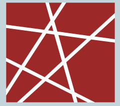
Master Course
Computer Networks
IN2097

Prof. Dr.-Ing. Georg Carle

Chair for Network Architectures and Services
Department of Computer Science
Technische Universität München
http://www.net.in.tum.de



Outline

- Measurements (cont.)
  - Flow Monitoring

- Transport Layer
  - Transport Layer Functions
  - UDP
  - TCP

IPFIX - IP Flow Information Export Protocol

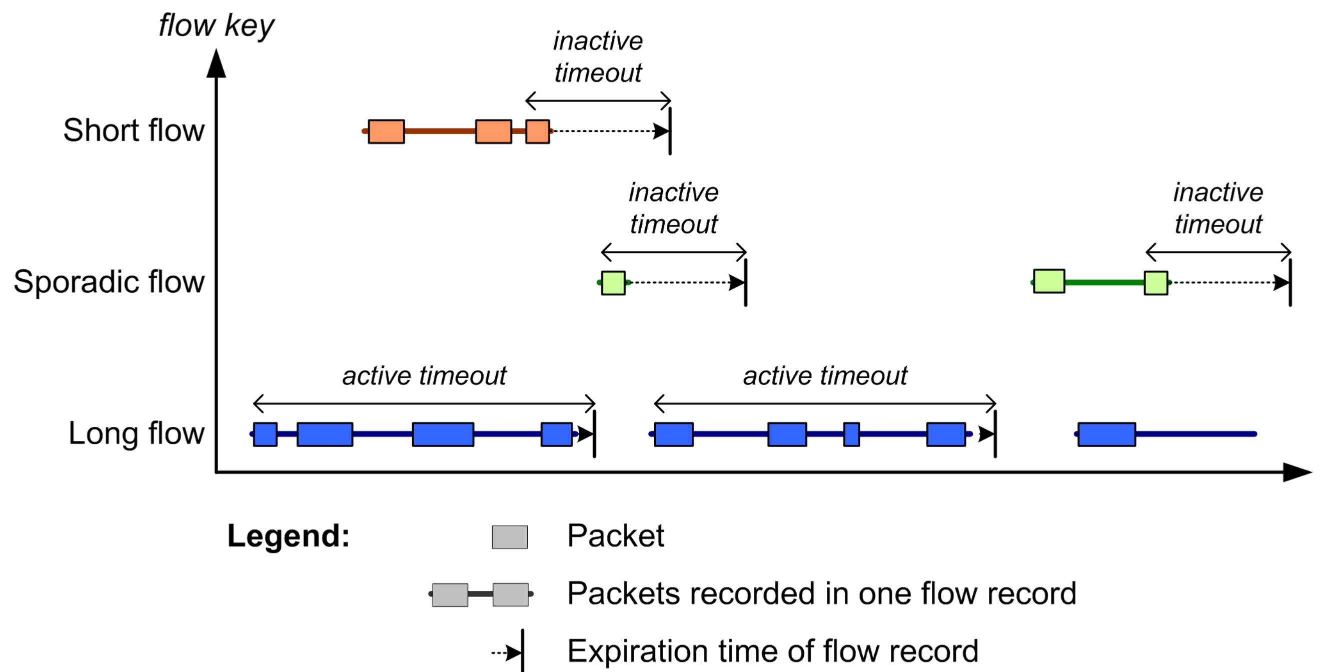
- RFCs
  - Requirements for IP Flow Information Export (RFC 3917)
  - Evaluation of Candidate Protocols for IP Flow Information Export (RFC 3955)
  - Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information (RFC 5101, **RFC 7011**)
  - Information Model for IP Flow Information Export (RFC 7012)
  - Guidelines for Authors and Reviewers of IP Flow Information Export (IPFIX) Information Elements (RFC 7013)
  - Bidirectional Flow Export using IP Flow Information Export (IPFIX) (RFC 5103)
  - IPFIX Implementation Guidelines (RFC 5153)
- Information records
  - **Template Record** defines structure of fields in **Flow Data Record**
  - Flow Data Record is a data record that contains values of the Flow Parameters

IPFIX: IP Flow Information Export

- IPFIX (IP Flow Information eXport) IETF Working Group
  - Standard track protocol based on Cisco Netflow v5…v9
- Goals
  - Collect usage information of individual data flows
  - Accumulate packet and byte counter to reduce the size of the monitored data
- Approach
  - Flows are represented by IP 5-tuple (protocol, srcIP, dstIP, srcPort, dstPort)
  - For each arriving packet, statistic counters of appropriate flow are modified
  - Whenever a flow is terminated (TCP FIN, TCP RST, timeout), its record is exported
  - Sampling algorithms can reduce the # of flows to be analyzed
- Transport protocol: transport of information records
  - SCTP must be implemented, TCP and UDP may be implemented
  - SCTP should be used
  - TCP may be used
  - UDP may be used (with restrictions – congestion control!)

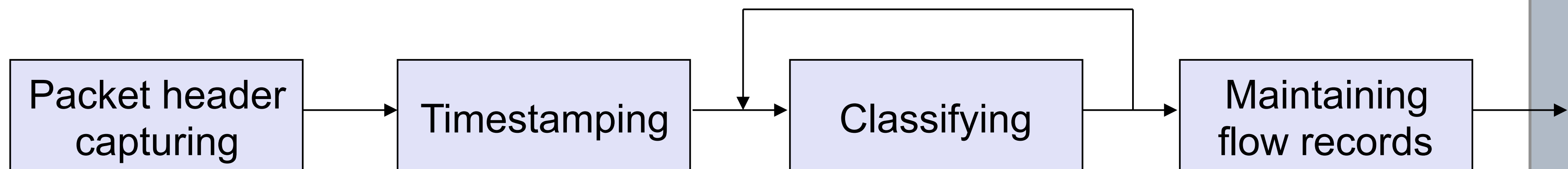
Flow-based Traffic Measurements

- **Export timeouts** (*flow expiration*)
  - *inactive timeout* ➔ export at the end of flow
  - *active timeout* ➔ export periodically for long-lived flows
  - timeouts can be configured (e.g., granularity ~60 sec)



IPFIX – Terminology

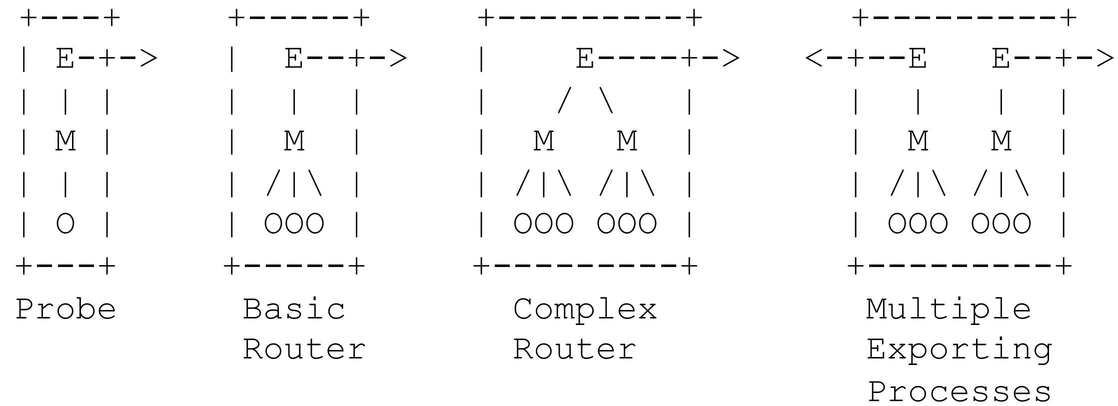
- IP Traffic Flow
  - A flow is defined as a set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties.
- Observation Point
  - The observation point is a location in the network where IP packets can be observed. One observation point can be a superset of several other observation points.
- Metering Process
  - The metering process generates flow records. It consists of a set of functions that includes packet header capturing, timestamping, sampling, classifying, and maintaining flow records.

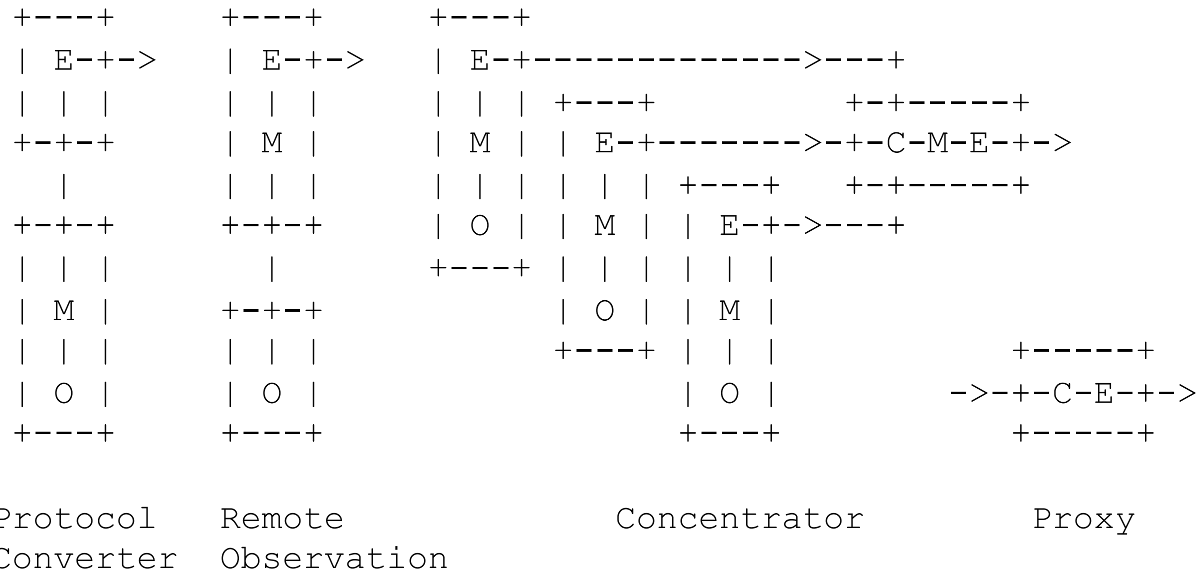


IPFIX – Terminology II

- Flow Record
  - A flow record contains information about a specific flow that was metered at an observation point. A flow record contains measured properties of the flow (e.g. the total number of bytes of all packets of the flow) and usually also characteristic properties of the flow (e.g. the source IP address).
- Exporting Process
  - The exporting process sends flow records to one or more collecting processes. The flow records are generated by one or more metering processes.
- Collecting Process
  - The collecting process receives flow records from one or more exporting processes for further processing.

IPFIX – Devices

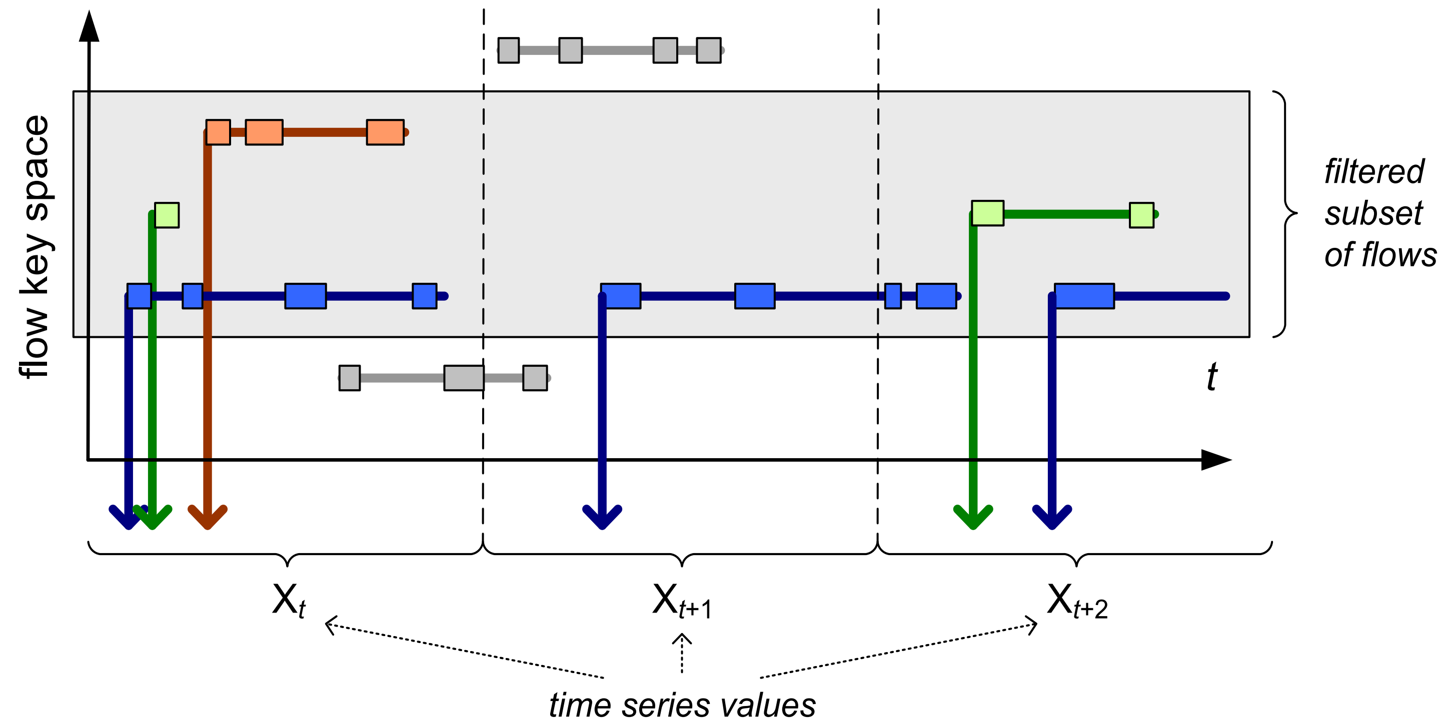




O ... Observation point
M ... Metering process
E ... Exporting process

Generation of Time Series from Flow Data (2)

- Aggregation according to start times



- Metrics to describe traffic volume
  - # Bytes, # packets, # flows

- Description of Flow-Key distributions

Generation of Time Series from Flow Data (2)

- Aggregation according to flow start time stamps

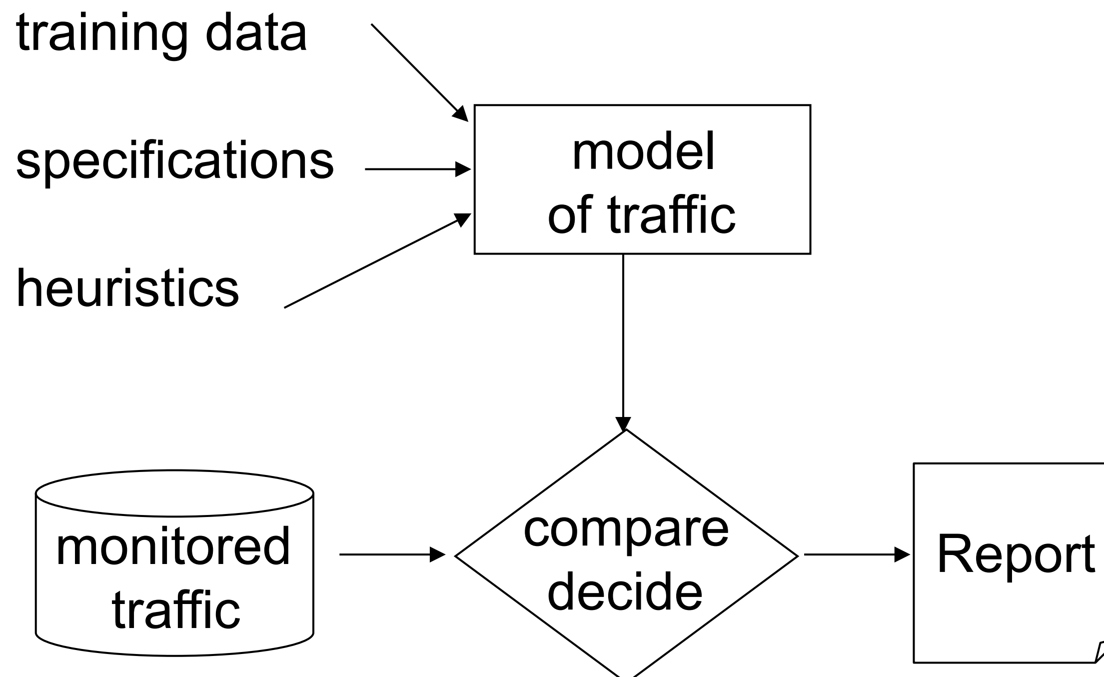
Start	End	Source-addr.	Dest.-addr.	Prot.	Source-port	Dest.-port	# of bytes	# of packets
...	...	...	...	...	...	...	...	...
34:44	34:59	10.0.1.5	10.0.1.7	6	3458	80	11043	22
34:45	35:04	10.0.1.7	10.0.1.5	6	80	3458	83921	35
35:07	35:15	10.0.1.4	10.0.1.7	6	58312	25	73849	44
...	...	...	...	...	...	...	...	...
39:58	39:10							
40:00	40:13	10.0.1.5	10.0.1.7	6	3458	80	11043	22
...	...	...	...	...	...	...	...	...

5 minutes interval

- Advantage
  - Amount of data independent of traffic volume, and traffic characteristics

Detecting Anomalies

- Modeling phase
  - derive model of normal traffic
- Analysis phase
  - compare observed traffic with normal traffic
  - report significant deviation from normal traffic
    (issues: percentage of false positives, false negatives; root cause)



Passive Measurements: Problems and Limitations

- **Passive monitoring can provide information on different levels**

Level	Information
Full Packets	Deep Packet Inspection, Payload Analysis, Session Analysis (limited to unencrypted traffic)
Packet Headers	Traceback, Delay Measurements, Statistical Traffic Classification
Flows	Host Activity Analysis, Dependency Analysis, Detection of Scans, Attacks, Traffic Anomalies
Traffic Aggregates	Accounting, Traffic Matrix Analysis, Detection of Traffic Anomaly and Network Outage

- **Encryption can make payload analysis impossible**
  - ⇨ But: Information can be obtained from the end systems
  - ⇨ Challenge: Information from host systems must be trustworthy

Cisco Netflow Implementations

- **Catalyst 6500 and Cisco 7600**
  - EARL6/EARL7 ASICs
  - 128k or 256k NetFlow TCAM
    (→ content-addressable memory)
  - hash tables: 16K or 32K rows with 8 elements
  - performance: 30 Mpps
- **Superior Engine 720**
  - EARL8 ASIC
  - 512k NetFlow TCAM
  - performance: 60 Mpps
- **Application Extension Platform (AXP)**
  - Intel x86 Processor (Core 2 Duo 1.86GHz)
  - RAM: 512 MB to 4 GB
  - HDD: 2 GB (flash) to 1 TB

Implementation of PSAMP / IPFIX

- Example: Vermont IPFix software, c.f.

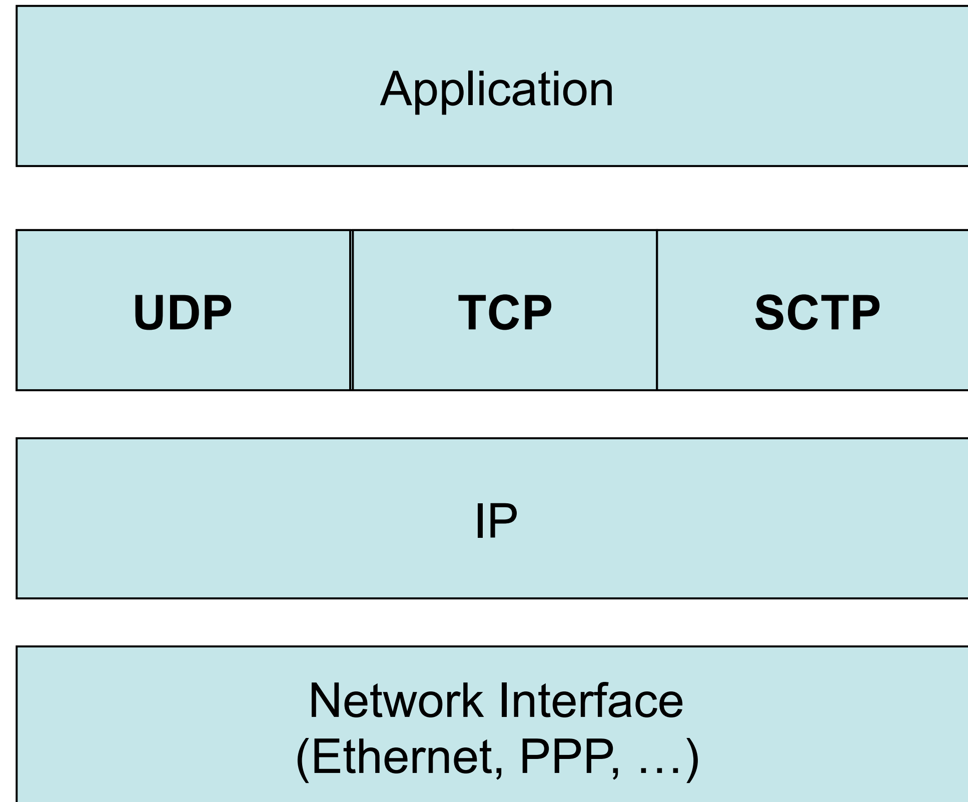
  http://www.history-project.net/index.php?show=vermont

  https://github.com/constcast/vermont/wiki

Transport Layer

Internet Transport-layer Protocols

- Unreliable, unordered delivery: UDP
  - simple extension of “best-effort” IP
  - no connection establishment (which can add delay)
- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup



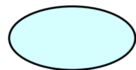
Multiplexing/Demultiplexing

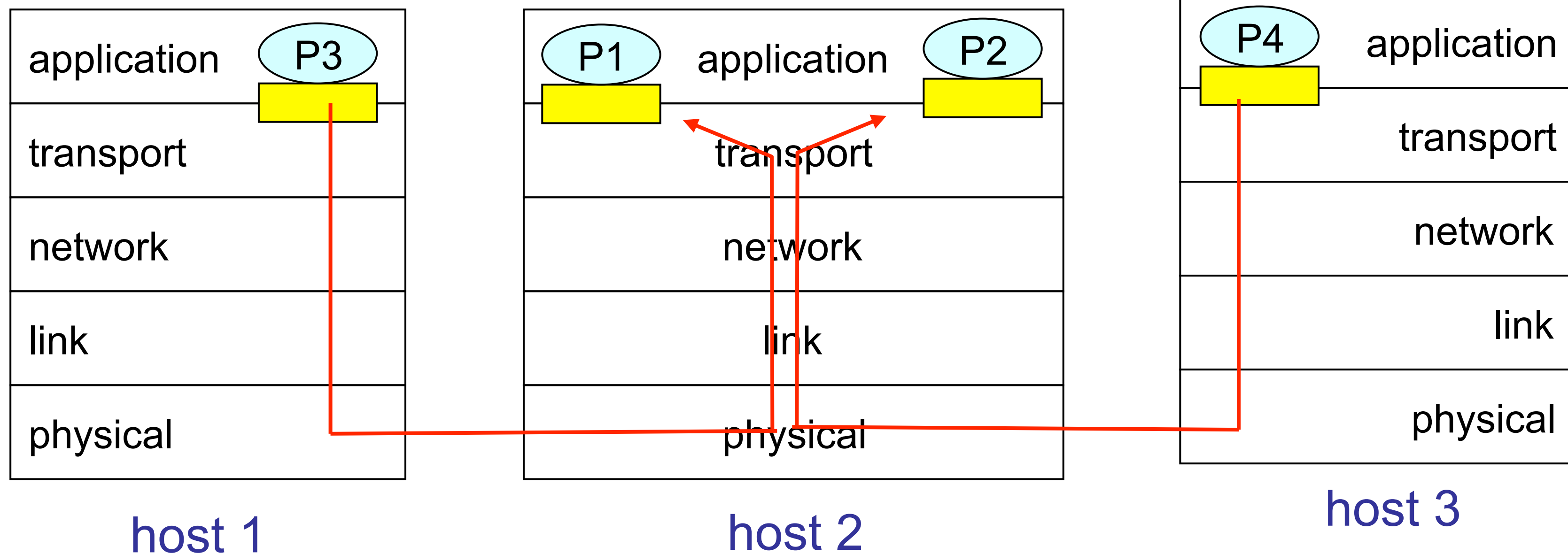
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket  = process



Connectionless Demultiplexing

- Create sockets with specific port numbers:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

- UDP socket identified by two-tuple:

  (dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with

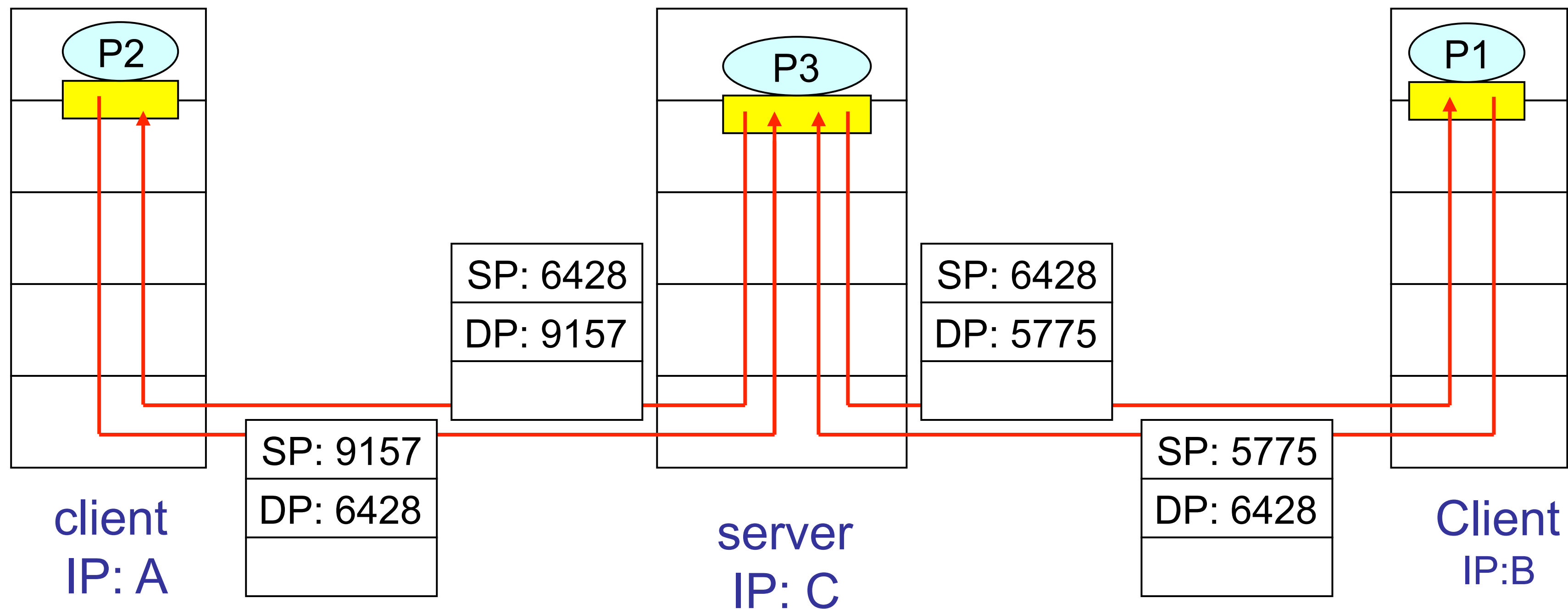
  different source IP addresses and/or

  different source port numbers

  directed to same socket

Connectionless Demultiplexing

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

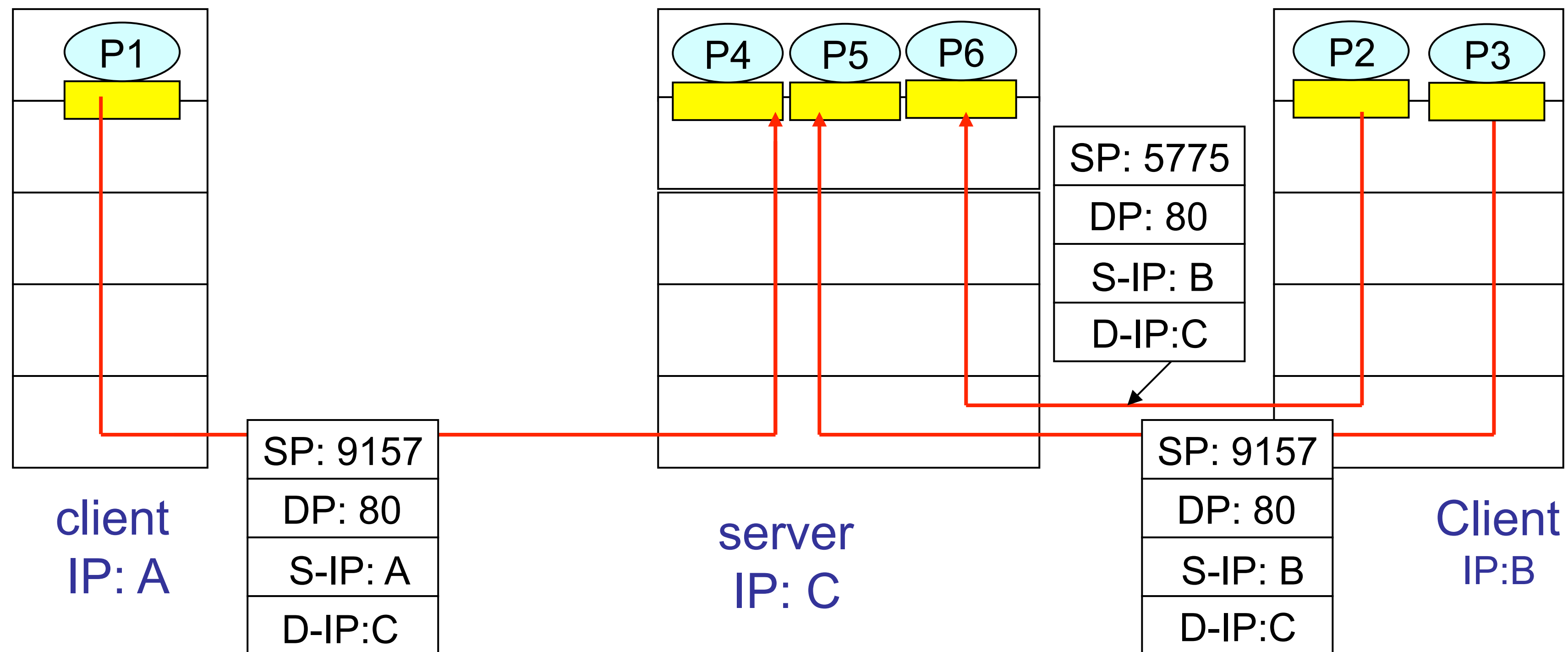


Source Port (SP) provides "return address"

Connection-Oriented Demultiplexing

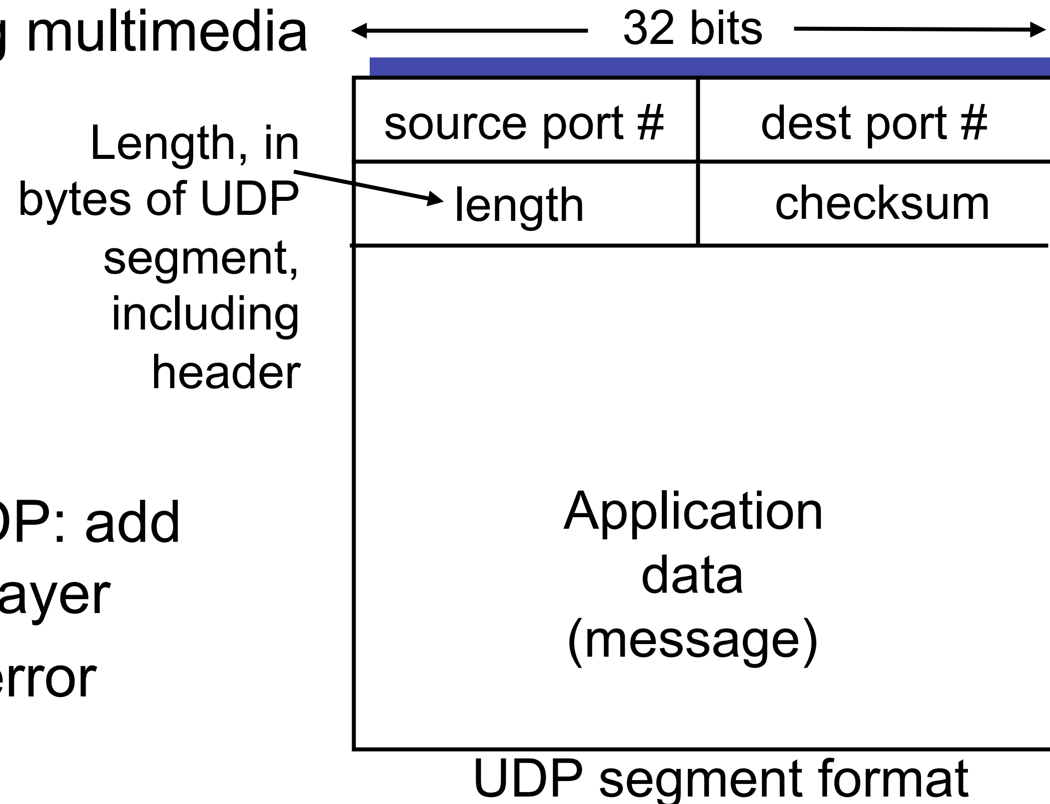
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Receiving host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP (TCP connection closed after transfer of requested object – c.f. HTTP 1.0) have different socket for each request
    vs. persistent HTTP (c.f. HTTP 1.1)

Connection-Oriented Demultiplexing



UDP: User Datagram Protocol [RFC 768]

- often used for streaming multimedia applications
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - ⇨ application-specific error recovery



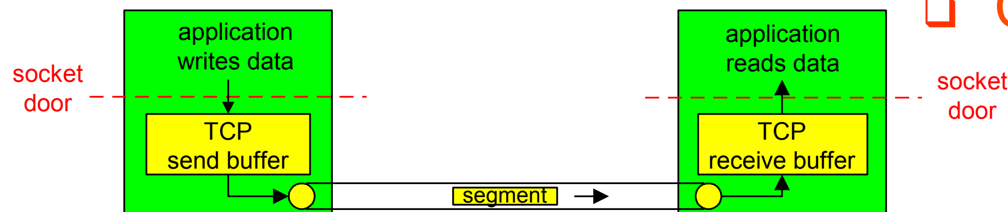
UDP segment format

- UDPlite: RFC 3828, Protocol Number 136
  - allows a potentially damaged data payload to be delivered to an application rather than being discarded
  - “Checksum Coverage” field instead of length field
    Minimum coverage: 8 byte (UDPlite header)

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*



- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver
- Congestion controlled:
  - Will not overwhelm network

TCP Header – c.f. RFC 793

0									16	31
Source Port									Destination Port	
Sequence Number										
Acknowledgement										
4 bit TCP header length	6 bit unused	URG	ACK	PSH	RST	SYN	FIN		Window	
Checksum									Urgent Pointer	
Options (0 or more 32bit words)										
Data ...										

Packet Header

TCP Header Options:

Maximum Segment Size (MSS) announcement (DSL, PPPoE ⇨ MSS Clamping)
Window Scaling, c.f. RFC 1072, 1323: 16 (orig.) + 14 = 30 bits window size
Selective Acknowledgements (SACK), c.f RFC 1072, 1323
Timestamps, c.f. RFC 1323
Nop

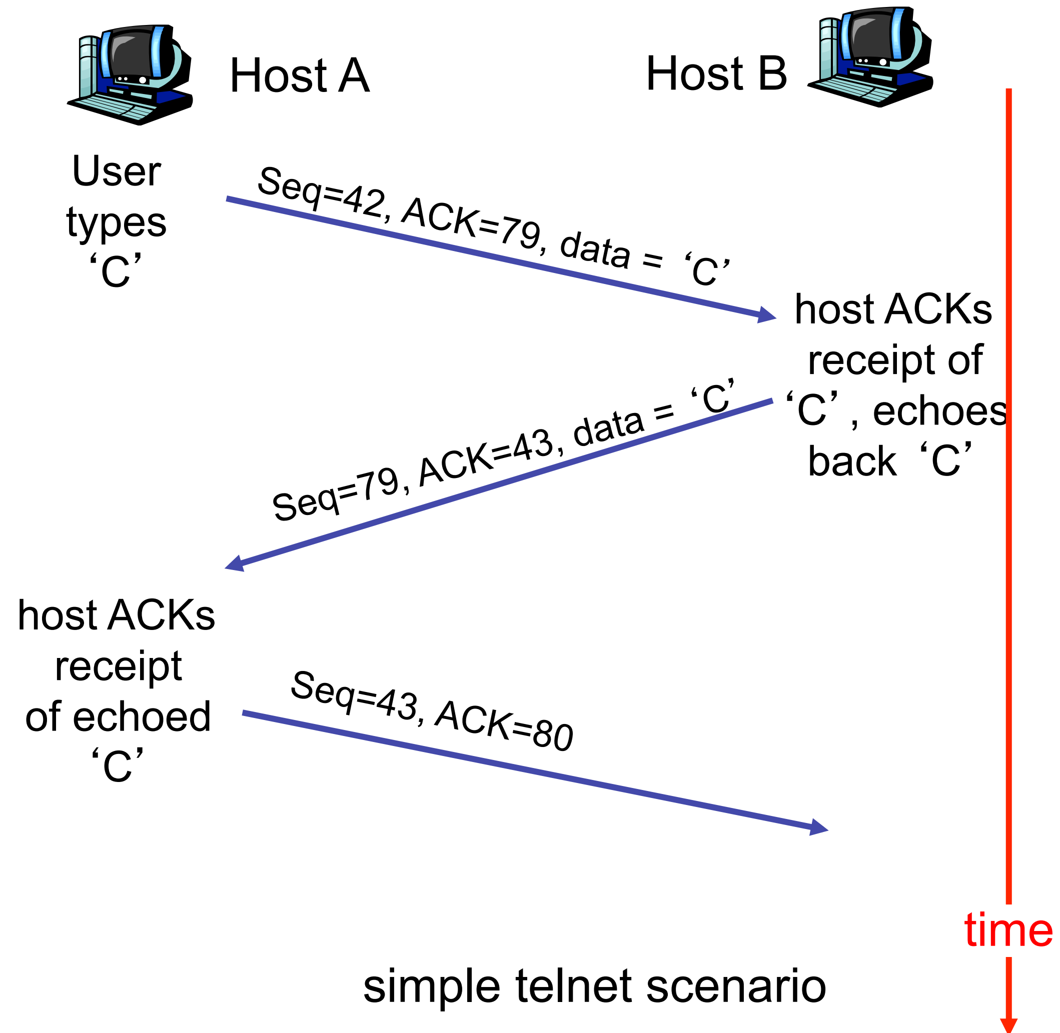
TCP seq. #’s and ACKs

Seq. #’s:

- byte stream “number” of first byte in segment’s data
- SYN and FIN flags increment Seq.#

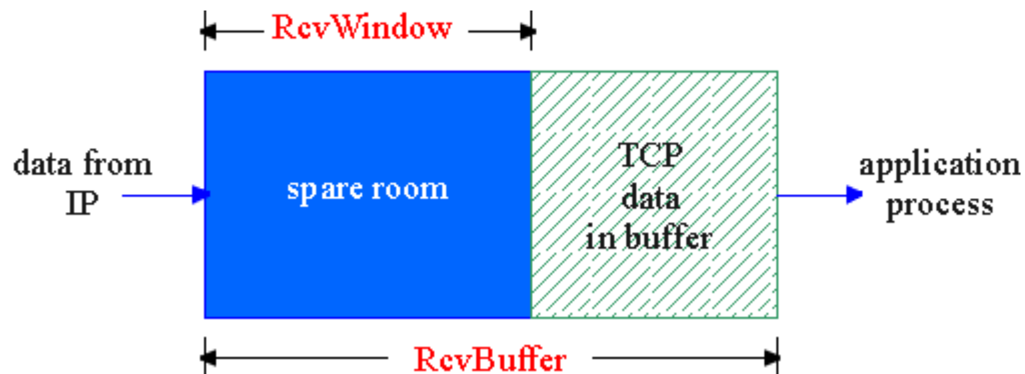
ACKs:

- seq # of next byte expected from other side
- cumulative ACK



TCP Flow Control

- Receive side of TCP connection has a receive buffer:



- Application process may be slow at reading from buffer
- Speed-matching service: matching the send rate to the receiving application's drain rate

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

- spare room in buffer
  = `RcvWindow`
  = `RcvBuffer-[LastByteRcvd - LastByteRead]`
- Rcvr advertises spare room by including value of **`RcvWindow`** in segments

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

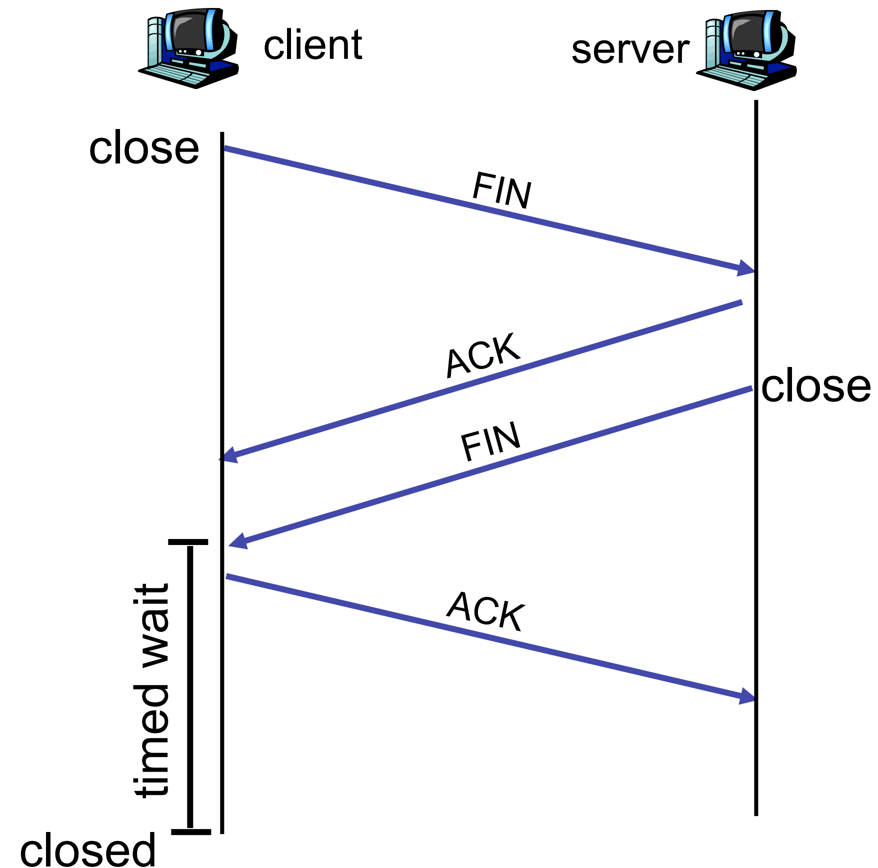
TCP Connection Management (cont.)

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.

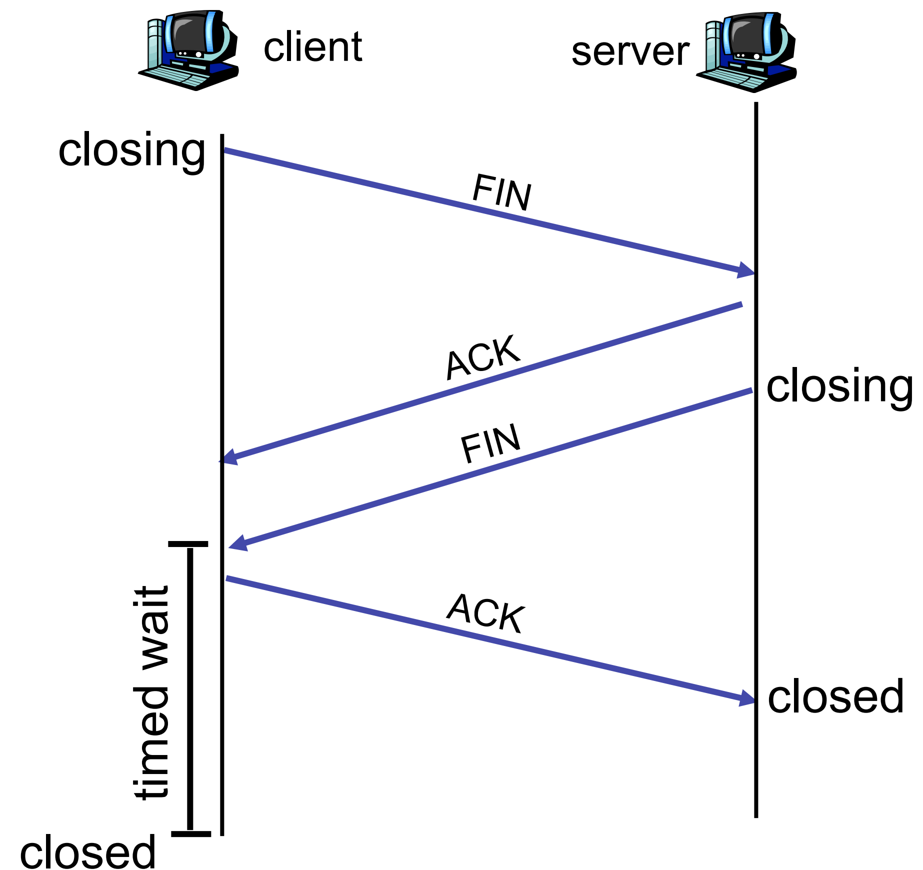


TCP Connection Management (cont.)

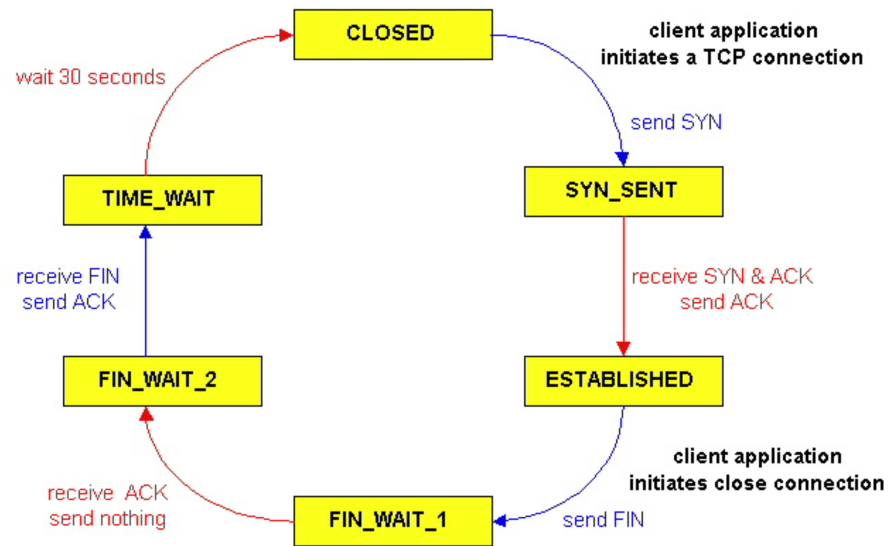
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

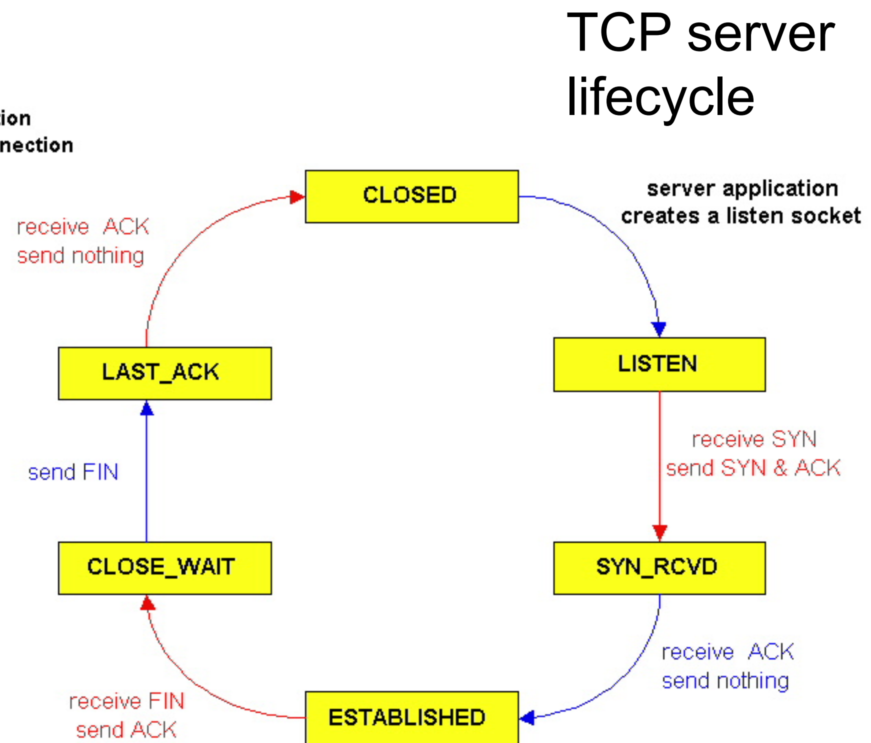


TCP State Machine – c.f. RFC 793



TCP client lifecycle

TCP server lifecycle



TCP Round Trip Time and Timeout

- Goal: estimate RTT for seting TCP timeout
- `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`
- EstimatedRTT=$(1-\alpha)$*EstimatedRTT +α*SampleRTT
  - exponential weighted moving average
  - influence of past sample decreases
    (more weight on recent samples than on older samples)
  - typical value: $\alpha = 0.125$

Example RTT estimation:

TCP Round Trip Time and Timeout

Setting the timeout

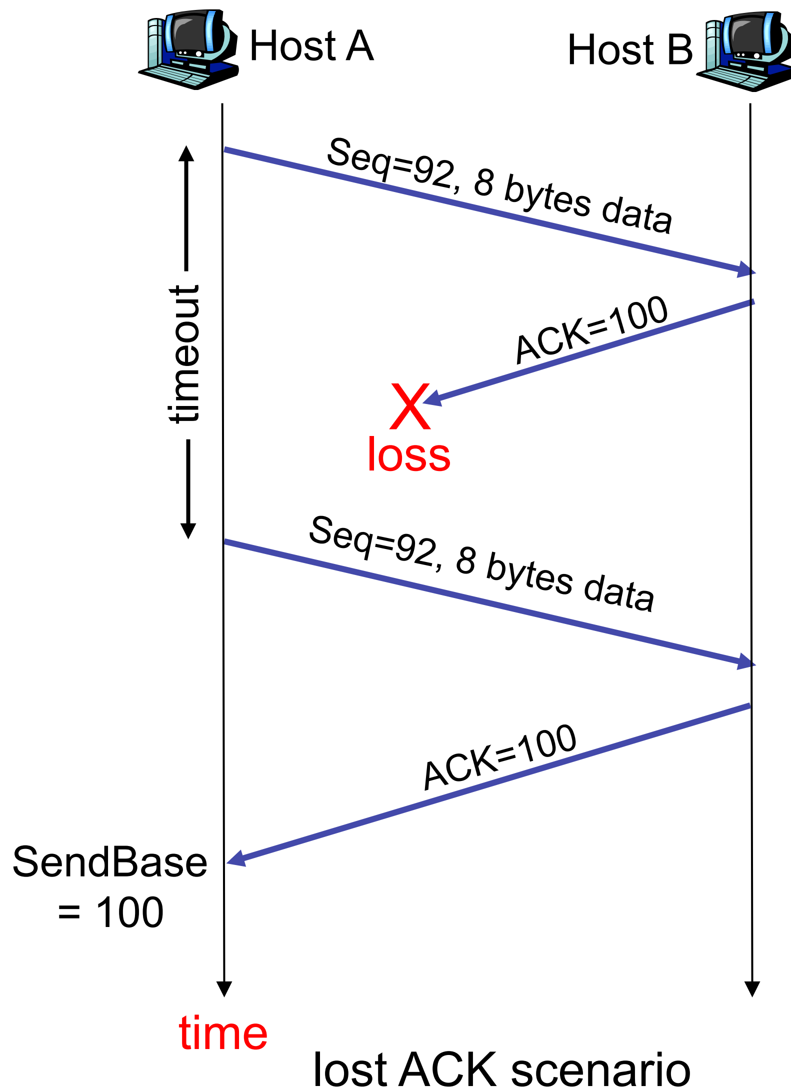
- **`EstimtedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT ->`** larger safety margin
- Estimate of how much SampleRTT deviates from EstimatedRTT (DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT)

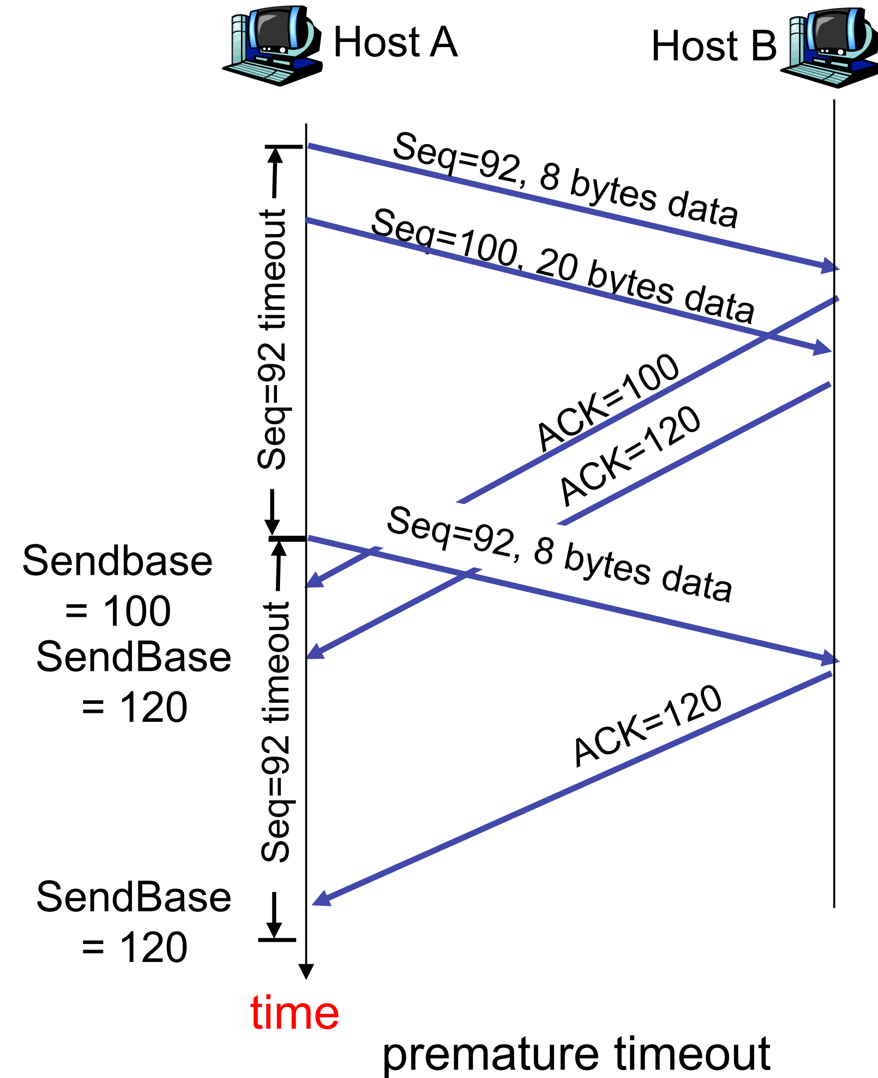
```
DevRTT = (1-β)*DevRTT +
                β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

- Set timeout interval

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP: Retransmission Scenarios





TCP ACK generation [RFC 1122, RFC 2581]

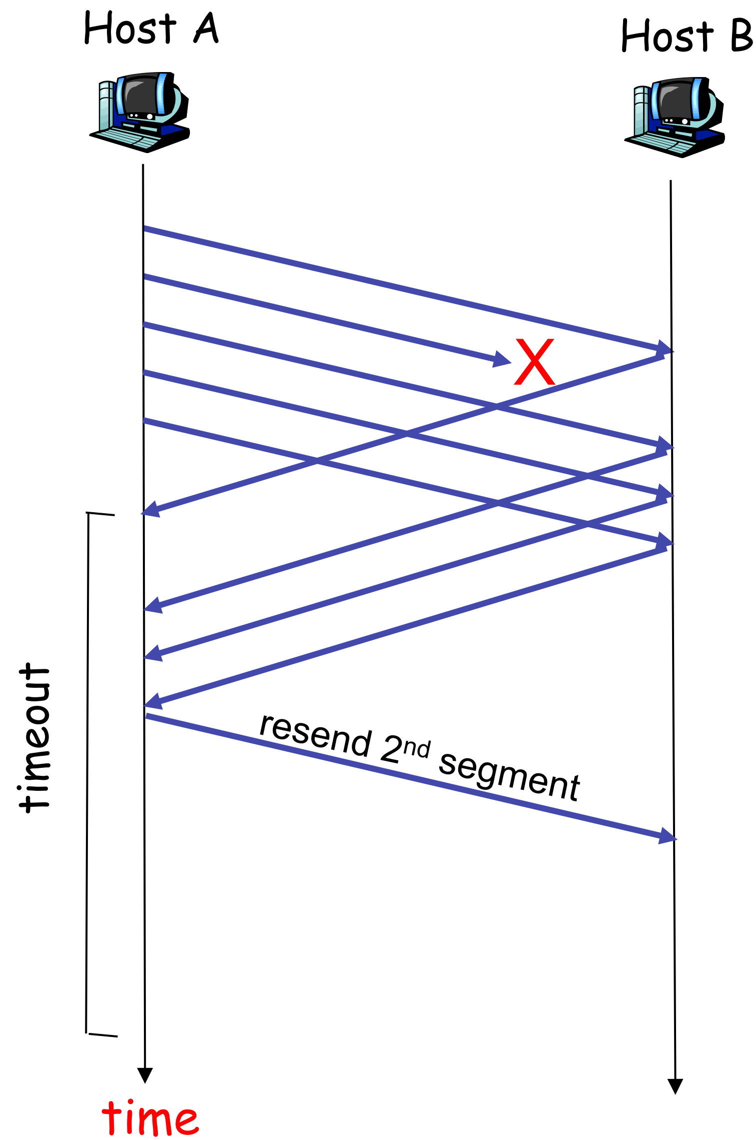
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte (which is lower end of gap)
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmit

- 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
  - If segment is lost, there likely will be many duplicate ACKs

- If sender receives 3 ACKs (i.e., "triple duplicate ACK") for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

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