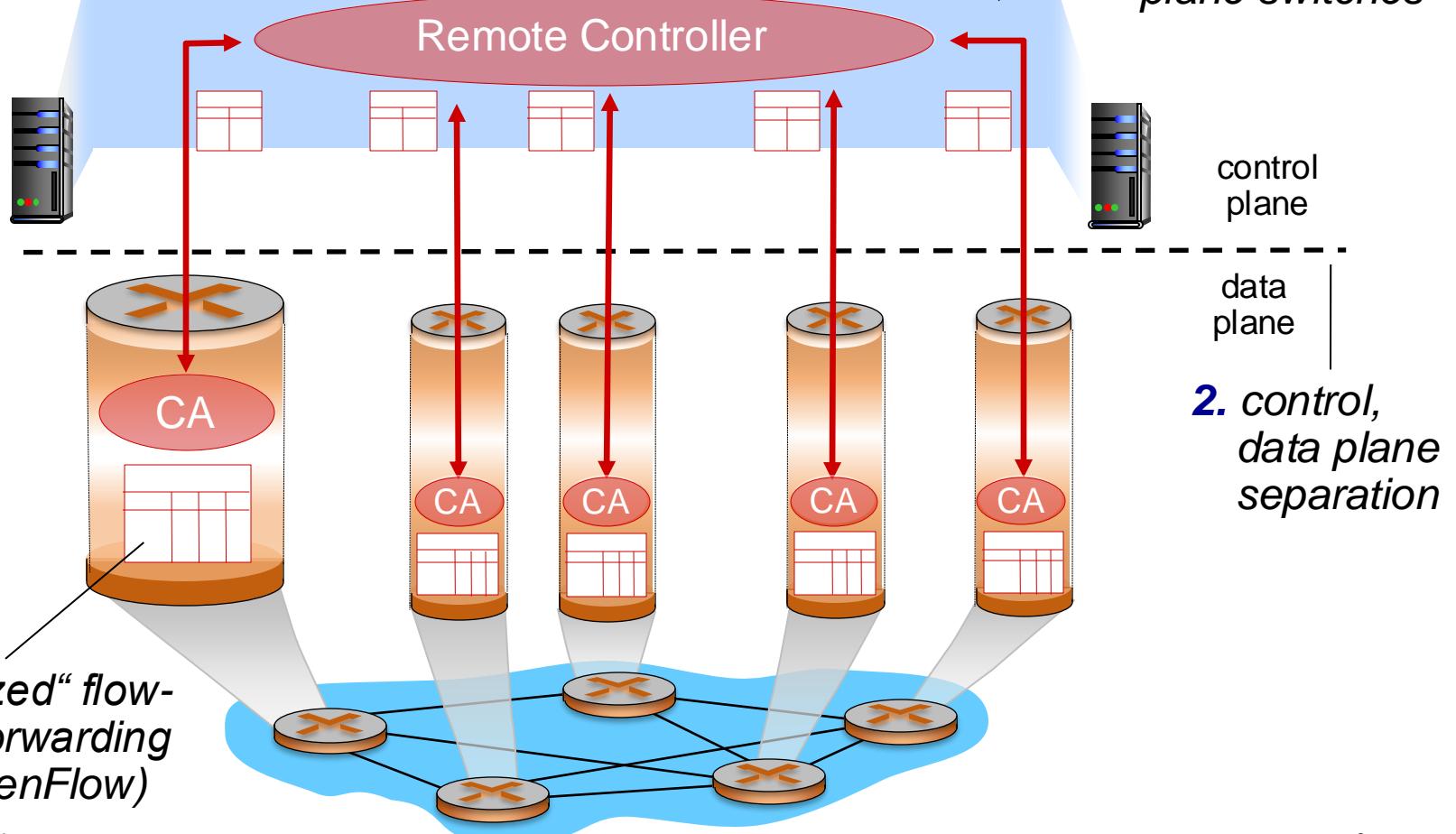
routing

access control

...

load balance

3. control plane functions external to data-plane switches



1: generalized “flow-based” forwarding  
(e.g., OpenFlow)

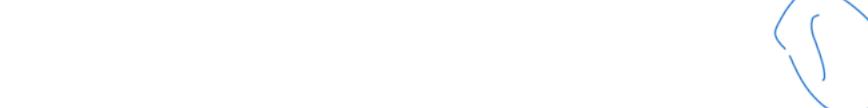
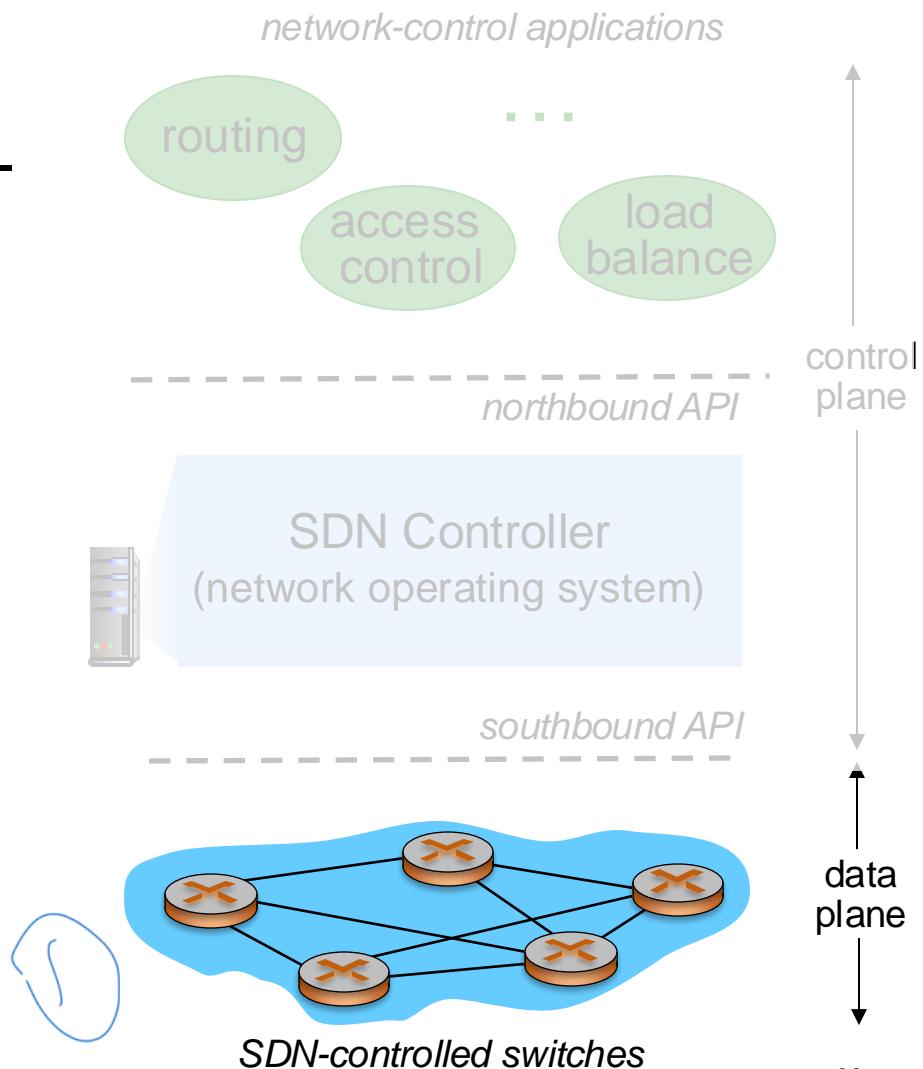
2. control,  
data plane  
separation

# SDN perspective: data plane switches



## Data plane switches

- fast, simple, commodity switches implementing generalized data-plane forwarding in hardware
- switch flow table computed, installed by controller
- API for table-based switch control (e.g., OpenFlow)
  - defines what is controllable and what is not
- protocol for communicating with controller (e.g., OpenFlow)

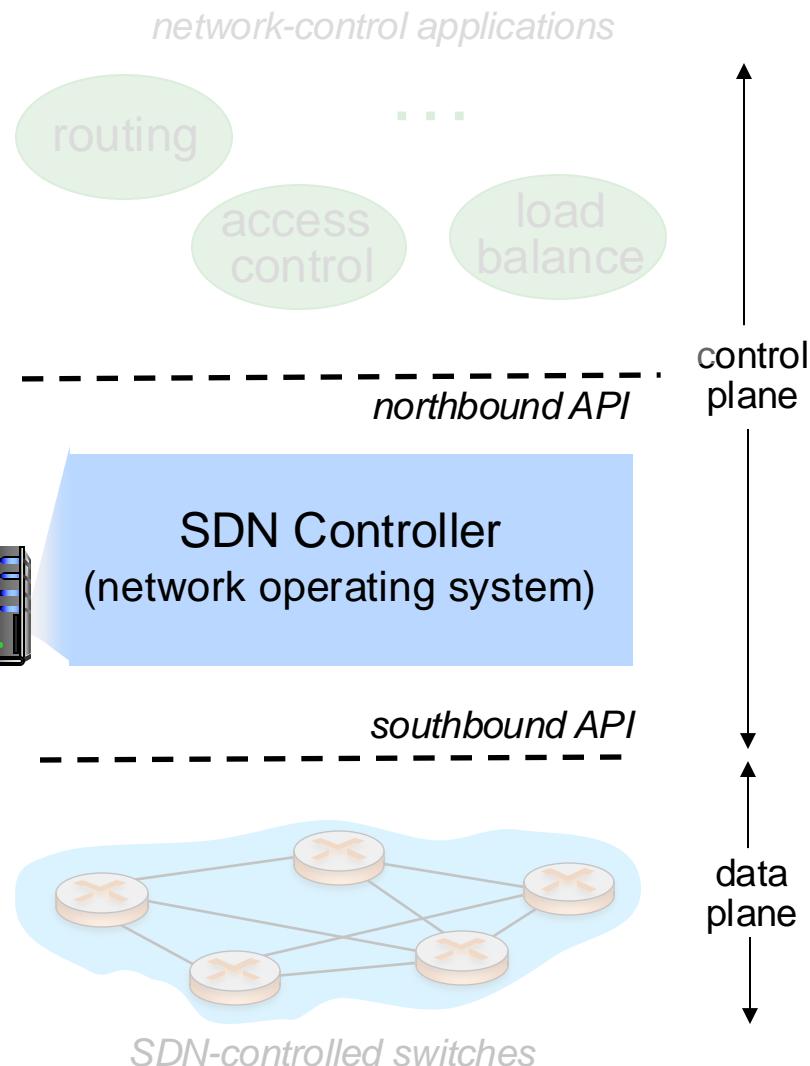


# SDN perspective: SDN controller



## SDN controller (*network OS*):

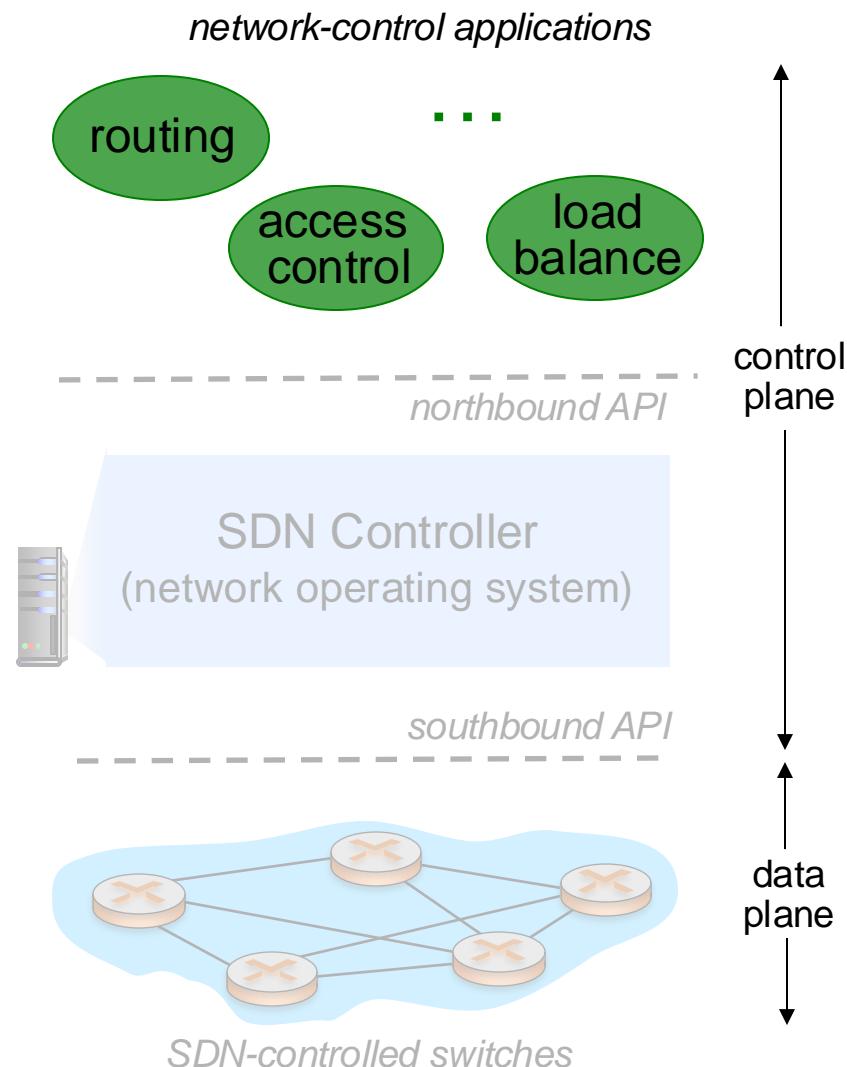
- maintain network state information
- interacts with network control applications “above” via northbound API
- interacts with network switches “below” via southbound API
- implemented as distributed system for performance, scalability, fault-tolerance, robustness



# SDN perspective: control applications

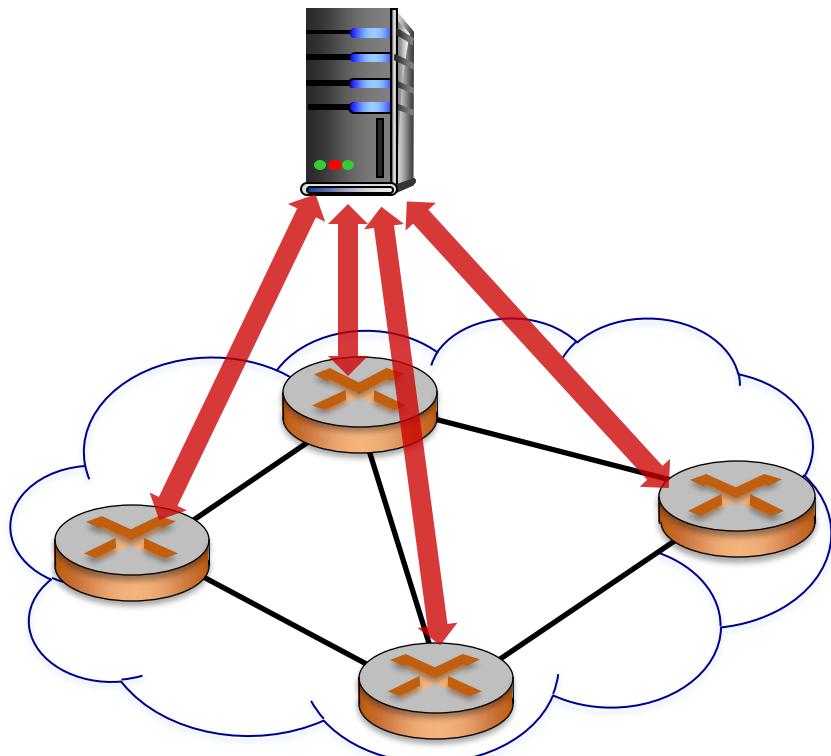
## network-control apps:

- “brains” of control: implement control functions using lower-level services, API provided by SDN controller
- unbundled: can be provided by 3<sup>rd</sup> party: distinct from routing vendor, or SDN controller



# OpenFlow protocol

## OpenFlow Controller

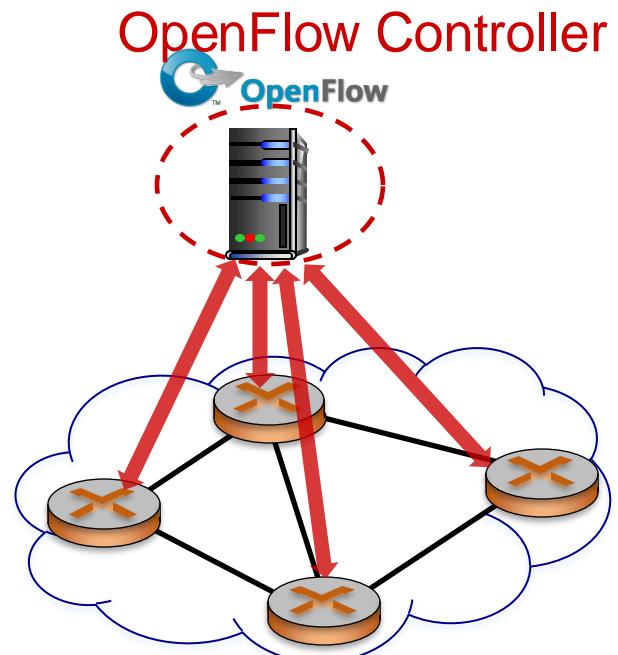


- operates between controller, switch
- TCP used to exchange messages
  - optional encryption
- OpenFlow messages:
  - controller-to-switch
  - switch-to-controller

# OpenFlow: controller-to-switch messages

## Key controller-to-switch messages

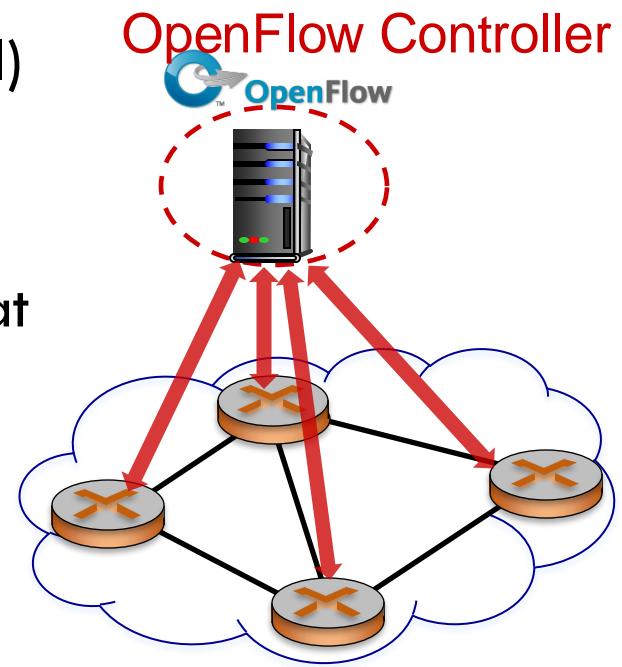
- **features**: controller queries switch features, switch replies
- **configure**: controller queries/sets switch configuration parameters
- **modify-state**: add, delete, modify flow entries in the OpenFlow tables
- **packet-out**: controller can send this packet out of specific switch port



# OpenFlow: switch-to-controller messages

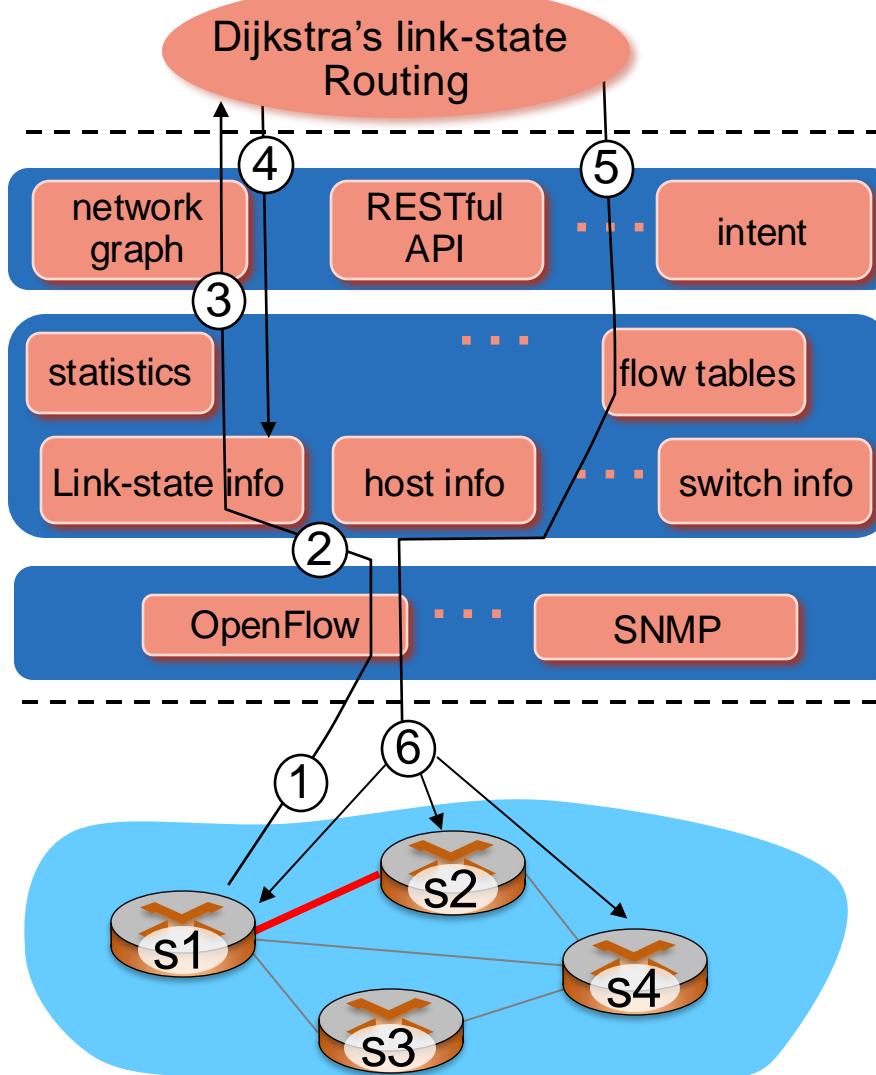
## Key switch-to-controller messages

- **packet-in:** transfer packet (and its control) to controller. See packet-out message from controller
- **flow-removed:** flow table entry deleted at switch
- **port status:** inform controller of a change on a port.



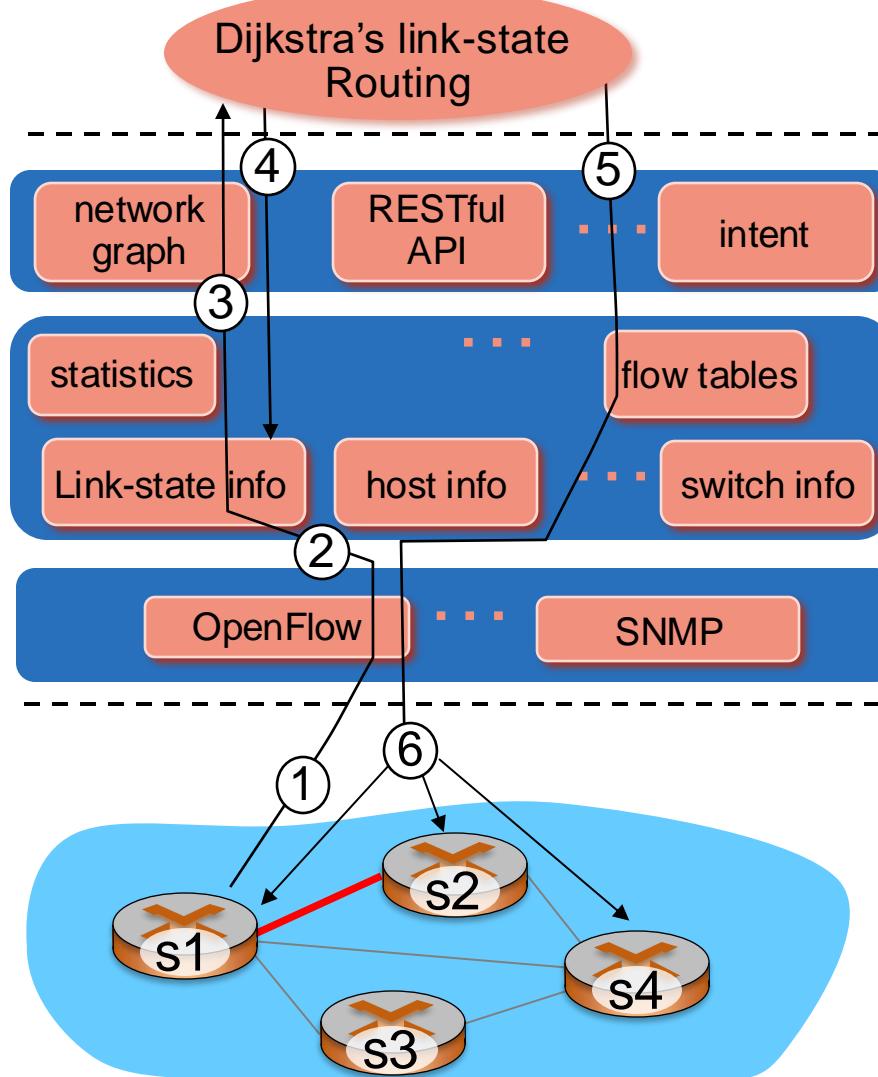
Fortunately, network operators don't "program" switches by creating/sending OpenFlow messages directly. Instead use higher-level abstraction at controller

# SDN: control/data plane interaction example



- ① S1, experiencing link failure using OpenFlow port status message to notify controller  
*openflow*
- ② SDN controller receives OpenFlow message, updates link status info
- ③ Dijkstra's routing algorithm application has previously registered to be called whenever link status changes. It is called.
- ④ Dijkstra's routing algorithm access network graph info, link state info in controller, computes new routes

# SDN: control/data plane interaction example



- ⑤ link state routing app interacts with flow-table-computation component in SDN controller, which computes new flow tables needed
- ⑥ Controller uses OpenFlow to install new tables in switches that need updating

# **Transport Layer**

# Transport Layer

第6章 第二部分，无线 transfer layer

## our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

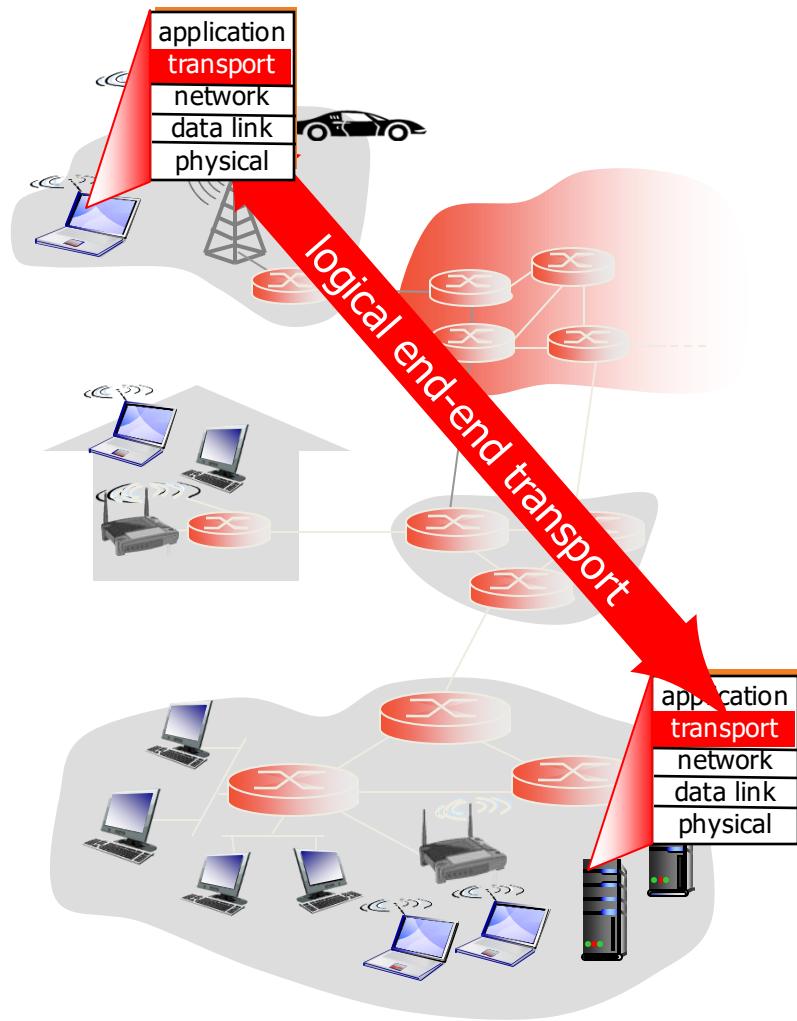
# Outline

- Transport-layer services
- Multiplexing/demultiplexing
- Connectionless transport (UDP)
- Principles of reliable data transfer
- TCP

# **Transport Services**

# Transport services and protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into **segments**, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet:TCP and UDP

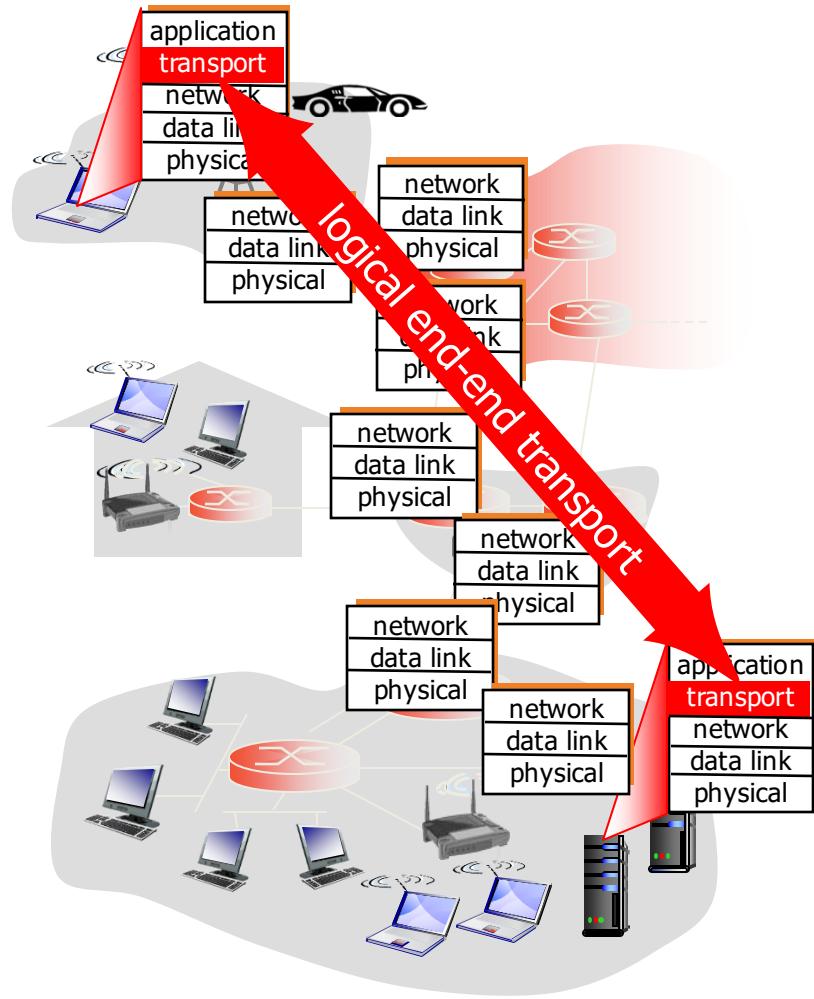


## Transport vs. network layer

- *network layer*: host-to-host communication
  - best-effort, unreliable
- *transport layer*: process-to-process communication
  - relies on, enhances, network layer services

# Internet transport-layer protocols

- IP: best effort service
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



# **Transport Services**

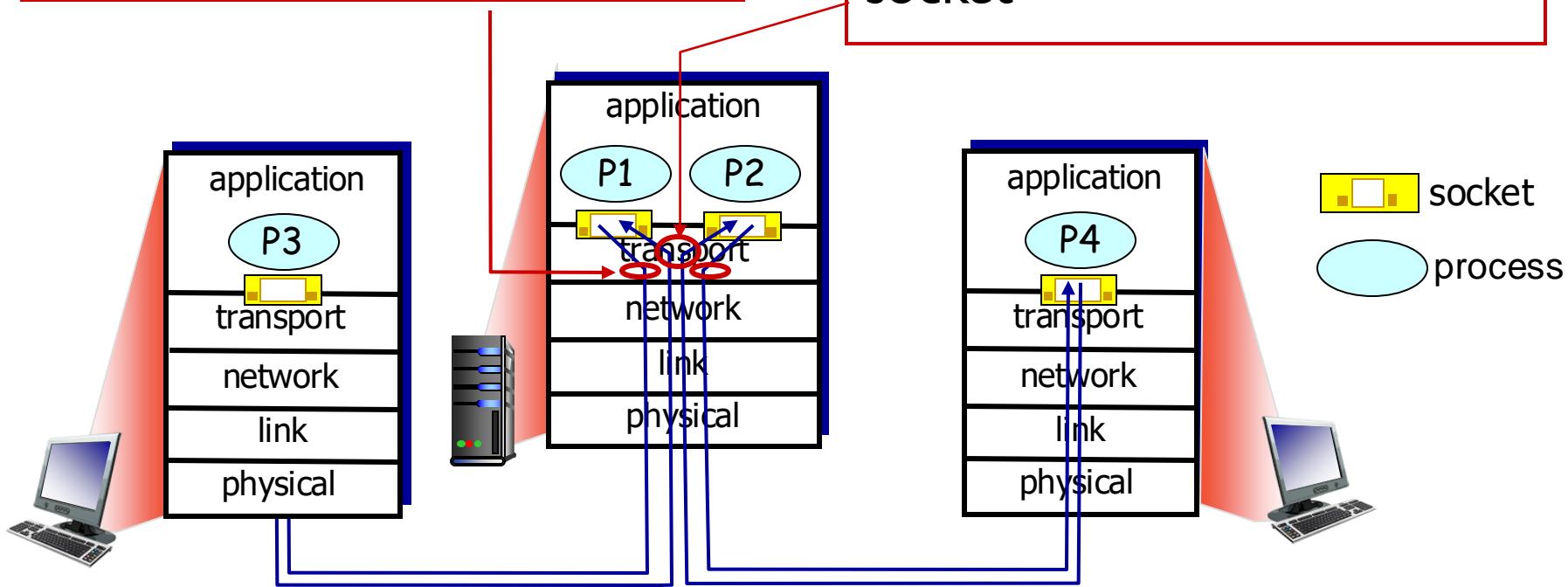
# Multiplexing/demultiplexing

*multiplexing at sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

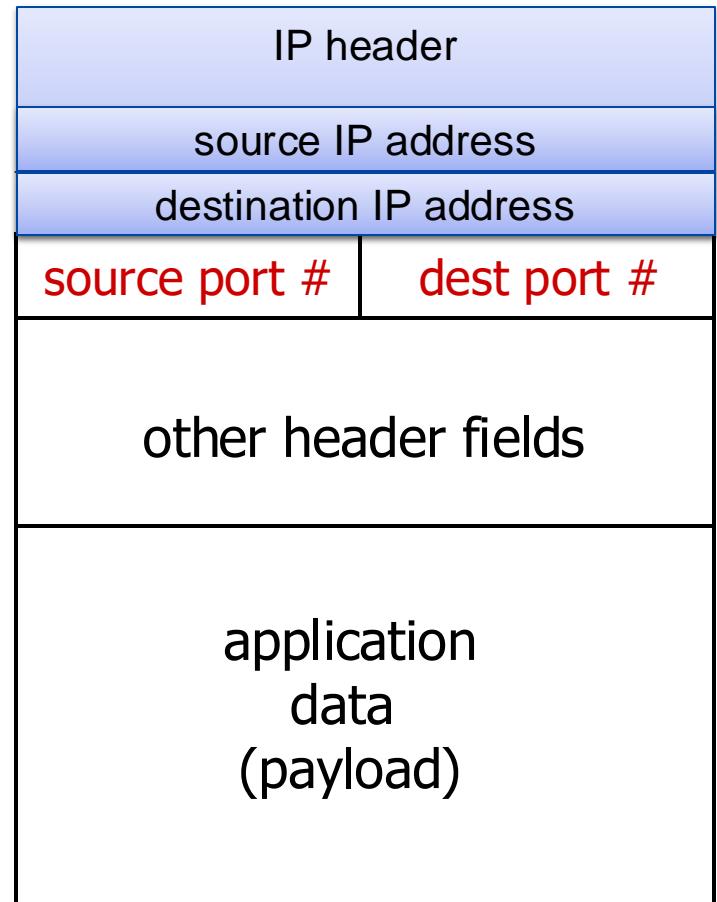
*demultiplexing at receiver:*

use header info to deliver received segments to correct socket



# How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

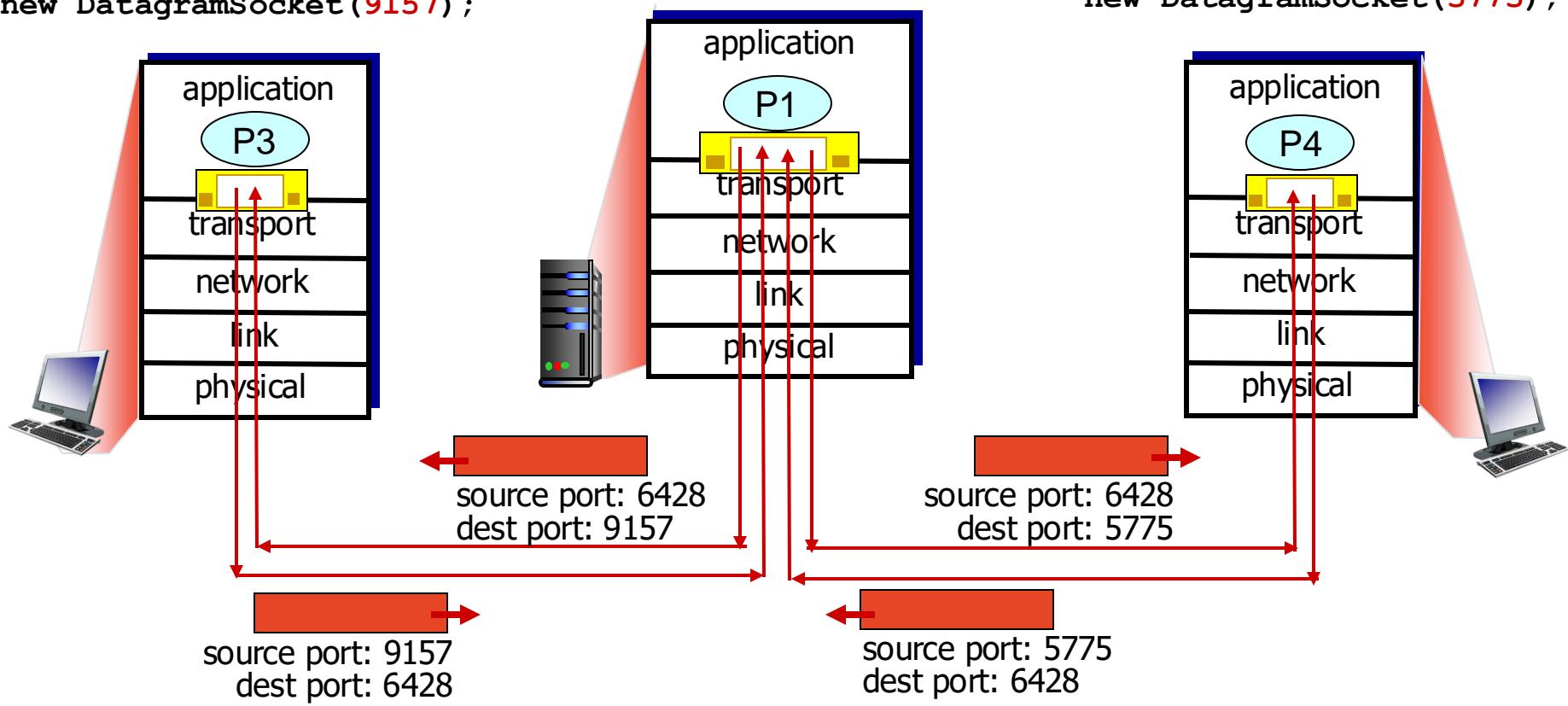
# Connectionless demux: example

VDP

```
DatagramSocket serverSocket =  
    new DatagramSocket(6428);
```

```
DatagramSocket mySocket2 =  
    new DatagramSocket(9157);
```

```
DatagramSocket mySocket1 =  
    new DatagramSocket(5775);
```

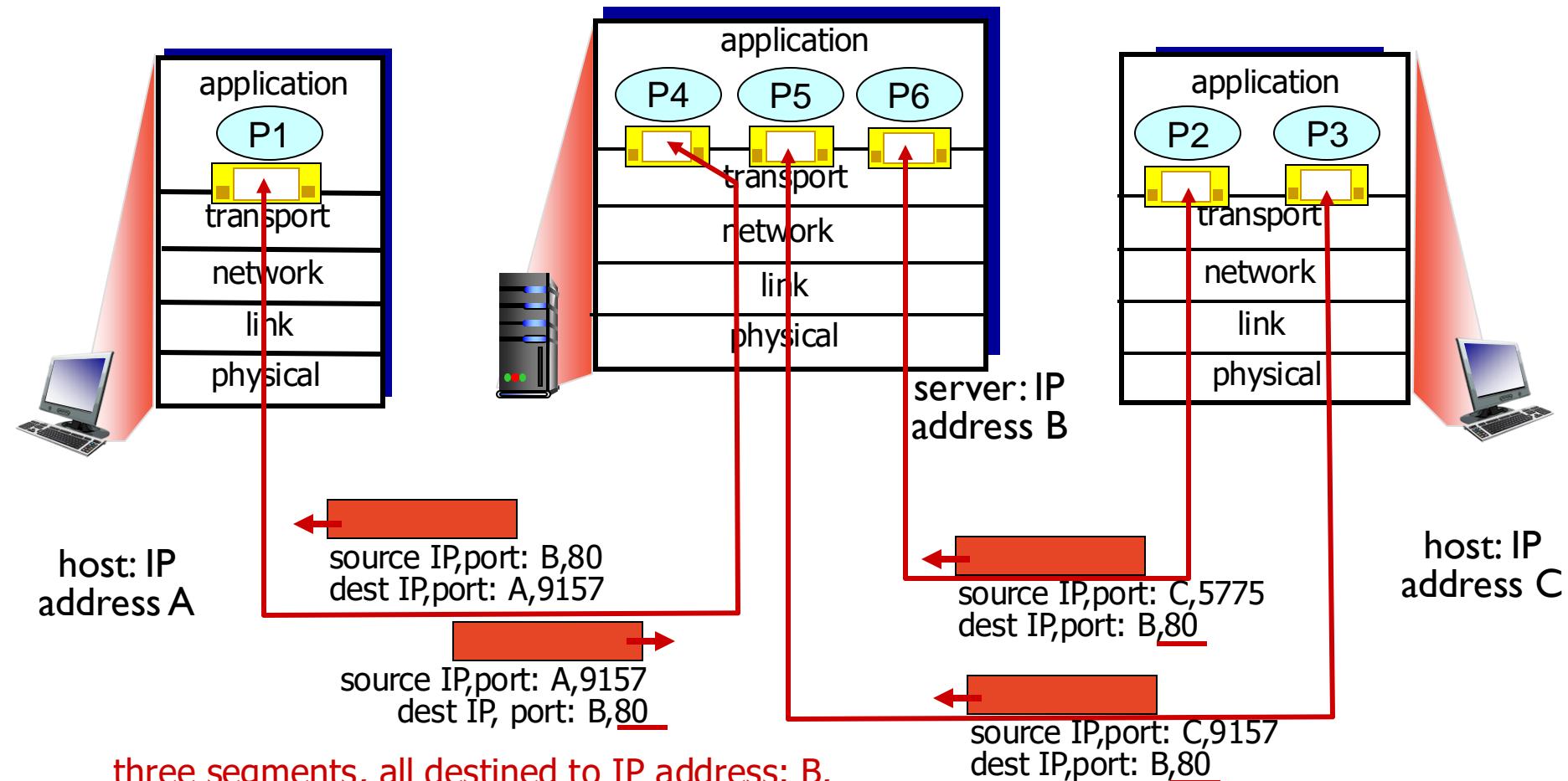


## Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - web servers have different sockets for each connecting client

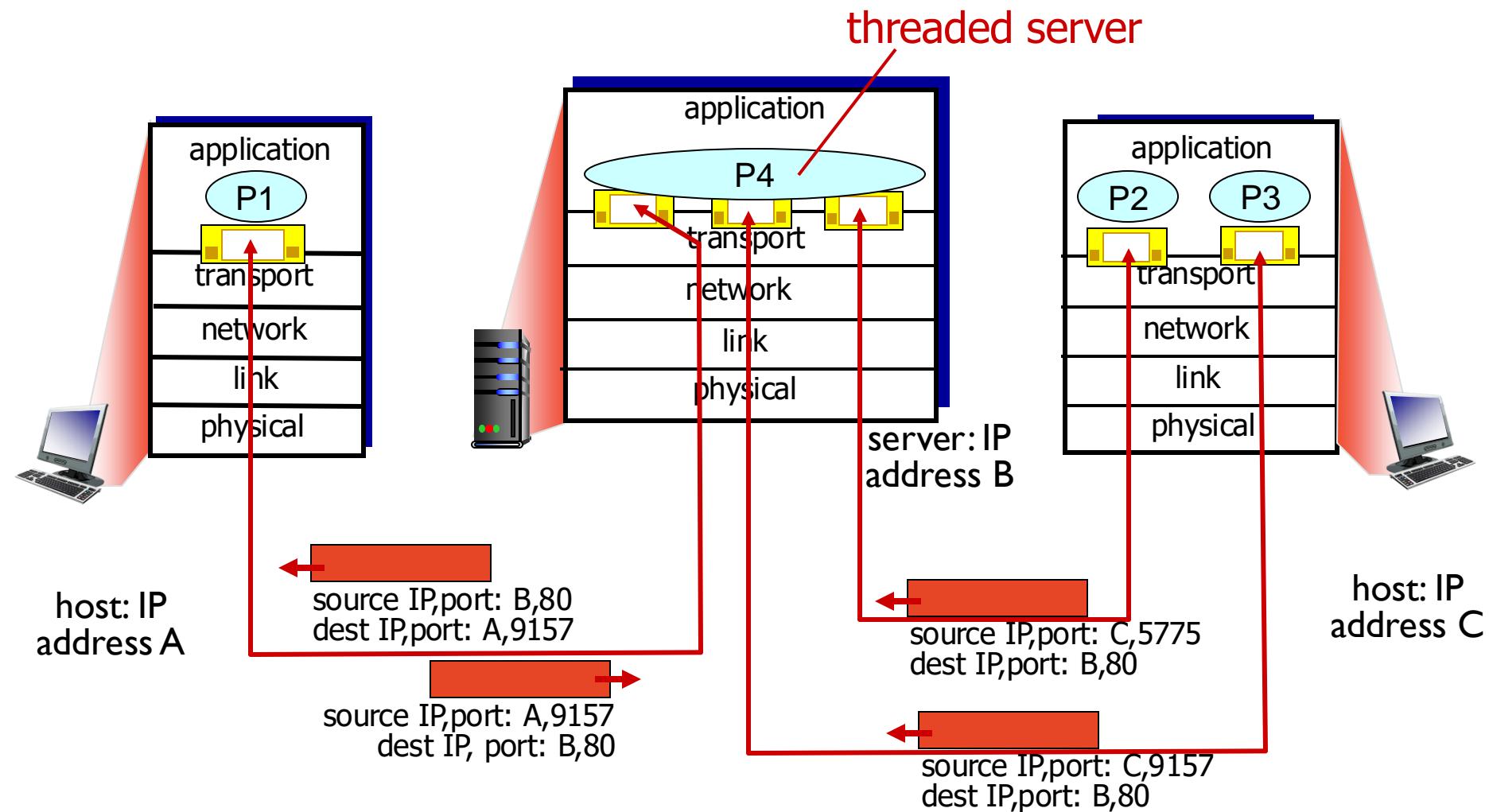
# Connection-oriented demux: example

TCP



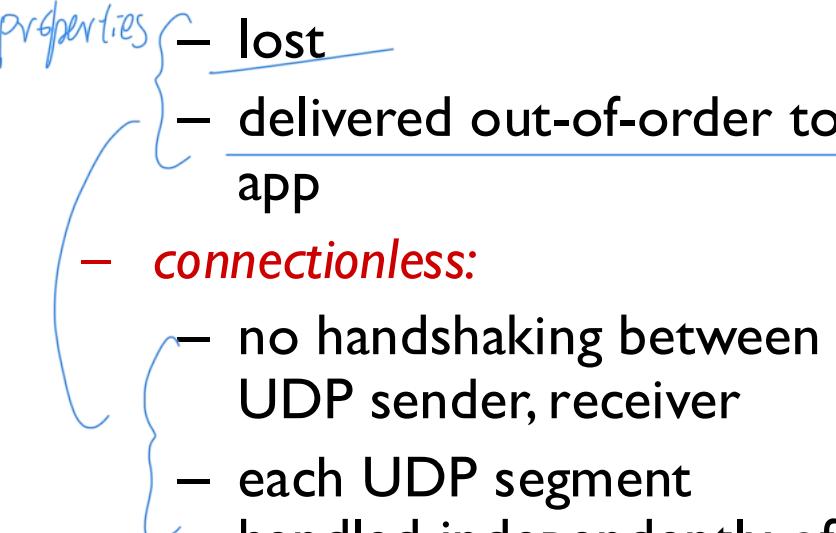
three segments, all destined to IP address: B,  
dest port: 80 are demultiplexed to *different* sockets

# Connection-oriented demux: example



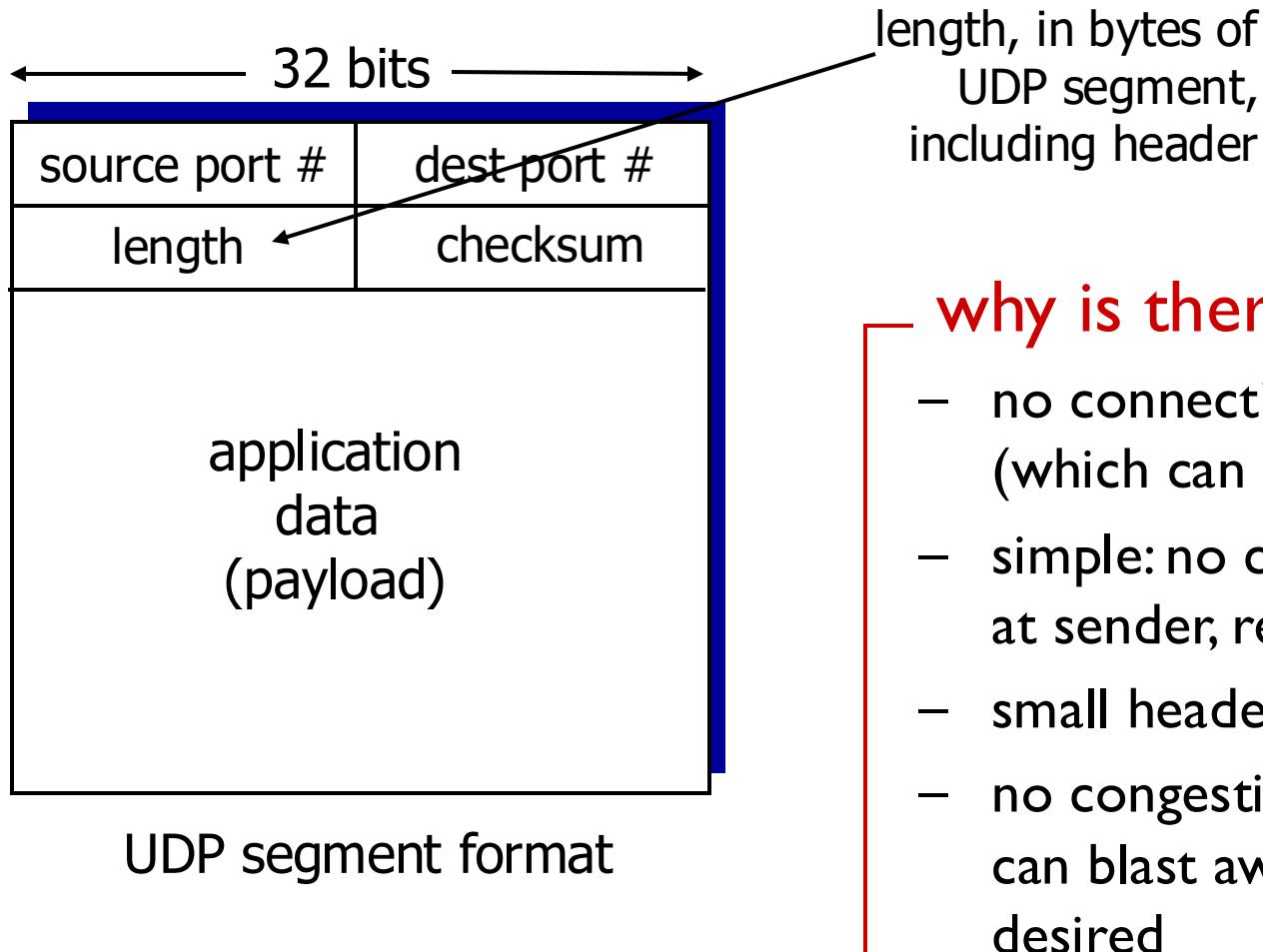
# **Connectionless Transport UDP**

# UDP: User Datagram Protocol [RFC 768]

- “no frills,” 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
- properties*
- 

- ❖ UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
- ❖ reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

# UDP: segment header



## why is there a UDP?

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

# UDP checksum

检测错误

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

## sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- sum: addition (one's complement sum) of segment contents
- checksum: complement of sum
- 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.

## Internet checksum: example

example: add two 16-bit integers

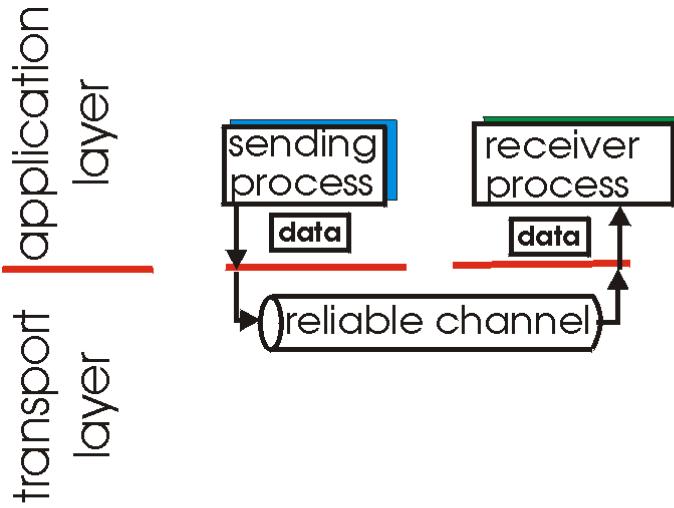
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
carryout	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
wraparound	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

# **Principles of Reliable Data Transfer**

# Principles of reliable data transfer

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

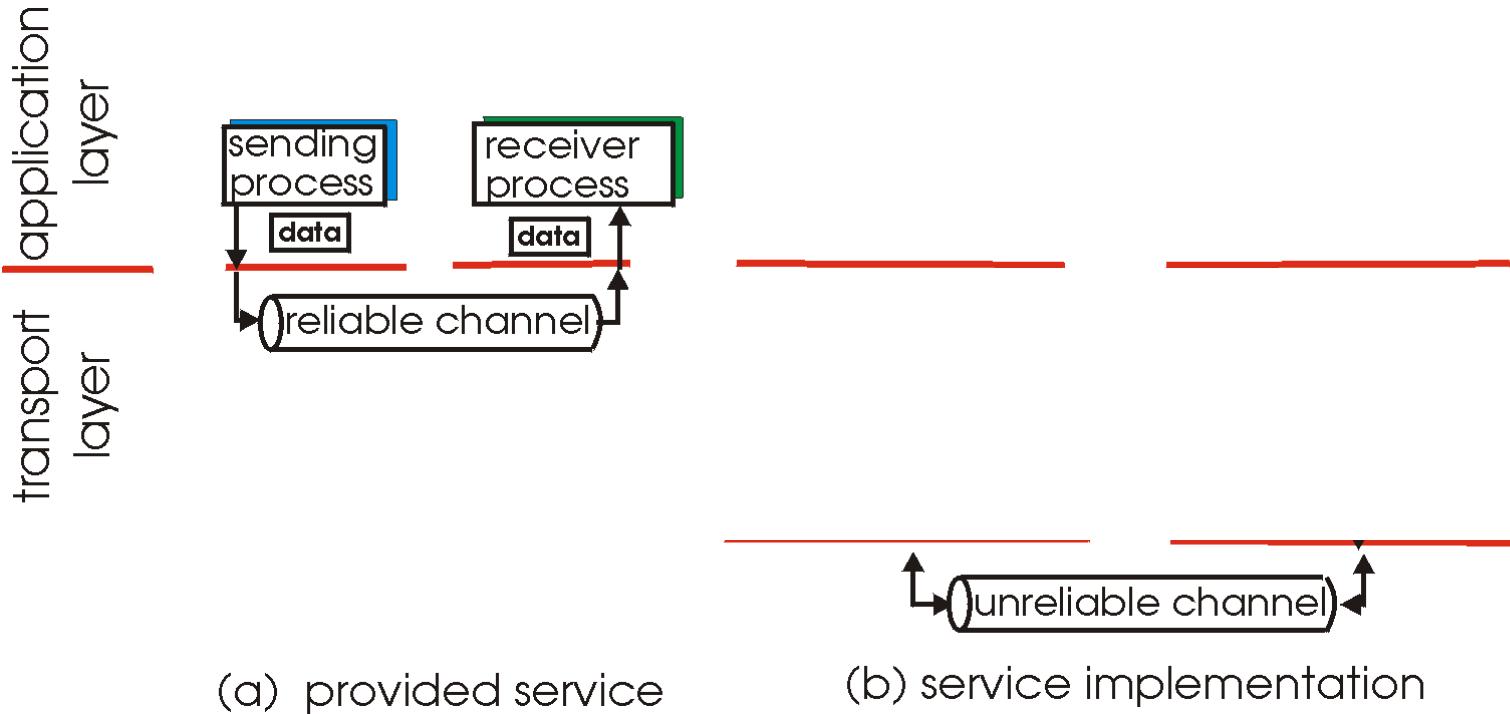


(a) provided service

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

# Principles of reliable data transfer

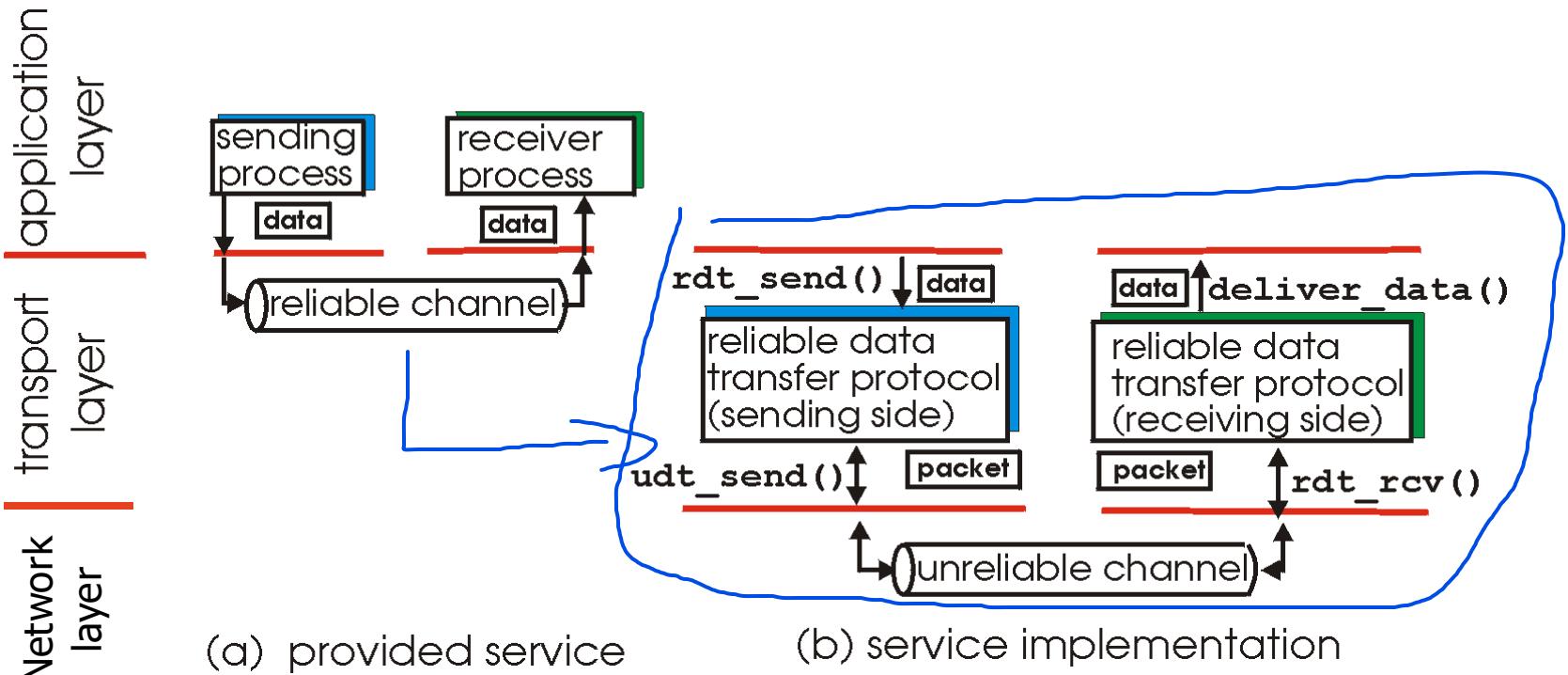
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

# Principles of reliable data transfer

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

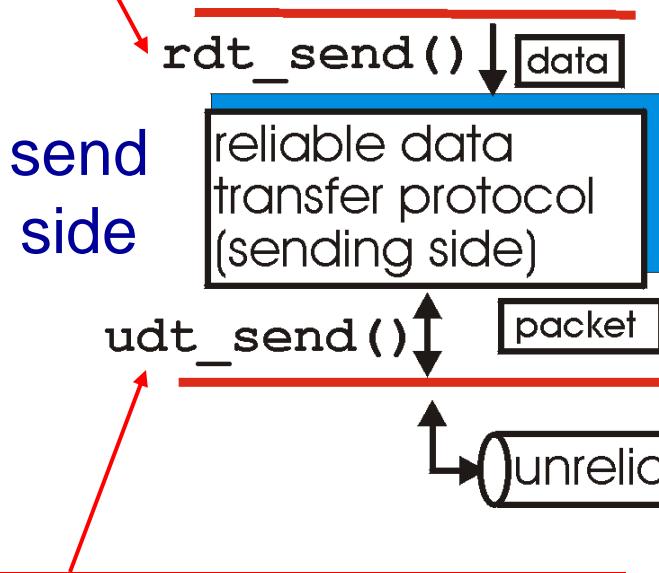


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

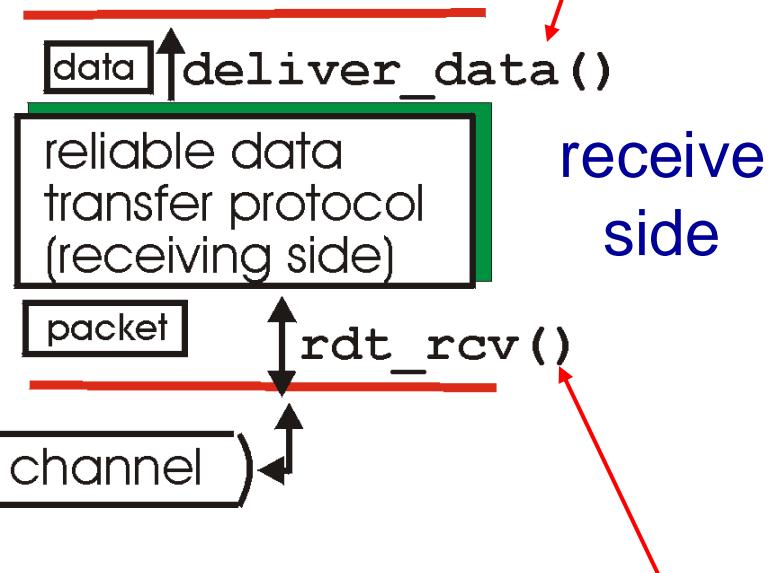
RDT

# Principles of reliable data transfer

**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

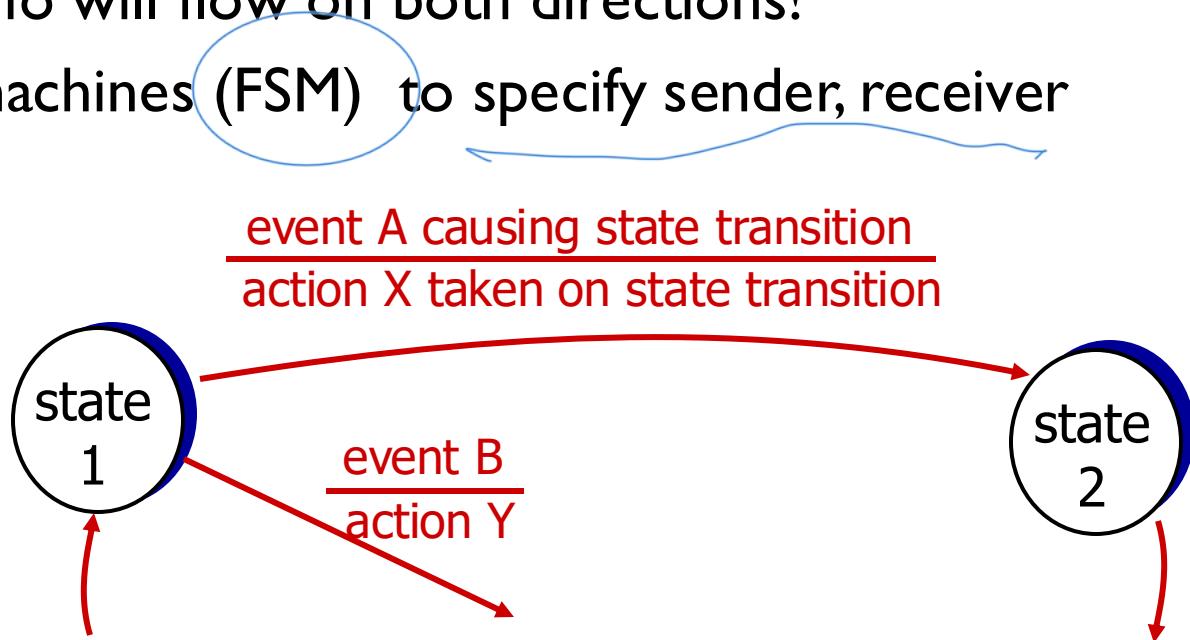
**rdt\_rcv()** : called when packet arrives on rcv-side of channel

# Principles of reliable data transfer

We will:

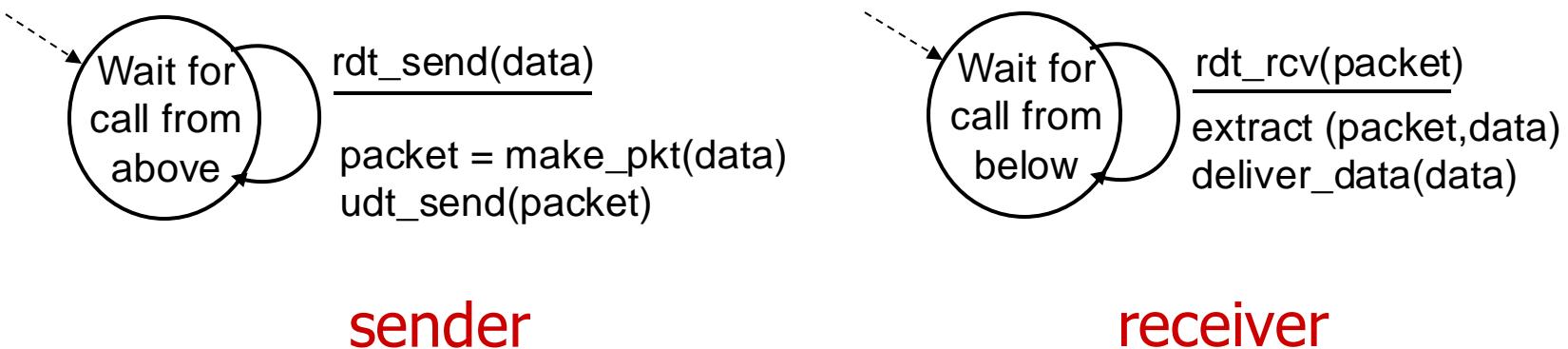
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

**state:** when in this “state”, next state and action uniquely determined by next event



# Principles of reliable data transfer

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel *RDT 1.0*
  - receiver reads data from underlying channel



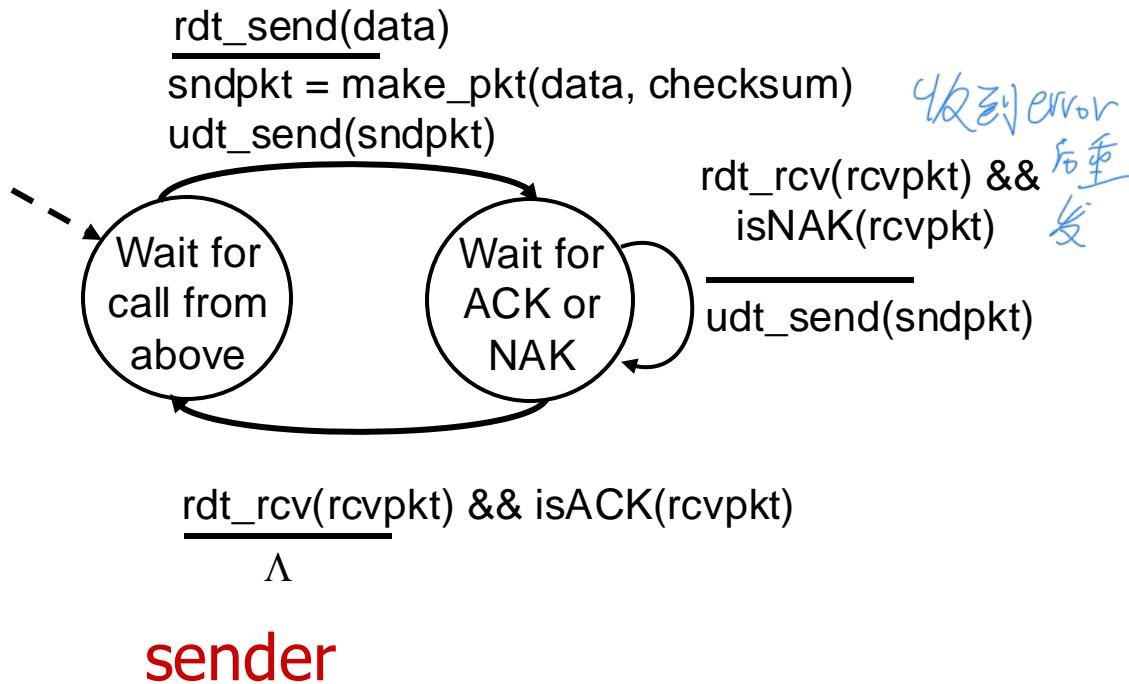
## rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

## rdt2.0: channel with bit errors

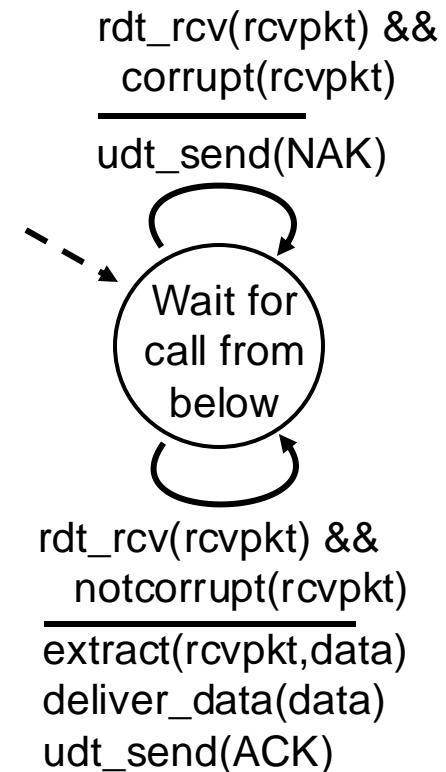
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- the question: how to recover from errors:
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

# rdt2.0: FSM specification



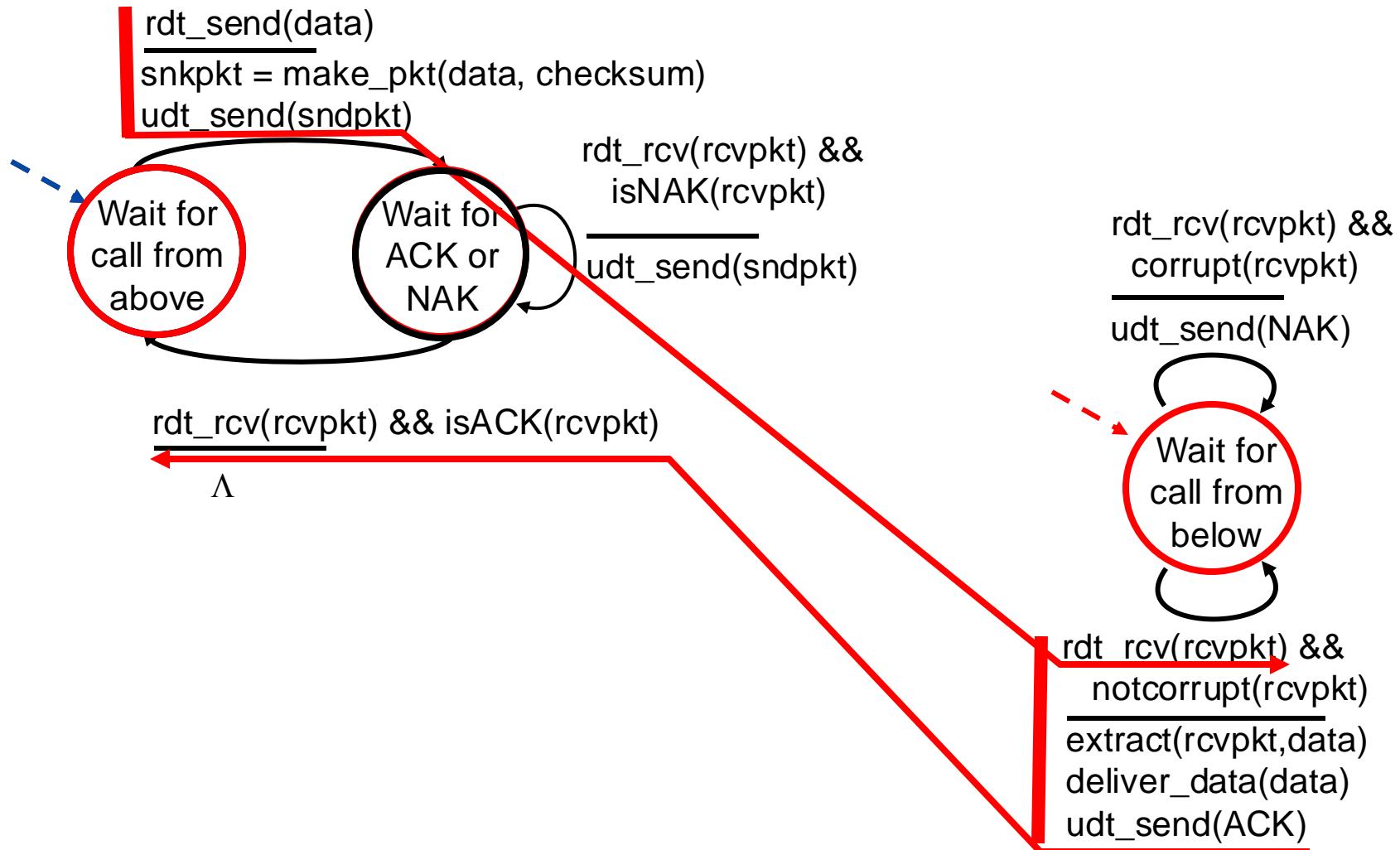
发送方

receiver

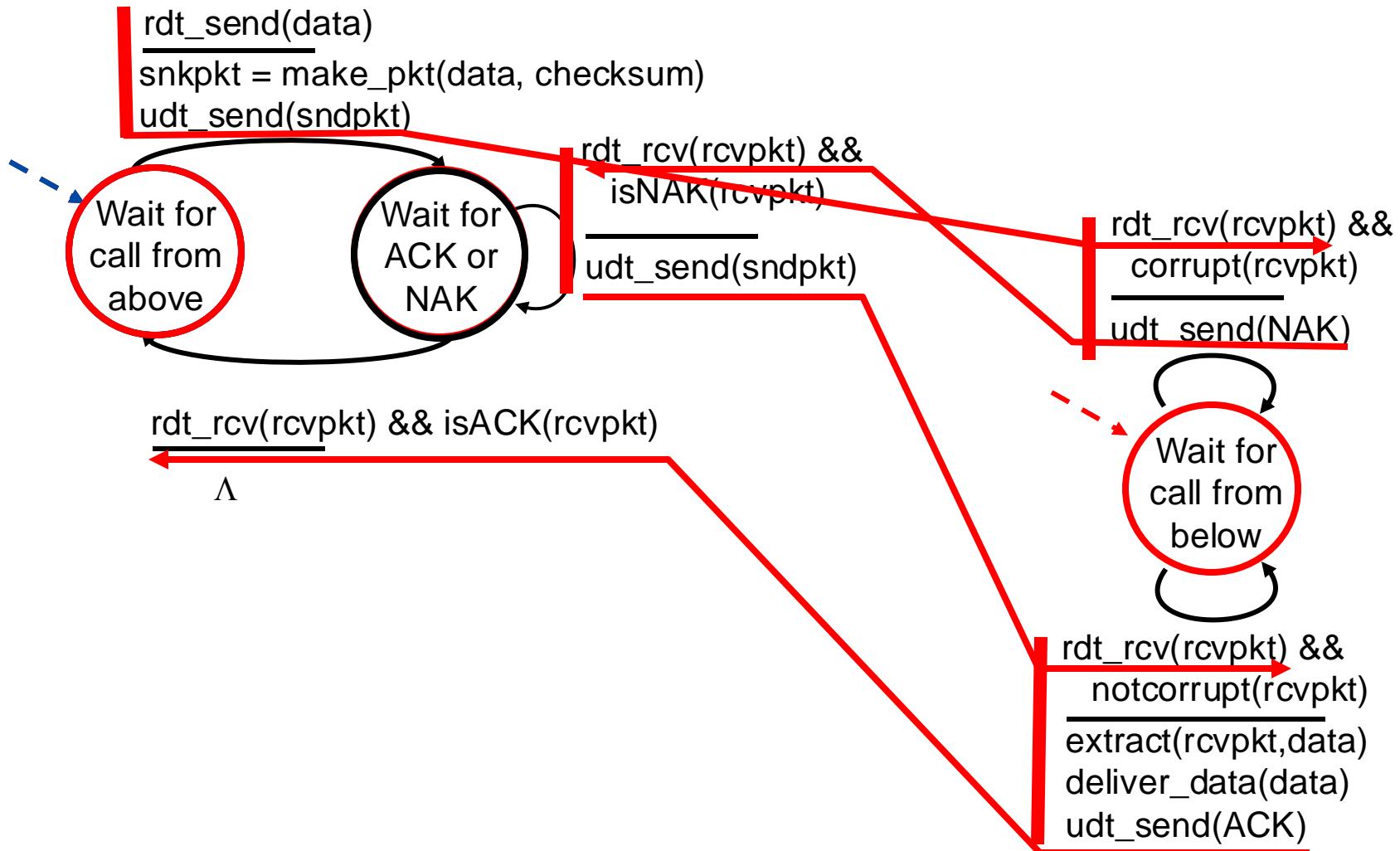


接收方

## rdt2.0: operation with no errors



## rdt2.0: error scenario



# rdt2.0 has a fatal flaw!

## what happens if ACK/NAK corrupted?

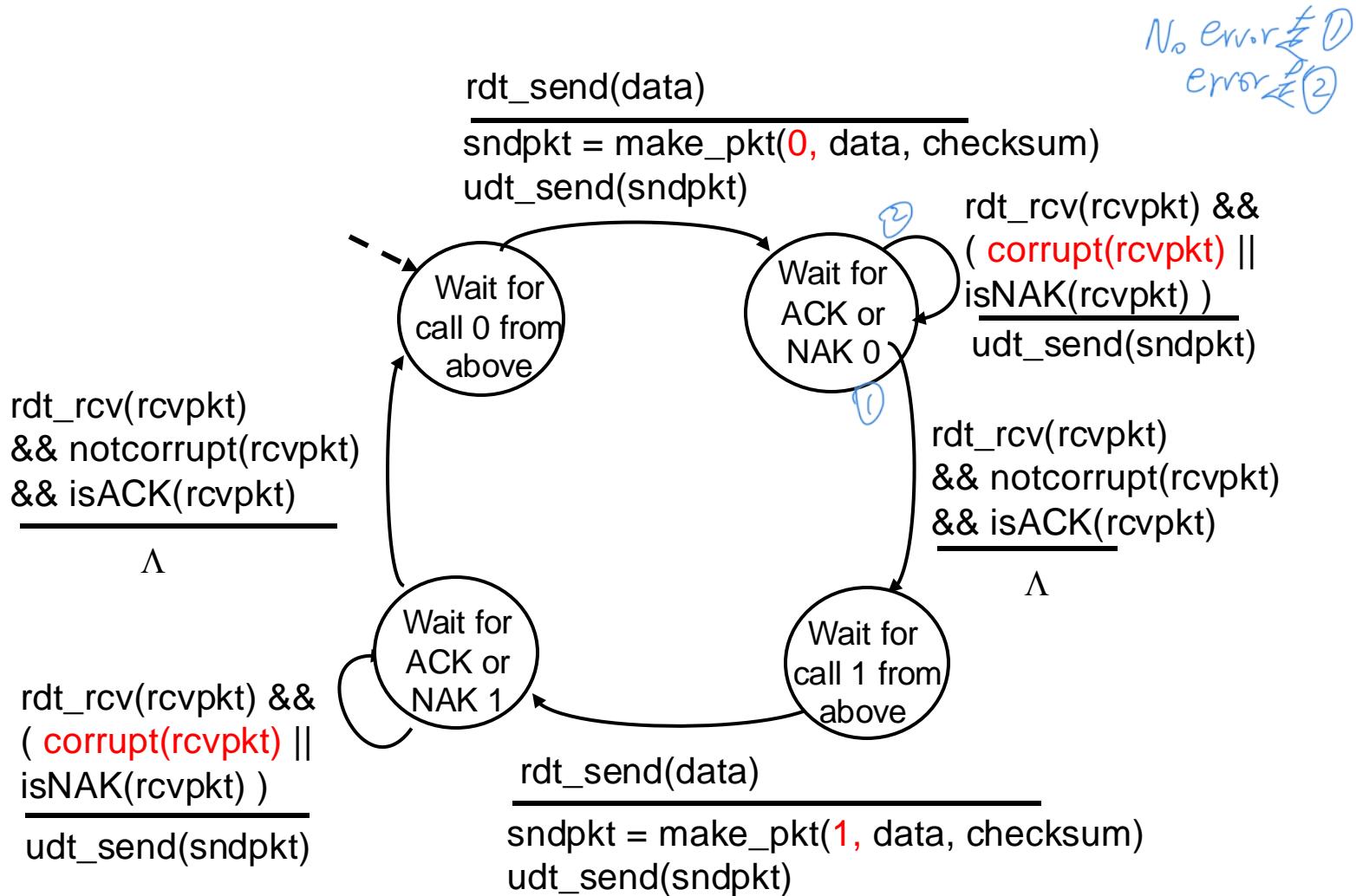
- sender does not know what happened at receiver!
- cannot just retransmit: possible duplicate

## handling duplicates:

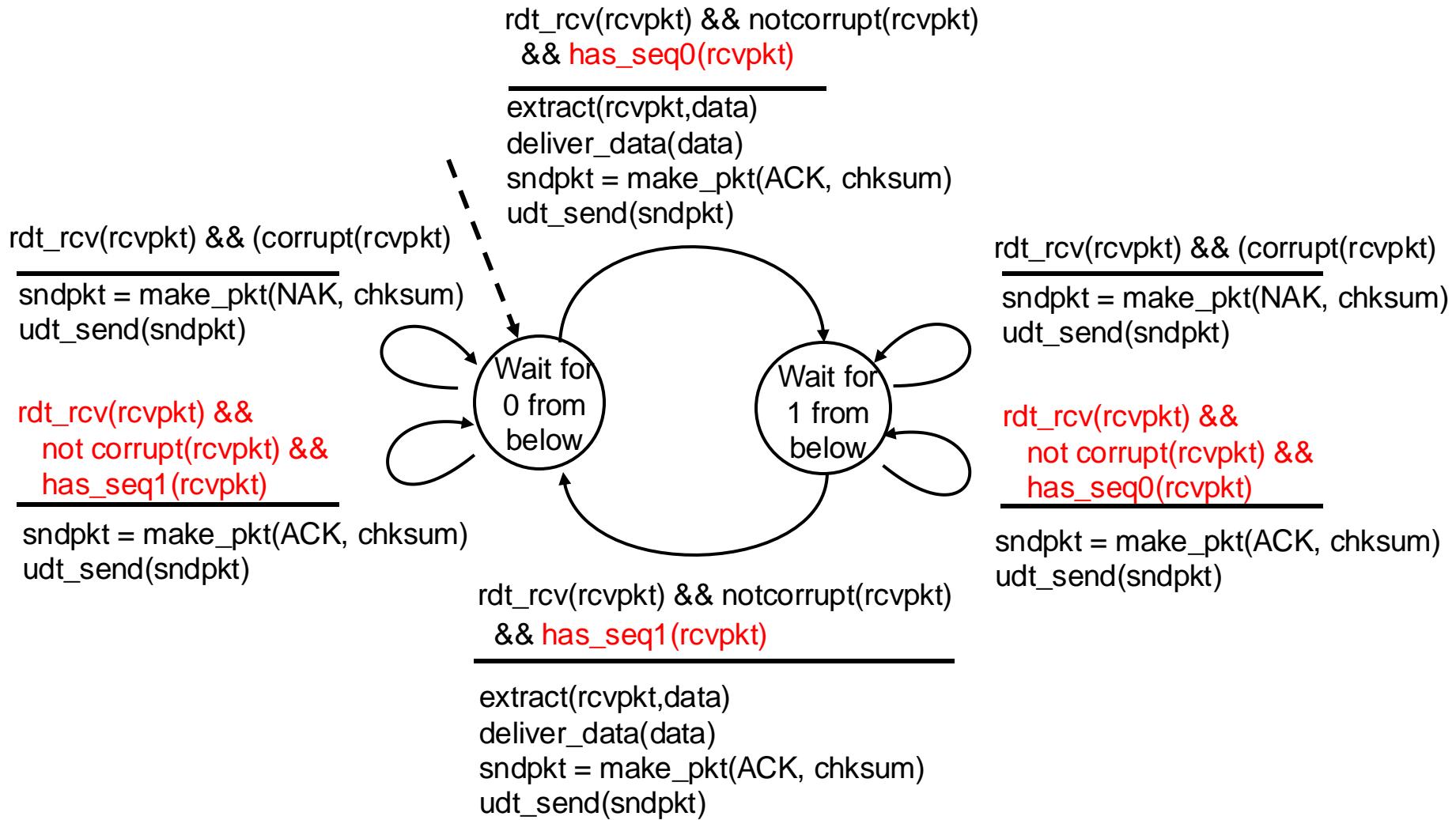
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (does not deliver up) duplicate pkt

**stop and wait**  
sender sends one packet,  
then waits for receiver  
response

## rdt2.1: sender, handles garbled ACK/NAKs



## rdt2.1: receiver, handles garbled ACK/NAKs



## rdt2.1: discussion

### sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice.
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

### receiver:

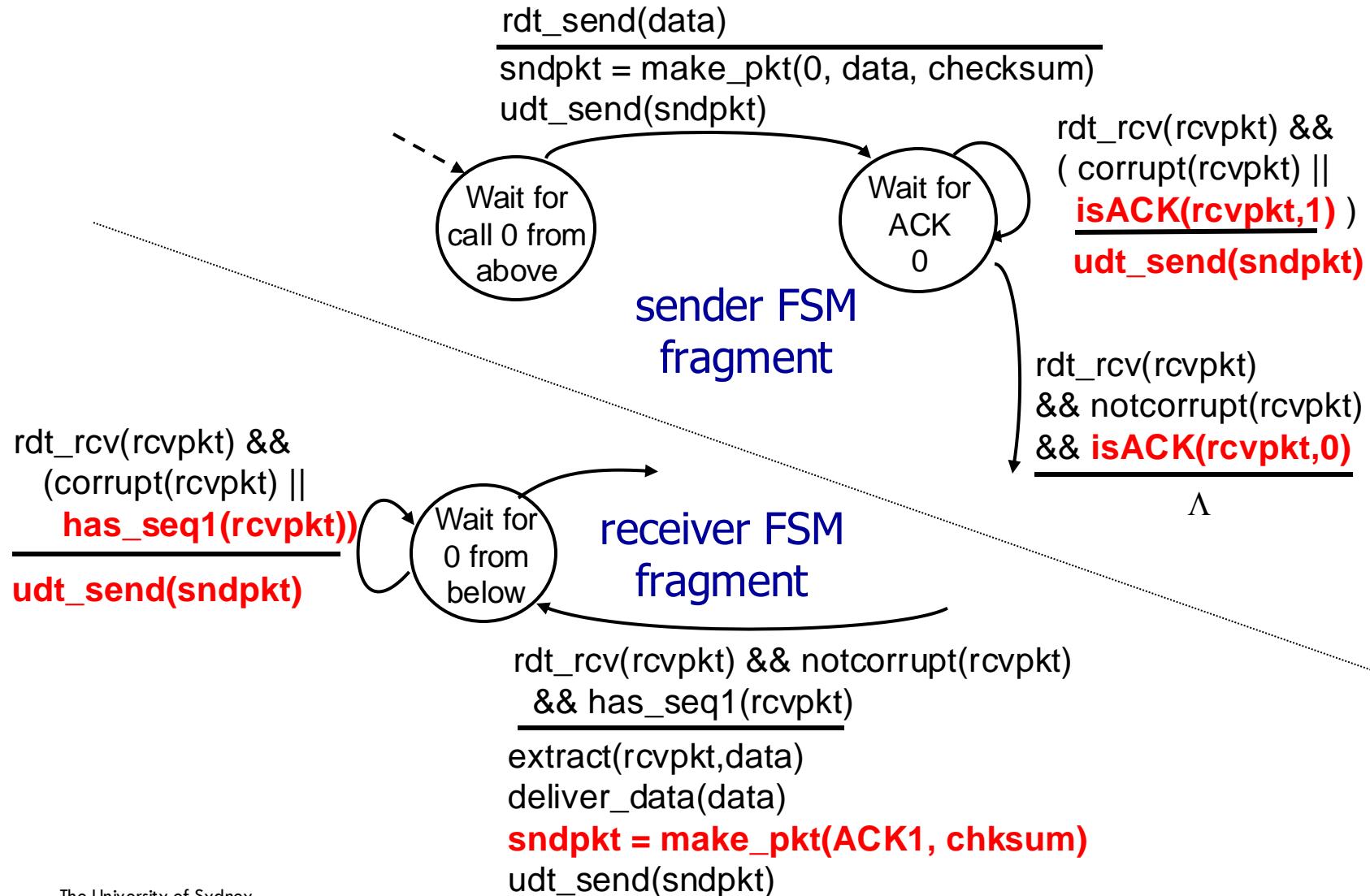
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #

## rdt2.2: a NAK-free protocol

另外一种方法

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- "unexpected" ACK at sender results in same action as NAK: *retransmit current pkt*

## rdt2.2: sender, receiver fragments



## rdt3.0: channels with errors and loss

new assumption: underlying

channel can also lose  
packets (data, ACKs)

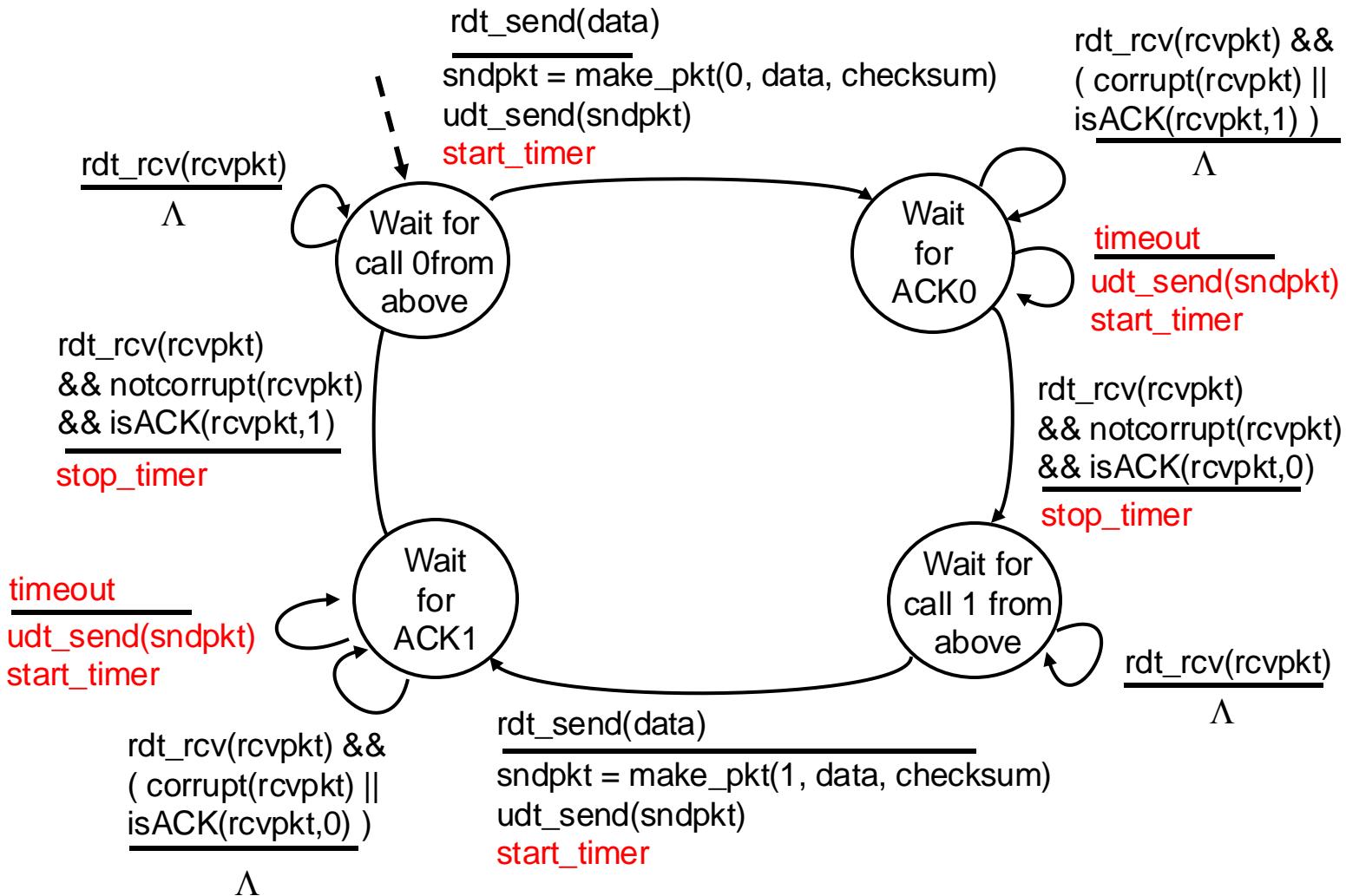
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

approach: sender waits

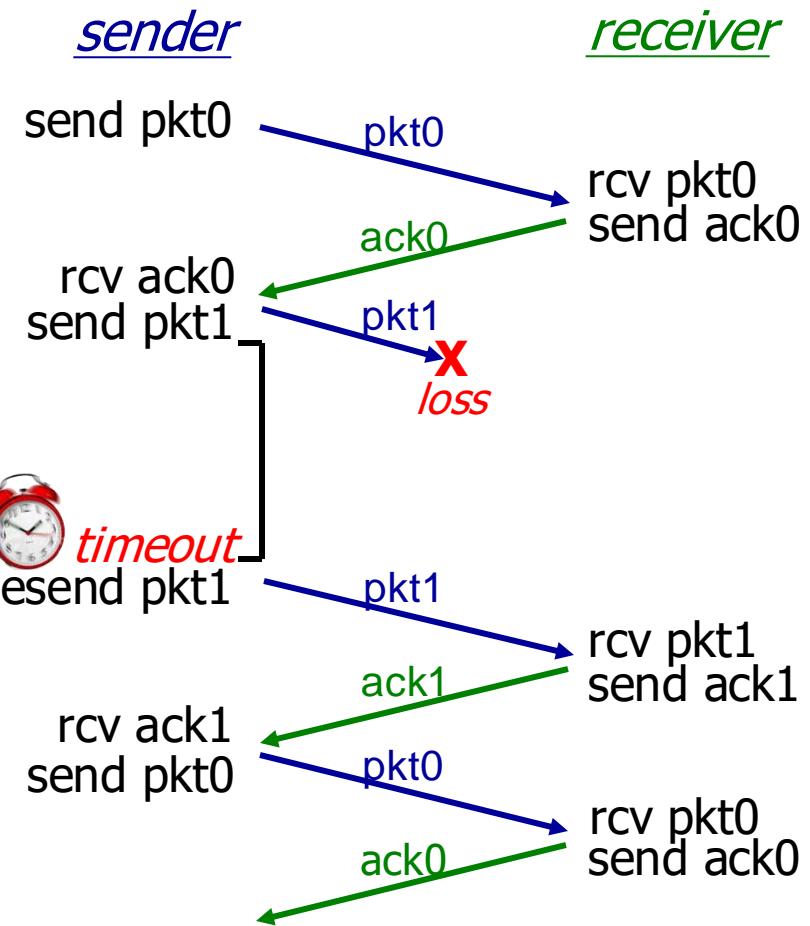
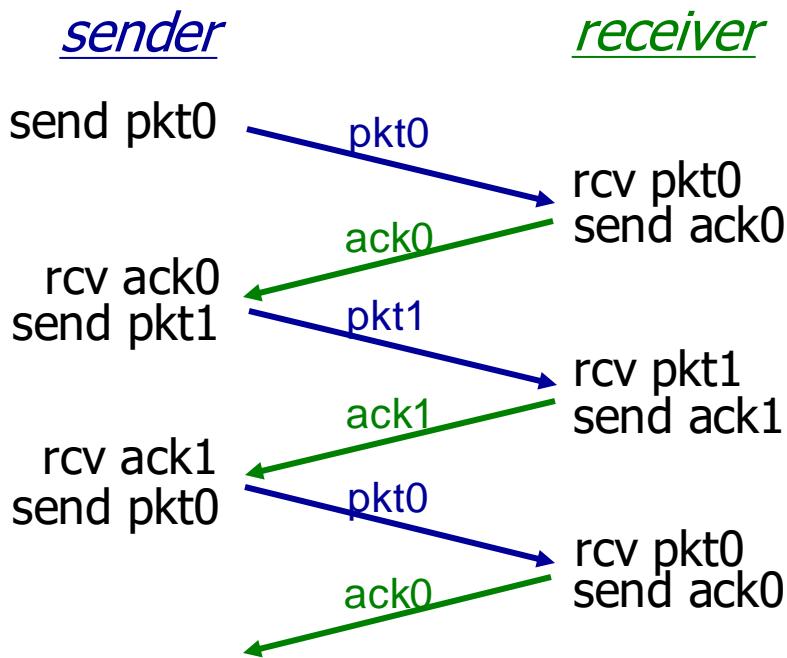
“reasonable” amount of  
time for ACK

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

# rdt3.0 sender



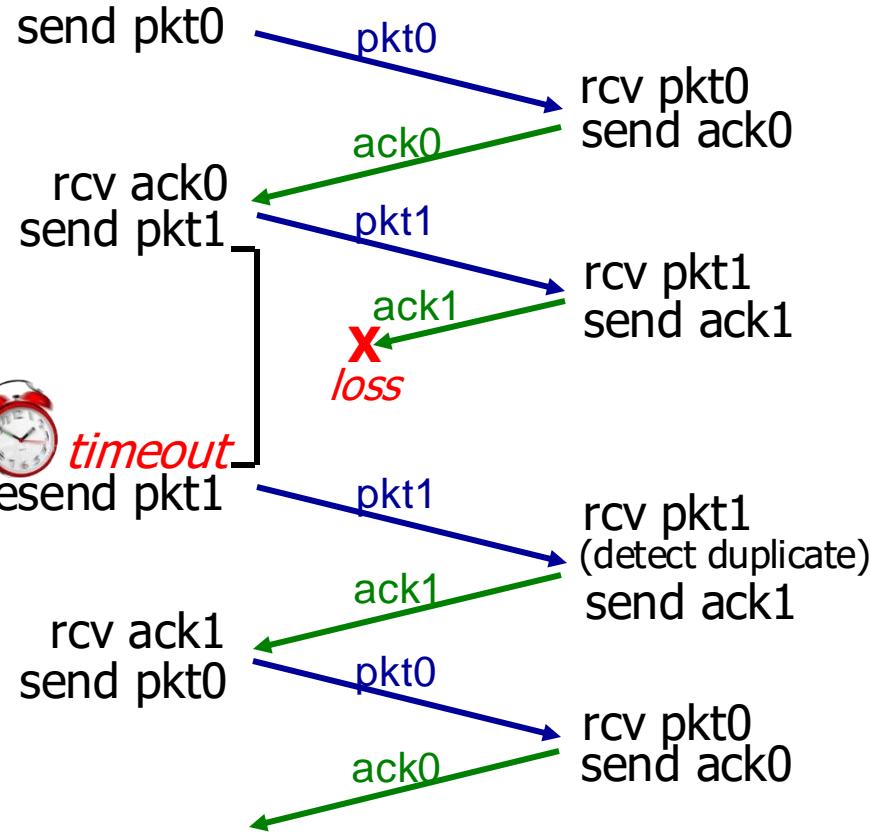
## rdt3.0 in action



(b) packet loss

## rdt3.0 in action

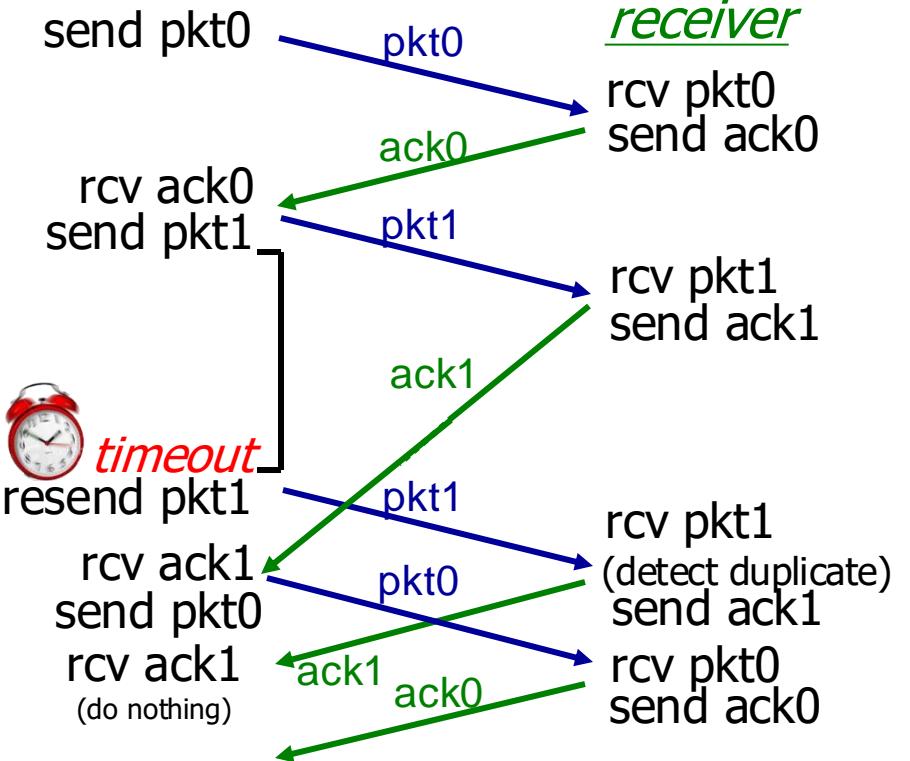
*sender*



(c) ACK loss

*sender*

*receiver*



(d) premature timeout/ delayed ACK

## Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

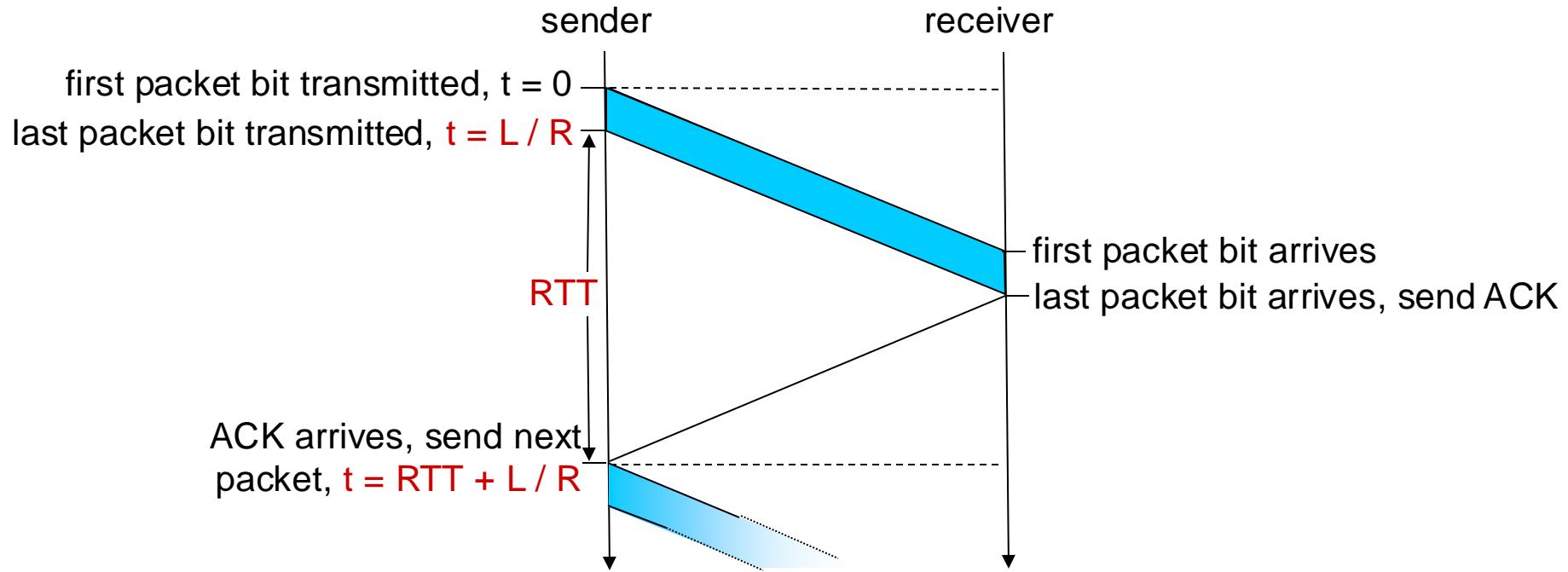
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- $U_{\text{sender}}$ : *utilization* – fraction of time sender busy sending

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- ❖ network protocol limits use of physical resources!

## rdt3.0: stop-and-wait operation



$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$